Master's Thesis

Data-Driven Ordering and Dynamic Pricing Competition on Online Marketplaces

Carsten Walther
carsten.walther@student.hpi.de

Hasso Plattner Institute for IT Systems Engineering
Enterprise Platform and Integration Concepts Chair

August-Bebel-Str. 88
14482 Potsdam, Germany
https://hpi.de/plattner

Supervisors:

Prof. Dr. Hasso Plattner
Dr. Matthias Uflacker
Dr. Rainer Schlosser
Martin Boissier

Hasso Plattner Institute
Potsdam, Germany

May 14, 2018

Abstract

Online marketplaces become increasingly competitive. Merchants frequently and automatically adjust offer prices to get an advantage over the competition. At the same time, merchants have to estimate demand to prevent understocking and overstocking. Finding optimal pricing and ordering strategies is challenging because demand is uncertain, sales are affected by competitors' prices, and pricing and ordering decisions are mutually dependent. Merchants' pricing and ordering decisions determine their profitability and economic viability.

We utilize dynamic programming to derive optimized pricing and ordering policies with the objective to maximize long-term profit. Demand is estimated based on historical market data using demand learning. Furthermore, we implemented a merchant that uses the proposed optimization approach. The merchant was tested and evaluated on the Price Wars platform. We compared performance of merchant strategies in duopoly and oligopoly simulations. As soon as a sufficient amount of sales observations is available, our merchant outperforms traditional rule-based merchants.

Contents

List of Figures

List of Tables

1

Introduction

Price updates on today's online markets increase in number and happen in shorter intervals. Merchants get an advantage by adjusting their prices to competitors' prices. Frequently updating prices is called dynamic pricing and it is used by merchants to increase profit compared to traditional pricing strategies. Online marketplaces are ideal environments for dynamic pricing strategies because price changes are cheap and automatable. Software agents have faster response times and do not need to rest compared to human agents. Additionally, these programs are able to analyze huge amounts of historical market data and can estimate customer buying behavior from it.

Another important task of a merchant is keeping track of the inventory. If the inventory level is too low, the merchant might miss potential sales due to a stock-out. But storing too many items causes high holding costs. This is the inventory control problem or ordering problem and it is about when to order how many items. It is crucial to accurately predict future demand for good ordering decisions.

Managing prices and inventory is difficult because pricing and ordering decisions influence each other. The optimal price depends not only on external factors like competitors' prices but also on the available inventory. When inventory is low, it could be a good decision to increase the price in order to reduce the demand. If the merchant reduces prices, the demand will increase, which must be considered in the order decision. Because of this mutual influence, ordering and pricing should be decided jointly.

Throughout this thesis, we make the following contributions:

- We derived an optimization model for joint pricing and ordering problems (Section 5.2). Optimal pricing and ordering policies are calculated with dynamic programming.

- We estimate customer demand based on historical market data using demand learning (Section 5.3).
- We extended our model to cope with competing merchants (Section 5.4).
- We improved computation time efficiency of our dynamic programming approach to allow quick merchant reactions to new market situations (Section 5.4.3).
- We implemented a merchant that utilizes our optimization models and runs on the Price Wars platform (Section 5.5).
- We made numerical studies to show the applicability of our solution (Sections 5.1.3, 5.1.4, 5.2.3 and 5.3.2).
- We benchmarked merchant strategies in duopoly (Section 5.4.6) and oligopoly scenarios (Section 5.4.7). Our merchant consistently outperforms traditional rule-based merchants.
- Further, simulations showed that undercutting competitors is advantageous but only with sporadic price raises to increase the overall price level (Sections 5.4.6 and 5.4.7).

The Price Wars platform is an open-source framework to simulate dynamic pricing competition on online marketplaces. We used the platform to test and evaluate the performance of our merchant strategy. An extension of the platform was necessary to be able to simulate joint pricing and ordering problem scenarios on it. We implemented a more realistic ordering and inventory control concept on the Price Wars platform. This was done by extending the platform by orders of multiple items, fixed order costs, inventory holding costs, and order delivery delay. Tests on the platform are comparable and reproducible and since the platform is open-source software, researchers and practitioners can test their merchant strategies in the same or similar problem scenarios.

Chapter 2 discusses literature that relates to this thesis. Chapter 3 defines our ordering and pricing problem in detail. Chapter 4 introduces the Price Wars platform and describes its components. Chapter 5 presents our approaches to the joint pricing and ordering problem under competition and evaluates results of our solutions. Chapter 6 describes extensions made to the Price Wars platform. Chapter 7 concludes this thesis and proposes future work.

2

Related Work

Merchants on online marketplaces have to deal with two groups of decisions. They decide when to order products and how many. This is what the ordering problem is about. Additionally, merchants decide offer prices for their products. The problem of a suitable price choice is the pricing problem.

Ordering Problem

The ordering problem, also called inventory control problem, has been studied for a long time. An overview about this problem for deterministic and stochastic demand is given by Scarf [35]. Harris [24] proposed a solution for the ordering problem with constant deterministic demand and Wagner and Within [44] proposed a solution under deterministic but varying demand.

If demand is uncertain, future demand must be estimated from past sales. One approach is to investigate specific classes of parameterized demand distributions and propose methods to find parameters, so that the demand distribution fits the experienced sales best [5]. Other publications propose methods to find ordering policies without assumptions about the underlying demand distribution [6, 25, 28]

Besbes and Muharremoglu [8] study the effect of access to information about missed sales on the necessity to explore inventory decisions.

Pricing Problem

Each retailer has to face the problem of choosing a selling price. Talluri and van Ryzin [41], Phillips [33], and Yeoman and McMahon-Beattie [46] provide an extensive overview about solution approaches for this topic.

If it is allowed to frequently update prices, the problem is also called dynamic pricing problem. Gallogo and van Ryzin [22] proposed an optimal solution if demand is exponentially distributed.

In real-world applications, demand is typically uncertain, but can be estimated based on past sales. Den Boer [17] surveys literature about this topic. Different approaches to deal with uncertain demand are Bayesian estimation [4], maximum likelihood estimation [12], and least squares estimation [27] among others.

Joint Ordering and Pricing Problem

Joint ordering and pricing is a combination of both problems and is a common problem in retail. One challenge is that ordering and pricing decisions affect each other. Elmaghraby and Keskinocak [20] review literature about the joint ordering and pricing problem. Solutions are proposed for different problem scenarios, if demand is known [16, 21, 39, 42].

If demand is uncertain, it must be estimated from past market observations. Bitran and Wadhwa [10] and Bisi and Dada [9] propose Bayesian based approaches for the ordering and pricing problem with uncertain demand. Adida and Perakis [1, 2] study this problem in a multi-product scenario without backordering.

Pricing and Ordering under Competition

Chen and Chen [15] provide an overview about the dynamic pricing problem under competition for single-product and multi-product scenarios. Schlosser and Boissier [36] proposed a method to find optimal pricing policies if competitor strategies are known. Dynamic pricing problems under competition with a finite time horizon have been studied [29, 37]. Adida and Perakis [3] consider joint pricing and inventory control in a duopoly.

Simulation Platforms for Pricing Competition

For testing and evaluating merchant strategies, we use the Price Wars platform [38], a framework to simulate dynamic pricing competition on online marketplaces. We chose this platform over other solutions (e.g., [18, 31, 34]) because its continuous time model makes the platform similar to real online marketplaces like Amazon. Additionally, users are unrestricted in their choice and implementation of merchant strategies.

3

Problem Description

This thesis investigates the combined problems of dynamic pricing and inventory control on an online marketplace under competition. Merchants order durable products from a producer with unlimited supply. They can request the current market situation and update prices on the marketplace. Prices can be changed over time but cannot be adjusted to different customers. Merchants sell a single product type, but solutions for this problem are easy to extend to multiple products with independent demand. There are no back orders. If a merchant runs out of stock, it might miss on sales. Merchants sell products over an infinite time horizon.

The merchants' costs consist of fixed and variable order costs and inventory holding costs. Fixed order costs are comparable to shipping fees and are the same per order, independent from the order size. Variable order costs are the cost per ordered item. Merchants pay holding costs for storing products in their inventory. The longer an item remains in the inventory, the higher is its holding cost. The only source of income for a merchant are sales on the marketplace.

Inventories have no capacity limit. However, sensible order strategies only use a limited capacity in order to save holding costs.

The objective is to find optimized ordering and pricing policies that maximize the expected profit. A difficulty is that merchants do not know the customer behavior. In order to adapt to the customer behavior, merchants have to observe customer actions and estimate the behavior based on these actions. Merchants can request information about the current market situation at any time. However, a merchant gets only data about own sales but not about competitor sales.

4

Price Wars Platform

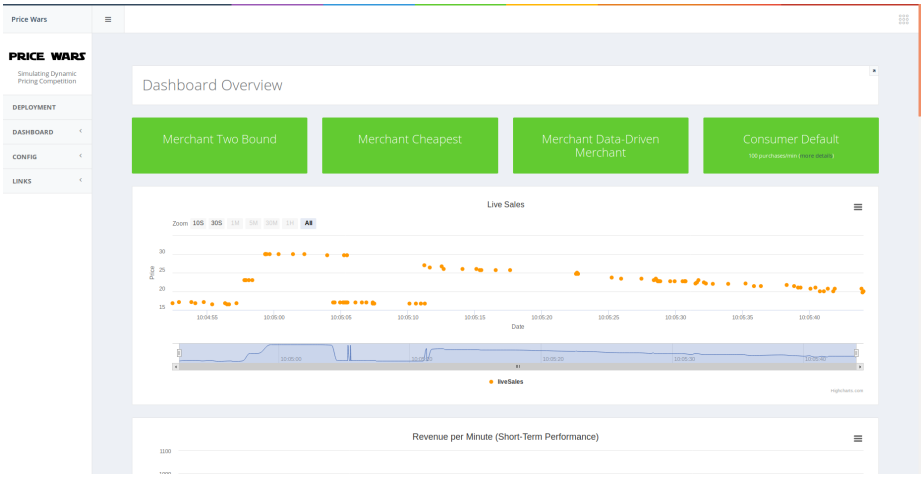

Figure 4.1. Screenshot of the Price Wars dashboard showing participating merchants, consumers, and a sales chart. Outside the screenshot are additional charts that visualize total profit per merchant, profit per minute, revenue per hour and minute, and market share.

The Price Wars platform is an open-source framework to simulate dynamic pricing competition on online marketplaces [11, 38]. Users can register their merchant on the platform to participate on the marketplace. The platform is a sandbox environment, which can be used to test and evaluate pricing strategies under different conditions. Developing and testing own pricing strategies can be time-consuming and possibly costly on a real online marketplace. The platform provides an HTML-based web user interface (UI) that allows users to configure and interact with the simulation. For example, they can explore how merchants react to a sudden increase in demand during the simulation.

The web UI includes a dashboard that visualizes price trajectories and several metrics, e.g., revenue and profit over time. The dashboard allows to observe evolution of the market and compare status and actions of competing merchants.

Formerly, merchants on the Price Wars platform focused solely on making suitable pricing decisions. We added features to the platform that also cause the ordering problem for the merchants. The additions are explained in Chapter 6.

4.1 Motivation

Testing new merchant strategies is time-consuming and potentially costly when done in production. The Price Wars platform is a tool to test merchant strategies in a sandbox environment and to study how strategies interact under competition. It is easy to set up and simulate different scenarios. Data-driven merchants are supported by providing datasets of past sales and market situations. The Price Wars platform is used for teaching purposes at the Hasso Plattner Institute.

4.2 Services and Agents

The Price Wars platform consists of multiple services and agents, communicate over RESTful APIs with each other. Services provide the platform's functionality and agents interact with the platform. Agents are merchants and consumers. Merchants and consumers can be added and removed while the platform is running. An overview of the architecture is presented in Figure 4.2. Each service and agent is explained in detail below:

Marketplace The marketplace is this platform's central service. Merchants use it to offer their products and consumers buy products on it. The marketplace manages merchant and consumer accounts. As a first step, merchants and consumers need to register on the platform, before they can perform authenticated actions like offering or buying products. Both, merchants and consumers can see all open offers on the marketplace. Merchants can adapt their prices according to competitors' prices with this information. Consumers inspect these offers to find their preferred offer. Each event that happens on the marketplace is written to a logging service. This data is processed to, e.g., calculate a merchant's revenue. Whenever a product has been sold, the marketplace notifies the selling merchant.

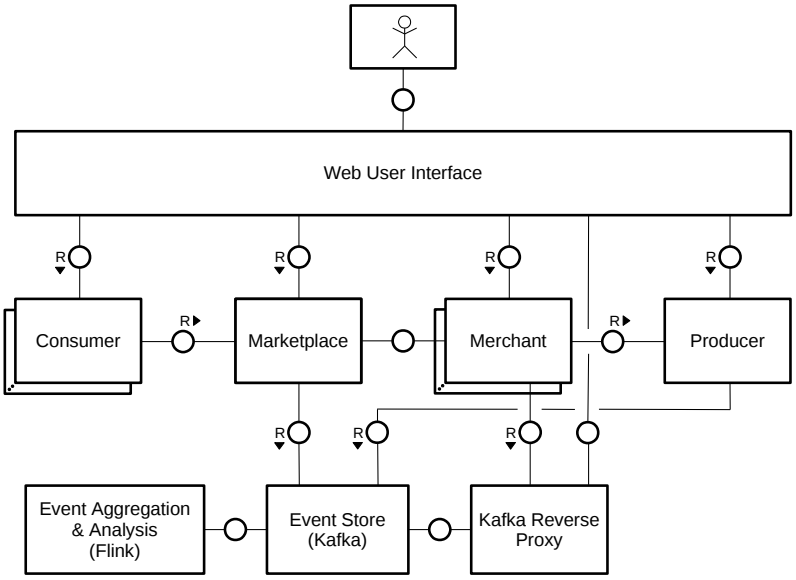

Figure 4.2. Architecture of the Price Wars platform in FMC notation [26].

Merchant Merchants sell products by offering them on the marketplace at a desired price. They can request open offers from the marketplace to adapt prices to competitors' offer prices. A merchant's pricing strategy can be a simple rule-based strategy like "always undercut the cheapest competitor" or a complex data-driven strategy that analyses the consumer behavior and/or competitors' strategies. Of course, also complex rule-based strategies or a hybrid of both approaches are possible. The Price Wars platform supports data-driven merchants by providing historical market and sales data. A merchant can be written in any programming language as long as the implementation complies with the platform's RESTful API. Documentation and an example merchant implementation is available on Github[1] to help users building merchants with custom strategies.

Consumer The consumer agent creates a stream of consumers who visit the marketplace, inspect available offers, and buy products. A consumer, that does not find any acceptable offers, leaves the marketplace without buying a product. The consumer agent implements different buying behaviors, which can be enabled, disabled, or mixed together in the web UI.

[1] Price Wars Merchant documentation and source code of its Python implementation on Github: `https://github.com/hpi-epic/pricewars-merchant`

Producer A merchant can restock his inventory with new products from the producer. Products can be of varying quality. Merchants pay a certain amount of money per product. This amount is specified by the producer.

Web User Interface The web UI allows the user to control and observe the marketplace simulation. Marketplace, merchants, consumer, and producer can be configured using the web UI. With this level of control, it is possible to test how merchants react to changing market conditions. For example, it can be tested how fast a merchant adapts the behavior if the consumer rate doubles. A dashboard contains charts that visualize merchant and consumer actions as well as merchants' short- and long-term profit and revenue. Figure 4.1 is a screenshot of the dashboard.

Event Processing Events on the platform are processed by three separate services *Apache Kafka*, *Kafka Reverse Proxy*, and *Apache Flink*.

Apache Kafka [23] is a stream database, which is used to store events that happen on the platform. Such events are for example orders from the producer, sales, and new offers on the marketplace.

Kafka Reverse Proxy supports data-driven merchants with data about past market situations and sales. The data is filtered – a merchant gets only information about his own sales – and transformed into the CSV format. Additionally, the web UI gets continuous updates for its charts from the Kafka Reverse Proxy.

Building onto Kafka, Apache Flink [13] is used to process and aggregate events to metrics like inventory level or profit per hour.

Additionally, the marketplace uses the databases Postgres [40] and Redis [14] to manage registered accounts, product offers, and rate limiting.

4.3 Deployment

Documentation and source code of the Price Wars platform is available on Github: `https://github.com/hpi-epic/pricewars`. The service-specific repositories provide instructions on how to set the services up. Each service can be deployed on a dedicated machine this way. Alternatively, the Price Wars platform can be deployed using docker [30]. This allows an easy setup on a single machine, which is useful to try the platform out and to develop on it. One command `docker-compose up` is enough to set up and run the whole Price Wars platform.

5

Dynamic Pricing & Ordering Optimization Models

This chapter presents dynamic optimization models for dynamic pricing and ordering problems. The goal is to find merchant strategies that maximize expected long-term profits. The setting is an online marketplace in which our merchant competes with other merchants in an oligopoly.

We estimate demand based on historical market data using demand learning with linear regression. The estimated demand is used in dynamic programming to generate optimized pricing and ordering policies.

A pricing policy defines a pricing decision for every situation a merchant can be in. Similar, an ordering policy defines an ordering decision for every such situation. In our case, pricing and ordering decisions depend on the current inventory level and on the current market situation.

We developed the optimization model in four phases to overcome the complexity of joint pricing and ordering problem under competition. The first phase in Section 5.1 is constrained to the ordering problem with known customer demand in a monopoly. Each following phase adds a new difficulty the merchant must manage. Section 5.2 describes the second phase, which adds free choice over the offer price. The merchant's chosen price will affect sales. In the third phase (Section 5.3), demand is no longer known and the merchant has to find ways to estimate future demand. The fourth and last phase (Section 5.4) represents the final problem: The joint ordering and dynamic pricing problem with uncertain customer behavior under competition. Section 5.5 describes our merchant implementation for the Price Wars platform.

5.1 Ordering Problem in Monopoly

The merchant focuses on the ordering problem in this scenario and makes ordering decisions that promise the most expected long-term profit. The merchant cannot make pricing decisions, instead all products are sold at a fixed price. Ordering decisions depend on the customer demand. The demand is stochastic and its distribution is known to the merchant. There are no backlogs. If a customer arrives while the merchant is out of stock, the merchant will miss the sale. The merchant operates in a monopoly and does not have to care about competitors.

5.1.1 Model of Ordering Problem

The merchant sell items over the marketplace. Items are offered at a fixed price a_{fix}, $a_{fix} > 0$, and each sold item generates the full offer price a_{fix} as revenue. We use discounting to increase the relevance of short-term profits. A discount factor δ, $0 < \delta \leq 1$, is applied to each time period. The time horizon is infinite. The merchant makes exactly one order decision in each discrete time period.

The merchant holds items in an inventory. Storing items in the inventory causes holding costs of l per item per time period, $l > 0$. The random inventory level at the start of period t is denoted by N_t, $t = 0, 1, \ldots$

The merchant can reorder items to increase the inventory level. The number of items ordered at time t are denoted by b_t, $b_t \geq 0$. An order of size $b_t = 0$ means that no order is made. The set of admissible orders quantities is denoted by B, $B \subseteq \mathbb{N}$. Each order causes order costs, which consists of fixed order costs c_{fix}, $c_{fix} \geq 0$, and variable order costs c_{var}, $c_{var} \geq 0$. Order decisions influence the merchant's profit based on ordering costs, holding costs, and future potential sales. Order costs are defined by, $b \in B$,

$$C(b) := \begin{cases} c_{fix} + c_{var} \cdot b & \text{if } b > 0 \\ 0 & \text{if } b = 0 \end{cases} \tag{5.1}$$

The probability to sell i items within one period of time, given enough items are available, is denoted by $P(i)$, $i = 0, 1, \ldots$. Seasonal demand effects are out of scope of this work. However, they can be easily included by making the probability function $P(i)$ depend on the time period t. The sales probability in the first phase does not directly depend on the price or market situation because the merchant sells at a fixed price in a monopoly. The random number of sold items from start of period $t-1$ to start of period t is denoted by X_t.

The merchant cannot sell more items than the inventory holds, i.e. $N_t \geq X_{t+1}$, $t = 0, 1, \ldots$

Depending on a given ordering policy $(b_t)_t$, the random accumulated discounted profit from time period t is, $t = 0, 1, \ldots$

$$G_t := \sum_{s=t+1}^{\infty} \delta^{s-t} \cdot (a_{fix} \cdot X_s - l \cdot N_{s-1} - C(b_{s-1}(N_{s-1})))$$

The objective is to find an ordering policy that maximizes the expected total profit $E(G_0|N_0)$. A list of all variables and parameters with description is given in Table A.1.

5.1.2 Dynamic Programming Approach

Goal of this section is to derive optimal ordering policies for the ordering problem. We use a dynamic programming approach to find the ordering policy promises the best expected profit for the stochastic control problem. The value function $V_t(n)$, $t, n = 0, 1, \ldots$, describes the best expected discounted future profit $E_t(G_t|N_t = n)$. N_{max} denotes an upper limit of the inventory level n and the order decision b, $0 \leq n, b \leq N_{max}$. This does not affect the optimal solution as long as N_{max} is greater than the biggest inventory level that can occur in the optimal policy.

We limit the time horizon to the end time T. If T is sufficiently large, the value iteration over V_t converged, so that the optimal ordering policy is not affected. The start value u defines the result of the value function at the end of the time horizon, $V_T(n) = u$ for all n. For now, the start value is set to $u := 0$.

We consider $V_t(n)$ with *instantaneous* and *delayed* orders. With instantaneous orders, available items from the start of the time period and the newly ordered items can be sold within that period. The following time period starts with an inventory of items that were not sold in the previous period. With the inventory level n_t, order quantity b_t, and demand i_t at period t, the inventory transition from n_t to n_{t+1} is defined by, $t = 0, 1, \ldots$

$$n_t \to \max(n_t + b_t - i_t, 0) \tag{5.2}$$

With delayed orders, only items that are available from the start of the time period can be sold within that period. The following time period starts with an inventory of items that were not sold in the previous period plus the number of

items ordered in the previous period. With the inventory level n_t, order quantity b_t, and demand i_t at period t, the inventory transition from n_t to n_{t+1} for delayed orders is defined by, $t = 0, 1, \ldots$

$$n_t \to \max(n_t - i_t, 0) + b_t \tag{5.3}$$

The value function with an *instantaneous arrival* of orders is, $t = 0, 1, \ldots, T-1$, $n = 0, 1, \ldots, N_{max}$,

$$V_t(n) = \max_{b \in B} \left\{ \sum_{i=0}^{N_{max}} P(i) \cdot \Big(a_{fix} \cdot min(i, n+b) - l \cdot (n+b) - C(b) + \delta \cdot V_{t+1}\big(min(max(n+b-i, 0), N_{max})\big) \Big) \right\} \tag{5.4}$$

The value function with *order delivery delay* is shown in the following equation, $t = 0, 1, \ldots, T-1$, $n = 0, 1, \ldots, N_{max}$,

$$V_t(n) = \max_{b \in B} \left\{ \sum_{i=0}^{N_{max}} P(i) \cdot \Big(a_{fix} \cdot min(i, n) - l \cdot n - C(b) + \delta \cdot V_{t+1}\big(min(max(n-i, 0) + b, N_{max})\big) \Big) \right\} \tag{5.5}$$

The set of order quantities B is a discrete set of integers and must contain zero to allow the merchant to make no order. The optimal ordering decision $b^*(n)$ is given by using arg max on (5.4) and (5.5).

That is for instantaneous orders, $n = 0, 1, \ldots, N_{max}$,

$$b^*_{instant}(n) = \arg\max_{b \in B} \left\{ \sum_{i=0}^{N_{max}} P(i) \cdot \Big(a_{fix} \cdot min(i, n+b) - l \cdot (n+b) - C(b) + \delta \cdot V_1\big(min(max(n+b-i, 0), N_{max})\big) \Big) \right\} \tag{5.6}$$

and for delayed orders, $n = 0, 1, \ldots, N_{max}$,

$$b^*(n) = \arg\max_{b \in B} \left\{ \sum_{i=0}^{N_{max}} P(i) \cdot \Big(a_{fix} \cdot min(i, n) - l \cdot n - C(b) + \delta \cdot V_1\big(min(max(n - i, 0) + b, N_{max})\big) \Big) \right\} \tag{5.7}$$

Equations (5.6) and (5.7) are similar to $V_0(n)$ but use arg max instead of max. The arg max is only used on the outermost term. Nested functions are normal value functions using max. Since (5.6) and (5.7) correspond to $V_0(n)$, the next nested value function is $V_1(n)$.

Equations (5.6) and (5.7) yield optimal policies to the ordering problem without and with order delay, respectively. The duration of the order delay in (5.7) is exactly the length of one period. It is possible to find optimal ordering policies for arbitrary order delays using this dynamic programming approach. However, this greatly increases the dynamic programming state and results in long computation times. In order have reasonable decision times for the merchant, we use the presented approach, which is a heuristic for order delays different from the length of the time period and otherwise an exact solution. The following phases will only consider the variation with delayed orders.

5.1.3 Numeric Example

This section shows explicit results of (5.6) and (5.7) for parameters given by the following example.

Example 1. Let the parameters be $N_{max} = 40$, $a_{fix} = 35$, $c_{fix} = 30$, $c_{var} = 20$, $l = 0.4$, $T = 500$, $\delta = 1$, $B = \{0, 1, \ldots, 40\}$, $t = 0, 1, \ldots, T-1$, $u = 0$. The length of a period is one second. Sale probabilities $P(i)$ are taken from Table 5.1

The time between arriving customers is exponential distributed with a mean time of one second. We simulated this process to get the following sales probabilities.

i	0	1	2	3	4	5	6	7	other
$P(i)$	0.189	0.316	0.261	0.146	0.061	0.020	0.006	0.001	0.0

Table 5.1. Probabilities to sell i items in one period, given enough items are available. The probabilities are based on a simulation of customer arrivals with exponential distributed waiting time between customers.

n	$b^*_{instant}(n)$	$b^*(n)$
0	17	18
1	16	18
2	0	17
3	0	16
other	0	0

Table 5.2. Ordering policies without and with order delivery delay for Example 1.

Using (5.6) and (5.7) results in ordering policies shown in Table 5.2. The merchant begins to order new items at a higher inventory level if orders are delayed. Additionally, the merchant restocks more items. This behavior reduces the risks of an stock-out until the ordered items arrive.

The optimal policy for the ordering problem with instantaneous order delivery falls into the class of (s, S) policies. Policies of this class are identified by two parameters. Parameter s is a threshold on the inventory level. If the inventory level falls below or to s, the merchant orders new items. The merchant orders as many items as needed to increase the inventory level to S. In our example, the resulting policy is a (1, 17) policy. The merchant orders if the inventory level is one or fewer and refills the inventory to 17 items. With delayed order delivery, the optimal policy is not a (s, S) policy because the inventory is refilled to different levels.

	no change	$l = 0.1$	$c_{fix} = 15$	$c_{var} = 27$	$a_{fix} = 55$
n	$b^*(n)$	$b^*(n)$	$b^*(n)$	$b^*(n)$	$b^*(n)$
0	18	35	14	17	19
1	18	34	13	17	19
2	17	34	13	16	18
3	16	33	12	0	17
4	0	32	11	0	17
other	0	0	0	0	0

Table 5.3. Effect of parameter changes on optimal ordering policy $b^*(n)$. Each column shows the results of $b^*(n)$ when changing one parameter. If not chosen differently, parameters are as in Example 1 with $l = 0.4$, $c_{fix} = 30$, $c_{var} = 20$, $a_{fix} = 35$.

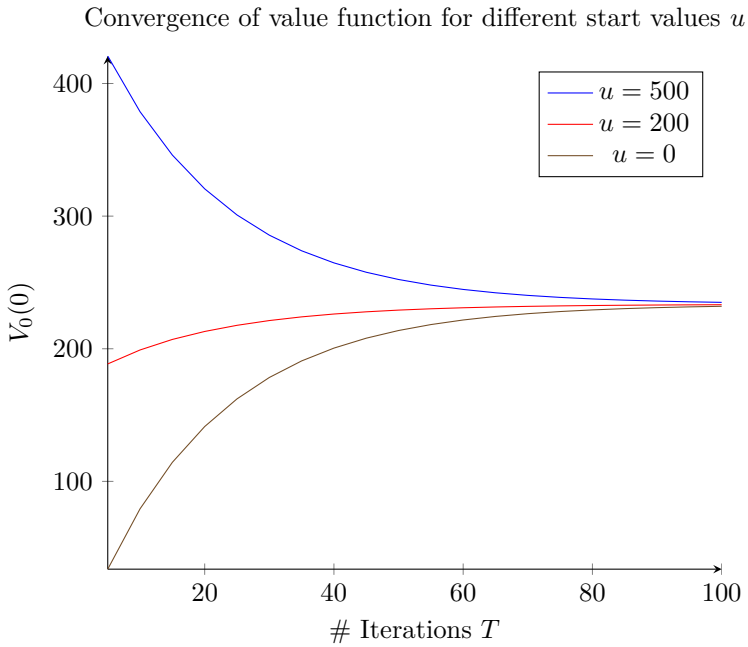

Figure 5.1. Influence of start value u on the final expected discounted profit V_0. The function converges with any start value to the same expected profit, given a sufficiently large T. However, V_0 converges faster if u is near the resulting expected discounted profit. $\delta = 0.95$. $T = 100$, other parameters as in Example 1.

Table 5.3 shows the effect of different parameters on $b^*(n)$. When holding costs decrease, the merchant makes bigger orders because it is less costly to hold these items in the inventory. The merchant buys less items per order if fixed order costs are lower. This results in more frequent orders and an on average lower inventory level, which saves holding costs. Higher variable order costs reduce the merchant's profit margin. The merchant orders only in few different sizes that promise the best trade-off between holding costs and order costs. The merchant risks stock-outs to avoid high order costs. If variable order costs are too high, the stops ordering completely. At a higher selling price, the merchant orders more products and refills the inventory earlier. The loss of potential revenue becomes more important than the additional holding costs.

5.1.4 Effect of Start Value on Convergence of Value Function

In Section 5.1.2, we used an arbitrary start value $u := 0$ and claimed that the value function $V_0(n)$, $n = 0, 1, \ldots, N_{max}$, converges to the optimal expected profit if T is sufficiently large. In the following example, we show that the value function converges to the same value independently from u.

Figure 5.1 shows results of $V_0(0)$ with varying T and start values u. The parameters are as in Example 1. The only differences are $\delta = 0.95$ and $T = 100$. The value function converges to the same result for any u. It does not matter if the start value is bigger or smaller than the resulting expected profit. However, the value function converges faster if u is near the optimal expected discounted profit. We will use this fact in later sections to speed up the policy computation.

5.2 Joint Ordering and Pricing Problem

In this section, the merchant is in the same situation as in Section 5.1 but additionally gains control over the selling price. With ordering and pricing decisions enabled, the merchant's actions on the marketplace are unconstrained. Demand in this scenario depends on the chosen selling price. A higher price typically results in less demand. As in the previous problem, the demand distribution is known.

We cannot separate the inventory and pricing problem and solve each in isolation to solve the whole problem optimally. Ordering and pricing decisions influence each other. Setting a price changes the demand, which requires a different ordering policy. The other way around, an ordering policy might require certain prices for a better control over inventory levels. This mutual influence is the reason, why ordering and pricing should be decided jointly. However, this is a more complex problem compared to solving the ordering problem and pricing problem separately.

5.2.1 Model of Ordering and Pricing Problem

This model is an extension of the model from Section 5.1.1. Instead of a fixed selling price a_{fix}, the merchant sets a price a_t in each period t, with $a_t \geq 0$, $t = 0, 1, \ldots, T-1$. The set of admissible pricing decisions is denoted by A, $A \subseteq \mathbb{R}_0^+$. Each sold item in period t generates a revenue of a_t. The pricing decisions a_t may depend on the current inventory level n_t.

The probability $P(i, a)$ to sell i items, $i = 0, 1, \ldots$, given enough items are available, depends now on price a. We assume that the merchant knows the price-dependent probability distribution in this scenario.

Depending on a given pricing and ordering policy $(a_t, b_t)_t$, which can depend on the current inventory level, the random accumulated discounted profit from time period t is, $t = 0, 1, \ldots$

$$G_t := \sum_{s=t+1}^{\infty} \delta^{s-t} \cdot (a_{s-1}(N_{s-1}) \cdot X_s - l \cdot N_{s-1} - C(b_{s-1}(N_{s-1})))$$

The objective is to find a joint pricing and ordering policy that maximizes the expected discounted total profit $E(G_0|N_0)$.

5.2.2 Extending the Dynamic Programming Approach by Pricing Decisions

As a basis, we use the dynamic programming approach shown in (5.5). The action space is extended by one dimension that contains all potential pricing decisions A. The resulting equation is, $t = 0, 1, \ldots, T-1$, $n = 0, 1, \ldots, N_{max}$

$$V_t(n) = \max_{\substack{a \in A \\ b \in B}} \left\{ \sum_{i=0}^{N_{max}} P(i,a) \cdot \Big(a \cdot min(i,n) - l \cdot n - C(b) + \delta \cdot V_{t+1}\big(min(max(n-i,0)+b, N_{max})\big) \Big) \right\} \tag{5.8}$$

There are three changes to the value function. The probability $P(i,a)$ is price-dependent, the revenue in one period depends on the chosen price $(a \cdot min(i,n))$, and the action space is extended by the set of pricing decisions A. The optimal ordering policy $b_t^*(n)$ and pricing policy $a_t^*(n)$ are given by using the arg max in (5.8), $n = 0, 1, \ldots, N_{max}$

$$(a^*(n), b^*(n)) = \operatorname*{arg\,max}_{\substack{a \in A \\ b \in B}} \left\{ \sum_{i=0}^{N_{max}} P(i,a) \cdot \Big(a \cdot min(i,n) - l \cdot n - C(b) + \delta \cdot V_1\big(min(max(n-i,0)+b, N_{max})\big) \Big) \right\} \tag{5.9}$$

Equation (5.9) is similar to $V_0(n)$ but uses arg max instead of max in the outermost term. Nested functions are normal value functions using max. Since (5.9) corresponds to $V_0(n)$, the next nested value function is $V_1(n)$.

n	$a^*(n)$	$b^*(n)$
0	-	5
1	29	4
2	29	0
3	29	0
4	28	0
5	28	0
6	27	0
7	27	0
8	27	0
9	26	0
10	26	0

Table 5.4. Optimal joint pricing and ordering policies for Example 2. The price decreases with increasing inventory level. $a^*(0)$ has no meaningful result because the merchant cannot offer items at any price if the inventory is empty.

5.2.3 Numeric Example

This section shows the explicit results of (5.9) for parameters specified in the following example.

Example 2. Let the parameters be $N_{max} = 10$, $c_{fix} = 5$, $c_{var} = 15$, $l = 0.5$, $T = 500$, $\delta = 1$, $A = \{0, 1, \ldots, 60\}$, $B = \{0, 1, \ldots, 10\}$. $P(i, a)$ is given by (5.11). Time periods have a length of one second.

For this example, we model sales probabilities with a Poisson distribution. The mean of potential sales per period $\lambda(a)$ is chosen to be highest if $a = 0$ and zero for $a \geq 40$. The probability function is, $0 \leq i \leq N_{max}, a \in A$,

$$\lambda(a) = \max(2 - 0.05 \cdot a, 0) \tag{5.10}$$

$$P(i, a) = \frac{\lambda(a)^i \cdot e^{-\lambda(a)}}{i!} \tag{5.11}$$

Using (5.8) and (5.9) results in policies shown in Table 5.4. The pricing policy sets decreasing prices with increasing inventory levels. This strategy reduces holding costs for inventories with many items by offering a lower price and selling items faster. If only few items remain in inventory, the policy uses higher prices

to increase the profit per sold product and increase the time until a new order is necessary.

The additional dimension of pricing decisions drastically increases the computation time for this problem compared to the previous model which considers only ordering decisions. The computation time increases by a factor of the number of pricing decisions. We present methods to reduce computation times of our dynamic programming approach in Section 5.4.3.

5.3 Demand Learning

This section extends the joint inventory and dynamic pricing problem from Section 5.2 by not providing the merchant with sales probabilities. The merchant needs to estimate the customer behavior in order to make effective ordering and pricing decisions. The merchant can get information about the customer behavior from past customer actions. The merchant can analyze historical market situation and sales in order to deduce future sales probabilities. This is called demand learning.

5.3.1 Estimating Sales Probabilities

In order to overcome the inventory and dynamic pricing problem with missing demand information, we split the whole problem into two parts. The first part is the training of a model from historical market situations and sales in order to predict sales probabilities. These probabilities are used in the second part to create optimized ordering and pricing policies. Given predicted sales probabilities, the dynamic programming solution from Section 5.2.2 can be used as it is, to calculate ordering and pricing policies.

Assuming, the merchant predicts sales probabilities correctly, the dynamic programming approach creates optimal policies. In other cases, the quality of policies will depend on the accuracy of predictions. The separation of demand prediction and policy creation lets us focus on the prediction of the demand probabilities, while having a working solution for the policy creation.

In order to use market and sales data for demand learning, it must be transformed into a suitable form. From the market data we know, at what times the market situation changed and what the market situation was. The sales data contains information about all of a merchant's sales, including when the sales happened. These two data sources can be combined and aggregated to a form that contains

the number of sales per time period for each market situation. The size of a period corresponds the period in which the merchant makes a order and pricing decision.

We use regression analysis for demand learning. The relationship between a dependent variable (sales per period) and explanatory variables (market conditions) in modeled and analyzed. Explanatory variables are extracted from a market situation $\vec{s}$. A market situation $\vec{s}$ is a vector of all open offers at one point in time. In this scenario, the demand depends only on the merchant's price. We use linear regression to find a relationship between the explanatory variables $\vec{x}(a, \vec{s})$ of a market situation $\vec{s}$ with pricing decision a and the sales per period for this situation. Linear regression is used to illustrate this approach and can be interchanged with other regression methods. The following equation shows how to predict mean sales per period for a market situation $\vec{s}$ with pricing decision a, $a \geq 0$.

$$\lambda(a, \vec{s}; \vec{\beta}) = \max(\vec{x}(a, \vec{s})^\intercal \cdot \vec{\beta}, 0) \tag{5.12}$$

$\vec{\beta}$ is a vector of weights for the regression model. $\vec{\beta} := (\beta_0, \ldots, \beta_{M-1})$, $\beta_i \in \mathbb{R}$, $i = 0, 1, \ldots, M-1$ and $\vec{x}(a, \vec{s}) \in \mathbb{R}^M$ and M is the number of explanatory variables. $\lambda(a, \vec{s}; \vec{\beta})$ has a lower bound of 0 to prevent negative mean sales. Given observed sales per period $y_1, y_2, \ldots, y_J$ from J historical market situations $\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_J$ with own prices $a_1, a_2, \ldots, a_J$, the optimal weights $\vec{\beta}^*$ are given by the linear least squares objective, $j = 1, 2, \ldots, J$,

$$\min_{\vec{\beta} \in \mathbb{R}^M} \sum_{j=1}^{J} (y_j - \lambda(a_j, \vec{s}_j; \vec{\beta}))^2 \tag{5.13}$$

The optimal weights $\vec{\beta}^*$ can be calculated with the following equation:

$$\vec{\beta}^* = \left(\sum_{j=1}^{J} \vec{x}(a_j, \vec{s}_j) \vec{x}(a_j, \vec{s}_j)^\intercal \right)^{-1} \left(\sum_{j=1}^{J} y_j \vec{x}(a_j, \vec{s}_j) \right) \tag{5.14}$$

The merchant is able to predict mean sales per period for a market situation with the regression analysis. However, sales probabilities are needed for the dynamic programming approach. The mean sales per period can be used as a parameter to create a suitable probability distribution. We describe the demand distribution with a Poisson distribution. This discrete distribution is a good approximation for the arrival and buying process of customers [45]. Equation (5.15) shows the calibration of the Poisson distribution with the linear regression from (5.12) in

order to obtain estimated sales probabilities for a market situation $\vec{s}$, $i = 0, 1, ...$, $a \in A$,

$$P(i, a, \vec{s}; \vec{\beta}^*) := e^{-\lambda(a,\vec{s};\vec{\beta}^*)} \frac{\lambda(a, \vec{s}; \vec{\beta}^*)^i}{i!} \tag{5.15}$$

Given the current market situation $\vec{s}$ and weights $\vec{\beta}$, $\vec{\beta} \in \mathbb{R}^{\mathbb{M}}$, the value function (5.8) is extended to the following function, $n = 0, 1, \ldots, N_{max}$,

$$\begin{aligned} V_t(n; \vec{s}, \vec{\beta}) = \max_{\substack{a \in A \\ b \in B}} \Bigg\{ \sum_{i=0}^{N_{max}} P(i, a, \vec{s}; \vec{\beta}) \cdot \Big(a \cdot min(i, n) - l \cdot n - C(b) \\ + \delta \cdot V_{t+1}\big(min(max(n - i, 0) + b, N_{max}); \vec{s}, \vec{\beta}\big)\Big) \Bigg\} \end{aligned} \tag{5.16}$$

For a fixed market situation $\vec{s}$ and fixed weights $\vec{\beta}$, $\vec{\beta} \in \mathbb{R}^{\mathbb{M}}$, optimal policies (cf. (5.9)) are derived by, $n = 0, 1, \ldots, N_{max}$,

$$\begin{aligned} (a^*(n), b^*(n)) = \arg\max_{\substack{a \in A \\ b \in B}} \Bigg\{ \sum_{i=0}^{N_{max}} P(i, a, \vec{s}; \vec{\beta}) \cdot \Big(a \cdot min(i, n) - l \cdot n - C(b) \\ + \delta \cdot V_1\big(min(max(n - i, 0) + b, N_{max}); \vec{s}, \vec{\beta}\big)\Big) \Bigg\} \end{aligned} \tag{5.17}$$

The merchant has access to a steadily growing collection of market situations and sales data over time. The quality of predictions is increased by periodic training that utilize the new data. There is no sales data available at the start of a merchant's career on the online marketplace. In that case, we use a rule-based ordering and pricing strategy until enough training data is available. The rule-based pricing strategy sets random prices to explore the customer behavior in different market situations. The exploration pricing strategy sets random prices. This allows the merchant to encounter many different market situations and obtain sales data for these situations.

Our explanatory variables $\vec{x}$ consist of the price a and the constant intercept 1. Other variations like $\sqrt{a}$ or a^2 are possible to recognize non-linear relations between demand and price. $\vec{x}$ will be extended by additional variables in Section 5.4. Additionally, the linear regression algorithm can be easily replaced

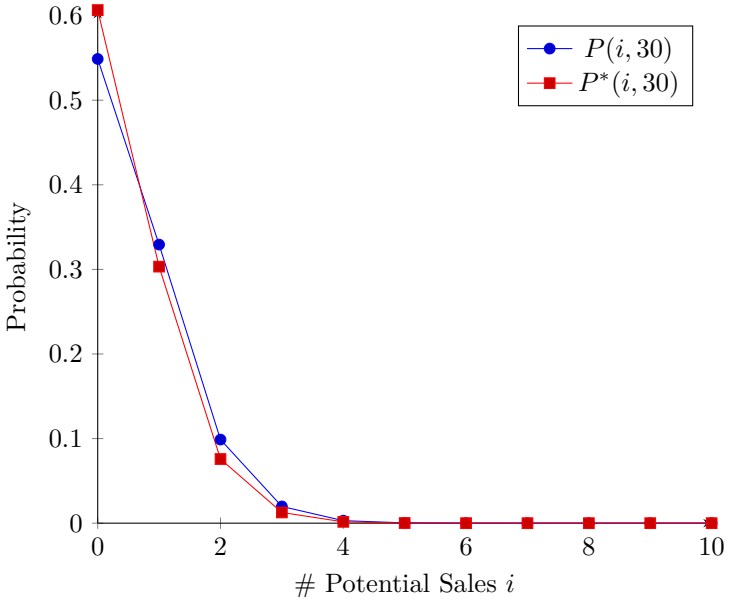

Figure 5.2. Comparison of predicted sales probabilities $P(i, a)$ with true underlying Poisson probabilities P^*, cf. (5.19), for a price of 30. The predictions are close to the true probabilities despite using only ten training examples scattered over a large price range.

by other regression approaches like decision tree regression or neural network regression.

5.3.2 Comparing Predicted and Actual Sales Probabilities

This is an experiment to evaluate how close predicted sales probabilities of our demand learning approach are to the true underlying sales probabilities. For the first example, the true demand function to sell i, $i \geq 0$, items in one period at price a, $a \geq 0$, is the following price-dependent Poisson distribution,

$$\lambda(a) = \max(2 - 0.05 \cdot a, 0) \tag{5.18}$$

$$P^*(i, a) = \frac{\lambda(a)^i \cdot e^{-\lambda(a)}}{i!} \tag{5.19}$$

We generated ten market situations as training data. Prices for those situations is chosen uniform randomly between 20 and 40. The sales per period for a market situation is a random sample from the distribution given by $P^*(i, a)$ and the price of that situation.

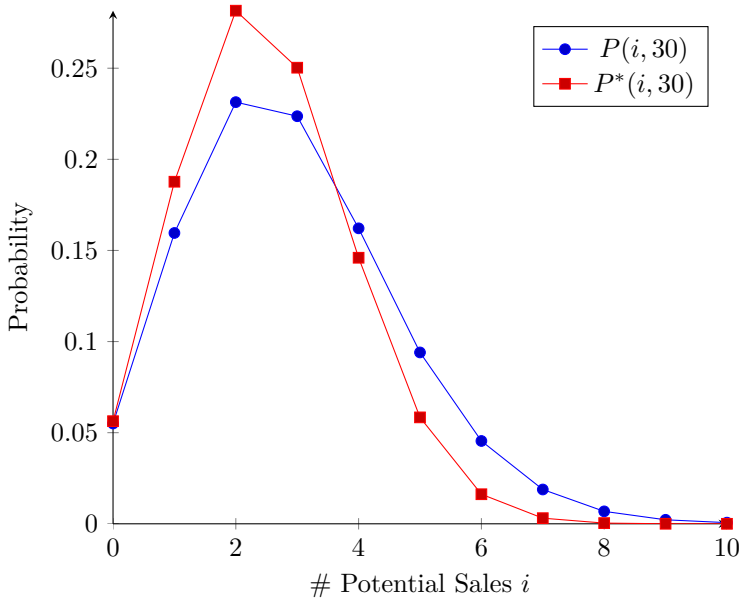

Figure 5.3. Comparison of predicted sales probabilities $P(i, a)$ with the true underlying binomial probabilities P^*, cf. (5.21), for a price of 30 based on ten training examples. The predictions are not as close as with the Poisson distribution but still a good approximation.

The generated training examples are used to train the demand learning model, cf. (5.13) and (5.14). Figure 5.2 shows sales probability predicted with (5.15) compared to true probabilities. The predictions are close to the true probabilities. This was possible despite only using only ten training examples scattered over a large price range.

This is an ideal example because the underlying probability distribution as well as the distribution used in the demand learning model are both Poisson distributions. In a second example, we want to evaluate the quality of predictions in a less ideal situation. We use a binomial distribution as the true underlying probability distribution, $a \geq 0, 0 \leq i \leq 10$

$$\gamma(a) = \max(-0.025a + 1, 0) \tag{5.20}$$

$$P^*(i, a) = \binom{10}{i} \gamma(a)^i (1 - \gamma(a))^{10-i} \tag{5.21}$$

Ten customers arrive every period and each customer has the price-dependent probability $\gamma(a)$ to buy a product. We chose parameters for $\gamma(a)$ to have a buy probabilities between 0.5 and 0.0 in the price range of 20 to 40.

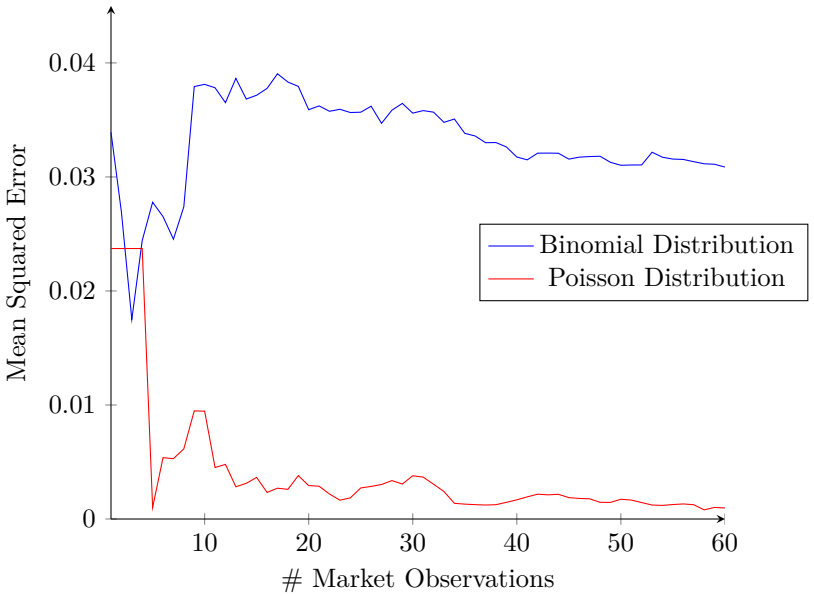

Figure 5.4. Estimation quality of sales probability with a true Poisson distribution and a true binomial distribution for different amounts of training examples. The quality is measured as mean squared error of the difference from predicted $P(i, a)$ to true probabilities $P^*(i, a)$, $i = 0, \ldots, 10$, $a = 20, 21, \ldots, 40$. Lower is better. The estimation error decreases with increasing amount of market observations.

Our demand learning model with a Poisson distribution is not able to perfectly fit the true underlying binomial distribution. However, the model is able to approximate that demand distribution, see Figure 5.3. Observations of customer arrival processes show that the arrivals are close to a Poisson distribution [45]. That is the reason why we use a Poisson distribution in our demand learning model.

Figure 5.4 shows the quality of predictions depending on how many training examples are available. The quality is measured as mean squares error of the difference between predicted $P(i, a)$ to true probabilities $P^*(i, a)$ over the parameters $i = 0, 1, \ldots, 10$ and $a = 20, 21, \ldots, 40$. Predictions of sales probabilities with an underlying Poisson distribution become more accurate with increasing number of training examples. Our demand learning model is bias towards Poisson distributed sales probabilities. This leads to a higher error for estimations of underlying binomial distributed demand probabilities.

5.4 Pricing & Ordering Optimization under Competition

Until now, the merchant offered items on the marketplace in a monopoly. This is typically not the case on real online marketplaces. Other merchants offer the same or similar products and everyone competes for market share. A common strategy to do that is to undercut competitors and offer the cheapest price. When multiple merchants pursue this strategy, the price will spiral downwards. Such a commercial competition is called price war. It is not the best strategy to always undercut the competitors because of decreasing profit margins. Sometimes it is better to also raise prices. This reduces sales, but the profit margin is higher. This section proposes a merchant that makes optimized ordering and pricing decisions under competition with the goal to maximize profit.

In the previous sections, the sales probability was only influenced by the merchant's own pricing decisions. In this scenario, the merchant's sales probability is also influenced by competitor's offers. The model is the same as in Section 5.2.1 with the exception that the number of sold items X_t within a period depends on the merchant's offer price and the competitor's offer prices. Merchants have access to historical market and sales data. The market data contains competitors' prices. However, sales data is limited to the merchant's own sales and does not contain sales from competitors.

Market situations can change frequently on online marketplaces. A merchant must be able to quickly and effectively react on new market situations to offer a suitable price for the new situation. Getting fast decisions is a challenge because the increasing complexity of the inventory and dynamic pricing problem over the last sections resulting in solutions with increasing computational effort. Nevertheless, there are ways to improve the speed of the dynamic programming approach which are described in Section 5.4.3.

5.4.1 Reacting to Competitor Offers

The inventory level was the only state on which ordering and pricing decisions were based on in the previous problems. With competitors on the marketplace, the merchant must also consider their offers when making ordering and pricing decisions. One possible approach is to add competitor offers to the state in the dynamic programming calculation. We decided against this solution because it drastically increases the state by multiple dimensions and therefore slows the computation down by multiple orders of magnitude due to the curse if dimensionality. Moreover, demand learning must predict transitions of competitor

offers from one time period to another. This means that competitor strategies must be learned, which requires a substantial amount of training data and a complex demand model to make sensible predictions.

Instead, we take the following approach. Ordering and pricing policies are created whenever the market situation changes. The policies are created specifically for the new market situation. The ordering and pricing decisions for this approach depend only on the merchant's inventory level. This keeps the computation fast and efficient. Additionally, only sales probabilities are predicted but not the competitors' price reactions. This can be done by a relatively simple demand model, which can make good predictions even with few market and sales data. The disadvantage is that new policies must be computed for each new market situation.

5.4.2 Predicting Sales Probabilities under Competition

Market situations become more complex with the introduction of competition. Previously, there was at most one active offer at a time. In this scenario, a market situation can contain as many offers as there are participating merchants. And the number of offers can change over time when merchants are out of stock. In Section 5.3, the merchant's offer price was sufficient to describe a market situation. With competition, the sales probability of an offer depends not only on its price but also on the price of the other offers. New explanatory variables are necessary for demand learning to describe the more complex market situations. We created two new explanatory variables besides the offer price in order to describe market situations with competition. A pricing decision is denoted by a and competitor prices of market situation $\vec{s}$ are denoted by $\vec{p}(\vec{s})$, $\vec{p}(\vec{s}) = (p_1, \ldots, p_K)$ where K is the number of competitor offers in that market situation, $K \in \mathbb{N}^0$. Our explanatory variables $\vec{x}(a, \vec{s}) = (x_0, x_1, x_2, x_3)$, with intercept $x_0 = 1$, are:

Own price

$x_1(a, \vec{s}) := a$

This explanatory variable holds the value of the merchant's current offer price. It helps to explain effects on the sales probability that are based on the total price. For example, a higher price generally results in less demand.

Price rank

$x_2(a, \vec{s}) := card(\{k = 1, \ldots, K \mid \vec{p}(\vec{s})_k \leq a\})$

This variable describes the merchant's relative position in the price ranking.

Even if a price change is small, it can have a huge impact on the sales probabilities if an offer is the cheapest or second cheapest offer.

Price difference to cheapest offer

$x_3(a, \vec{s}) := a - \min(\bigcup_{k=1}^{K} \{\vec{p}(\vec{s})_k\} \cup \{a\})$

This is the difference in price from the merchant's offer to the overall cheapest offer. If the merchant has the cheapest offer, the variable value will be €0. It is a relative variation of the 'own price' metric. This variable could explain effects like: a customer is willing to pay €5 more than the cheapest product but not €10 more.

An alternative explanatory variable is the price difference to the cheapest competitor offer. This metric uses the minimum of all prices without the merchants own price a. This metric contains the information about an price advantage over the cheapest competitor offer. However, the edge case must be considered if there are no competitor offers.

With these explanatory variables it is possible to describe the effect of the competition and the merchant's offer price on the sales probabilities. More explanatory variables can be easily defined. This way, it is possible to estimate demand that depends by new offer attributes like quality or rating.

The predicted sales probability are used in the dynamic programming approach to compute ordering and pricing policy. As explained above, the policies are computed for each new market situation. At this point, the computation takes around 10 seconds to complete. This is too long for the merchant to quickly and frequently react on the changing market. In the next section, we will reduce this time to allow shorter decision intervals for the merchant.

5.4.3 Efficient Dynamic Programming

Goal of this section is to reduce the time it takes to compute policies using dynamic programming. This allows the merchant to react faster on new market situations. We present three approaches to reduce computation time: using a better start value, adapt decision sets, tweaking the number of iterations, and early stopping. Some approaches trade off some of the value function's accuracy against time efficiency.

Figure 5.1 shows that dynamic programming converges faster if the start value u is near the resulting expected profit. We assume that the expected profit from two successive market situations is similar because the market situations will not fundamentally change in that short time frame. Setting $V_T(n) = V_0^{old}(n), n =$

$0, 1, \ldots, N_{max}$, where $V_0^{old}(n)$ is the expected discounted profit with a starting inventory of n items from the previous computation, will reduce the time until dynamic programming converges. Even if the assumption does not hold true and there is a big change in expected profit, the computation converges most of the time faster than with our previous start value $u = 0$.

Another way to increase the efficiency of the dynamic programming approach is to change the set of pricing decisions A and set of ordering decisions B to only contain relevant decisions. E.g., a order quantity of 100 is an irrelevant decision if the final policy only uses order quantities around 10. The irrelevant decision can be removed from the decision set without affecting the result. The dynamic programming approach is run for a reduced number of periods T for approximations of pricing and ordering policies. If A^*_{old}, $A^*_{old} \subseteq A$, is the set of pricing decisions of the previously computed policy, the adapted pricing decision space is set to

$$A := [\max(\min(A^*_{old}) - d, 0), \max(A^*_{old} + d)] \tag{5.22}$$

If B^*_{old}, $B^*_{old} \subseteq B$, is the set of ordering decisions of the previously computed policy, the adapted pricing decision space is set to

$$B := \{0\} \cup [\max(\min(B^*_{old} \setminus \{0\}) - d, 0), \max(B^*_{old} + d)] \tag{5.23}$$

The adaption parameter d, $d \in \mathbb{N}^0$, controls how many new decisions are added to the adapted decision sets. The set of potential ordering decisions is a special case because it must always contain 0. We narrow pricing decisions to integer prices. However, this concept can easily be extend to allow steps smaller or greater than 1 between prices.

The new decision sets are used to compute pricing and ordering policies again. The second dynamic programming computation is faster because it works on a reduced pricing and ordering decision sets. This process can be repeated in order to adapt the decision sets multiple times. This approach is not only faster than computing policies on the full decision sets, it also removes the limitation of having a static lower and upper limit on the price and order quantity. The decision sets can be adapted until they contain relevant decision. No prior knowledge of relevant prices and order quantities is necessary.

The end time T is a parameter that affects the number of iterations and therefore the computation time of the dynamic programming approach. The choice of this parameter is a trade-off between precision for large T and efficiency for

small T. One observation is that the policy converges faster than the calculated expected profit. This can be used to reduce T without affecting the precision of the resulting policies. It is possible to further reduce the end time T, if approximations of the optimal pricing and ordering policies are sufficient.

Early stopping ends the computation before reaching T iterations. This is done if the calculated policies do not change over a number of iterations. It is likely that these policies already converged to the optimal policies.

5.4.4 Evaluation of Efficient Dynamic Programming

The goal of our efficient dynamic programming approaches is to reduce the computation time in order to be able to react faster to new market situations. We measured the runtime performance of the dynamic programming variations to show the impact of the optimizations. All variations calculate pricing and ordering policies for the same market situation for a comparison. The market situation contains one own offer and two competitor offers.

The baseline for the measurements is the pure dynamic programming approach without any of the methods from Section 5.4.3. We set the end time $T = 500$, i.e., we iterate 500 times.

The first optimization method sets the start value to the expected profit from the last computation, $V_T(n) = V_0^{old}(n)$ for all n, $n = 0, 1, \ldots, N_{max}$. This way the start values are closer to the final expected profit compared to a start value of 0. The value function converges faster with an appropriate start value. We utilize this fact and reduce the number of iterations to 200 in order to decrease the computation time with only a minimal precision loss.

The next optimization method is the adaption of the ordering and pricing decision sets. There are 5 successive dynamic programming calculations with 40 iterations each. This are in total again 200 iterations, but the decision sets are adapted between calculations. This reduces the size of the decision sets without omitting relevant decisions. Relevant decisions are decisions that occur in the optimal policy. Smaller decisions sets lead to a more efficient computation of pricing and ordering policies. Adapted ordering and pricing decision sets are reused from the previous computation similarly to the start value.

The last optimization method is early stopping. This stops the dynamic programming calculation if the ordering and pricing policies do not change after a number of iterations. The assumption is that the polices are already converged, or are almost converged and are close enough to the optimal policy. Early stopping

Method	Runtime [ms]	Error [%]
Baseline	9 425	0.53
Set start value to result of previous computation	3 734	0.98
Adapt ordering & pricing decisions	1 550	5.71
Early stopping	747	5.71

Table 5.5. Computation time improvements of the dynamic programming approach with the proposed optimization methods. Each row contains the optimizations from the rows above. The last column (Error [%]) is the percent error between calculated profit $V_0(0)$ and actual expected profit $V_0^*(0)$. Runtimes were measured on a i7-3517U CPU.

saves computation time by cutting out iterations that do not have a great impact on the resulting policies. The results are listed in Table 5.5. Note that each row in the table also contains the optimizations from the rows above.

Applying all three optimization methods to the dynamic programming approach reduces the computation time in our example from 9.4 seconds to 0.75 seconds, a reduction by 92%. This is fast enough to quickly update prices after a new market situation. These optimizations come at the cost of precision of the dynamic programming approach. The precision loss for setting the start value to the expected profit from the previous result and for early stopping are neglectable. In contrast, the adaption of the decision sets introduce an percent error of around 5% to the resulting expected profit. Despite the precision error, the computed policies usually are identically to the optimal policy. We are willing to accept the the loss in precision in order to increase the efficiency of the dynamic programming approach.

5.4.5 Merchant Description and Simulation Setup

Our merchant implementation with the proposed optimization model is called data-driven merchant. A new training on all training data every minute. The period length is 4 seconds. $N_{max} = 40$, $T = 40$, $d = 5$, and $\delta = 1$. The decision spaces are adapted up to 5 times after T iterations each.

Our merchant, called data-driven merchant, competes with two rule-based merchants. The cheapest merchant always undercuts the cheapest competitor by 0.30. Only if the cheapest competitor price is higher than the upper price bound of 30, the cheapest merchant sets a price of 30 instead. If no competitor offer is available, the cheapest merchant sets the price to the upper price bound. The merchant makes a new order, if the inventory level falls below 6 items. In that

case, the merchant orders as many items as needed to refill the inventory to 20 items. The period length is 4 seconds.

The second rule-based merchant, called two bound merchant, undercuts the cheapest competitor offer by 0.30, similar to the cheapest merchant. However, the merchant has a upper and lower price bound. If the cheapest competitor offer price is below the lower price bound of 17, the two bound merchant sets the price to the upper price bound, 30. Moreover, if no competitor orders are available or all competitor prices are above the upper price bound, the price is also set to the upper price bound. This merchant makes a new order if the inventory level falls below 4 items. In that case, the merchant orders as many items as needed to refill the inventory to 15 items. The period length is 4 seconds.

Consumers are configured to visit the marketplace at an average rate of 100 consumers per minute. The time between arriving consumers is exponential distributed with a mean of 0.6 seconds. They dismiss offers costing 80 or more. The remaining J offers $\vec{o} = (o_1, o_2, \ldots, o_J)$ have prices $\vec{p} = (p_1, p_2, \ldots, p_J)$. The maximal price in $\vec{p}$ is denoted by p_{max}, the sum of these prices is denoted by p_{sum}. A consumer buys one item from the remaining offers at random with the probability distribution by (5.24), j = 1, ..., J.

$$P(\text{Buy from } o_j) = \frac{p_{max} + 1 - p_j}{J \cdot (p_{max} + 1) - p_{sum}} \tag{5.24}$$

If there are no offers with prices below 80, the consumer leaves the marketplace without buying anything.

Simulations have a duration of 15 minutes. $c_{fix} = 10$, $c_{var} = 15$. Holding costs are 3 per minutes for all merchants. That is $l = 0.2$ for the data-driven merchant. If not mentioned otherwise, the following simulations are run with the configuration listed here.

5.4.6 Duopoly Simulation

We investigate the profitability of the proposed merchant in a duopoly scenario in this section and in an oligopoly scenario in the following section. Our data-driven merchant competes with each of the rule-based merchants. In the last duopoly scenario, two data-driven merchants with the same strategy compete with each other.

Figure 5.5. Price trajectories in a duopoly of data-driven merchant and cheapest merchant. Dots in the chart are price updates and bars are sales events. Parameters as in Section 5.4.5.

Data-Driven Merchant vs. Cheapest Merchant

We simulated competition in a duopoly between the data-driven and cheapest merchant for 15 minutes. Figure 5.5 shows how both merchants undercut each other. With variable order costs of 15, the data-driven merchant does not reduce the price below 20 and instead raises the price to 25/26. The increased price has lower sales probabilities but a higher profit margin. The cheapest merchant increases the price sometimes. When the data-driven merchant is out of stock, there is no competitor offer for the cheapest merchant to undercut. When this is the case, the cheapest merchant uses a default price. The final results of this simulation are shown in Table 5.6. The data-driven merchant has overall more costs but makes more profit because of a higher revenue than the cheapest merchant.

Data-Driven Merchant vs. Two Bound Merchant

Running a duopoly with the data-driven and two bound merchant does not result in both merchants undercutting each other as we have seen in the previous

Merchant	Profit	Revenue	Holding Cost	Order Cost
Data-Driven	7 285.78	20 599.00	588.22	12 725.00
Cheapest	5 796.11	17 165.10	418.99	10 950.00

Table 5.6. Simulation results of a duopoly with the data-driven and cheapest merchant. Parameters as in Section 5.4.5.

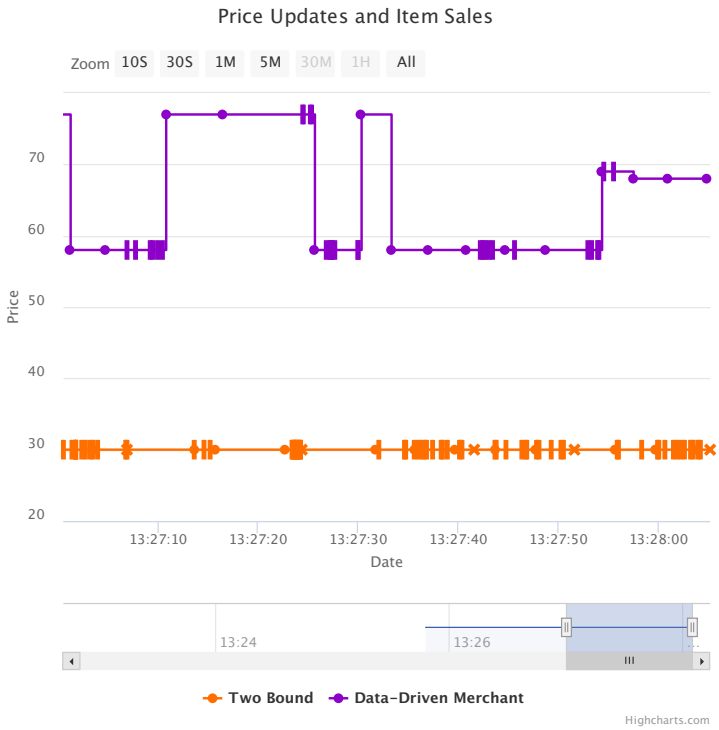

Figure 5.6. Price trajectories in a duopoly of data-driven merchant and two bound merchant. Parameters as in Section 5.4.5.

example between the data-driven and cheapest merchant. Figure 5.6 shows that the data-driven merchant offers to a much higher price compared to the competitor. The two bound merchant sets the price to the maximum that the merchant's configuration allows.

Customers are unlikely to buy from the data-driven merchant if other offers are only at half the price. However, the data-driven merchant learned that there is a good chance to sell products at a high price because the two bound merchant is prone to stock-outs.

The inventory graph in Figure 5.7 shows clearly that the data-driven merchant only sells products if the two bound merchant is out of stock. Customers start

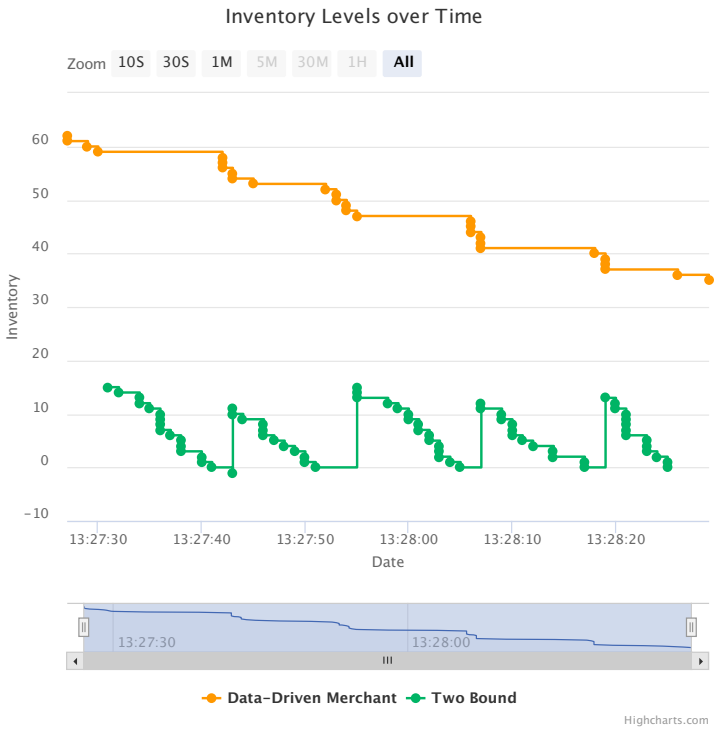

Figure 5.7. Inventory levels over time in a duopoly of data-driven merchant and two bound merchant. The two bound merchant has many stock-out because of reordering only if the inventory level falls below 4 items. Parameters as in Section 5.4.5.

Merchant	Profit	Revenue	Holding Cost	Order Cost
Data-Driven	15 571.60	24 761.00	1 484.39	7 705.0
Two Bound	13 379.67	29 138.90	229.22	15 530.0

Table 5.7. Simulation results of a duopoly with the data-driven and two bound merchant. Parameters as in Section 5.4.5. Parameters as in Section 5.4.5.

buying the expensive products if no cheaper alternative is available. The data-driven merchant is able to make more profit than the two bound merchant without offering a cheaper price. Overall, both merchants made more profit compared to the duopoly with the data-driven and cheapest merchant because they offered products at higher prices. The results are shown in Table 5.7.

The two bound merchant could have made more profit with a higher upper price bound. However, the rule-based merchant cannot adapt rules and therefore cannot appropriately react to this unexpected market situation.

In a second simulation of the competition between the data-driven merchant and the two bound merchant, we increased the number of items the two bound

Figure 5.8. Price trajectories in a duopoly of data-driven merchant and two bound merchant. The two bound merchant restocks the inventory to 25 instead of 15 items and reorders if the inventory falls below 7 instead of 4 items to reduce the number of stock-outs. Other parameters as in Section 5.4.5.

merchant holds. The two bound merchant restocks to 25 items instead of 15 and makes a new order whenever the inventory level falls below 7 items instead of 4. This reduces the risks of having stock-outs.

The data-driven strategy from the previous simulation to offer items at a high price alongside the two bound merchant's offers is no longer viable. The probability to sell items at the previous high prices is small because stock-outs for the two bound merchant are less likely to happen.

The data-driven merchant expects the most profit from undercutting the competitor. This results in both merchants undercutting each other as shown in Figure 5.8. The two bound merchant is programmed to raise the price if it falls below a threshold. However, the data-driven merchant was the first to increase the price in this simulation. The results are shown in Table 5.8. Both merchants made less profit compared to the previous simulation because they frequently decrease prices to have the lowest prices.

Merchant	Profit	Revenue	Holding Cost	Order Cost
Data-Driven	5 858.79	18 984.00	595.20	12 530.0
Two Bound	5 230.10	16 952.70	527.60	11 195.0

Table 5.8. Simulation results of a duopoly with the data-driven and two bound merchant. The two bound merchant holds more items in the inventory to reduce the number of stock-outs. The two bound merchant restocks the inventory to 25 instead of 15 items and reorders if the inventory falls below 7 instead of 4 items to reduce the number of stock-outs. Other parameters as in Section 5.4.5.

Figure 5.9. Price trajectories in a duopoly of two identically configured data-driven merchants. Parameters as in Section 5.4.5.

Data-Driven Merchant vs. Data-Driven Merchant

The next simulation is about the competition between two identical copies of our data-driven merchant. The price chart in Figure 5.9 shows again the typical zig-zag pattern of two merchants undercutting each other and periodically pushing the price up to increase profit margins. The competition between two data-driven merchants happens at a higher price level (around 32-51) compared to the previous simulation (around 20-26). This results in an overall higher profit for both competing merchants. The data-driven merchants order new items

Merchant	Profit	Revenue	Holding Cost	Order Cost
Data-Driven	17 361.30	30 936.0	804.69	12 770.0
Data-Driven 2	15 650.58	27 282.0	721.41	10 910.0

Table 5.9. Simulation results of a duopoly with two data-driven merchants with the same configuration. Parameters as in Section 5.4.5

Merchant	Profit	Revenue	Holding Cost	Order Cost
Data-Driven	5 944.13	21 938.00	943.87	15 050.00
Cheapest	5 386.90	23 770.80	903.89	17 480.00
Two Bound	5 038.63	20 148.30	644.67	14 465.00

Table 5.10. Simulation results of the oligopoly scenario with the data-driven, cheapest, and two bound merchant. The data-driven merchant made the most profit. The cheapest merchant made the most revenue. Parameters as in Section 5.4.5 with a simulation duration of 30 minutes.

whenever the inventory level falls below 6 items and restock the inventory to around 28 items.

Results of this simulation are shown in Table 5.9. Merchants do not collect the same sales experiences which may result in different pricing and ordering policies. Different policies and (bad) luck on customer decisions are the reasons for differences in revenue, profit, and costs between both merchants.

5.4.7 Oligopoly Simulation

This section shows how our merchants performs in an oligopoly. The merchants sell products on the Price Wars platform. The objective is to maximize the expected discounted profit.

We simulated the scenario with competition on the Price Wars platform. The web UI shows pricing and ordering decisions during the simulation (see Figures 5.10 and 5.11). The data-driven merchant learned the advantage of undercutting competitors' offers. However, this creates a high price competition and offer prices decrease. With shrinking profit margins, it becomes unprofitable to set the price below competitors' prices. The data-driven merchant pushes the price up in such a situation. This action is especially successful if the competitors also increase their offer prices.

Figure 5.10. Price trajectories in an oligopoly scenario on the Price Wars platform. Our data-driven merchant competes with two rule-based merchants. The merchants have a race for the cheapest price. The data-driven merchant sometimes raises the price to increase the price niveau. Parameters as in Section 5.4.5 with a simulation duration of 30 minutes.

Table 5.10 shows the results of a half an hour competition between three merchants. The data-driven merchant outperformed all competitors. Our data-driven merchant made around 10% more profit than the cheapest merchant and 18% more than the two bound merchant. Interestingly, our merchant did not make the most revenue, but the cheapest merchant did. The cheapest merchant sold the most items but with low profit per item. Our merchant made the most profit by saving a lot of order cost compared the to cheapest merchant. The data-driven merchant orders on average more items than the competitors. This results in higher holding costs but saves on fixed order cost.

5.5 Merchant Implementation

Merchants on the Price Wars platform can be written in any language as long as they comply with the platform's REST APIs. We decided to implement our merchant in the Python programming language [43] for the following reasons.

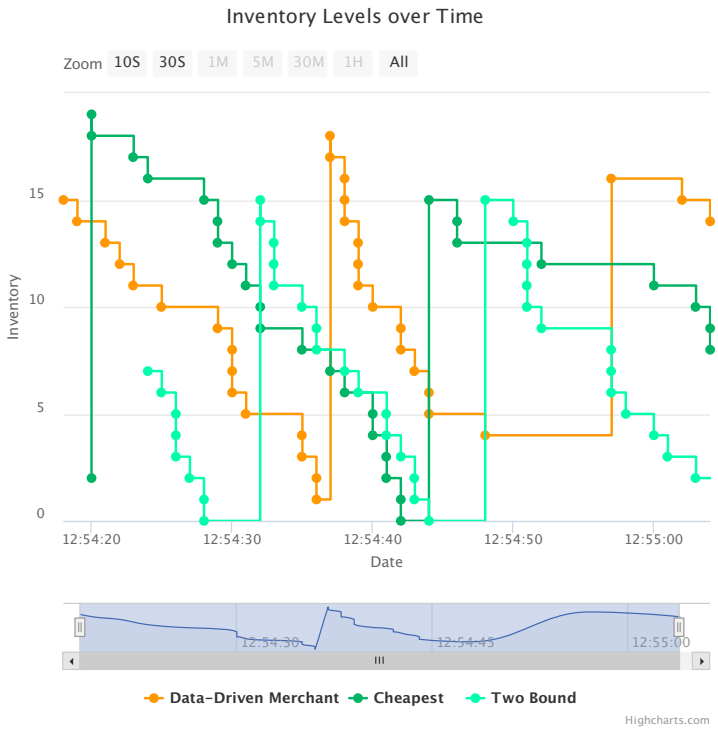

Figure 5.11. Inventory levels over time in an oligopoly scenario on the Price Wars platform. Our data-driven merchant competes with two rule-based merchants. Parameters as in Section 5.4.5 with a simulation duration of 30 minutes.

Python has great library support for numerical computing. These libraries allow a concise and efficient implementation without reinventing the wheel. The Price Wars platform offers a Python implementation of the RESTful API for the merchant to communicate with the platform's services. Lastly, it is possible to quickly create prototypes in Python.

Our merchant consists of four components as it is shown in Figure 5.12. The merchant regularly checks the marketplace for open offers, updates prices, and orders items from the producer. After enough time passed, the merchant requests new market and sales data from the Kafka Reverse Proxy and provides it to the demand learning component to analyze demand.

The merchant server receives sales events from the marketplace and configuration updates from the web UI. On such requests, the merchant server calls the appropriate action in the main merchant component.

The merchant makes ordering and pricing decisions based on policies that are computed by the policy optimizer component. The policy component contains our dynamic programming approach. The merchant provides all arguments that

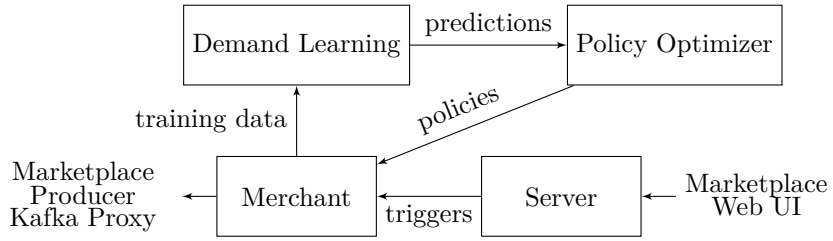

Figure 5.12. Architecture of our merchant implementation. The merchant retrieves training data from platform for the demand learning component. The policy optimizer component uses sales probability predictions from the demand learning component to create ordering and pricing policies. The merchant server triggers a merchant action for requests from the marketplace or web UI.

are necessary for the policy creation and sales probabilities are requested from the demand learning component.

The dynamic programming function is the computational most expensive part of the merchant. An efficient implementation reduces the time needed for a pricing and ordering decision. We create a vector that has the dimensions inventory levels, ordering decisions, pricing decisions, and demand. The expected profit is calculated for each possible situation and decision that occur in this vector. The expected profits are used to find the most profitable decisions and to create the ordering and pricing policy. We use fast and vectorized array operations from the Numpy library [19] to compute the policies. Python is a high-level programming language and has a lot of computational overhead [7]. Numpy provides data structures and functions implemented in the C programming language to overcome Python's overhead for numeric computations.

The merchant's demand learning component is responsible for estimating sales probabilities and for bringing market and sales data into a form that can be used for training. The module uses linear regression to learn and predict the demand. We use the scikit-learn library [32] for a reliable and fast linear regression implementation. As an additional benefit, it is easy to change between regression algorithms using scikit-learn. The demand learning is implemented in a way that make it easy to add new or change existing explanatory variables. Only only single function (named `extract_features`) must be changed to add new explanatory variables.

The merchant's code is open source and can be found on Github: `https://github.com/CarstenWalther/pricewars-merchant`.

6

Extending the Price Wars Platform

The Price Wars platform[1] is a framework to test and evaluate merchant strategies in online marketplace environments under competition. Merchants' main focus is on the dynamic pricing problem, i.e., on making good pricing decisions depending on the current market situation. Formerly, merchants could only order one item at a time from the producer. The simplest and best ordering strategy was to order a new item whenever an item was sold. This was okay, since ordering was not in focus.

Ordering decisions become import for merchants in this thesis and we extended the Price Wars platform accordingly to create a more realistic ordering and inventory control process. Merchants in the real world order multiple items at once to reduce fixed order costs. They have to decide at what point in time to order and how many items to order. This leads to the inventory control problem. Merchants have to keep customer demand in mind to avoid overstocking and understocking.

We extended the Price Wars platform to allow simulations of merchants that compete against each other by making ordering and pricing decisions. We added new costs to the platform that might influence merchants' ordering decisions. Moreover, merchants must deal with delivery times for their orders.

The extensions are explained in detail in the following sections. The two Sections 6.5 and 6.6 present helpful additions to evaluate merchant strategies. The implementation of our merchant is not discusses in this chapter. Instead, it can be found in Section 5.5.

[1] Price Wars on Github: `https://github.com/hpi-epic/pricewars`

6.1 Ordering Multiple Items

Merchants must be able to order multiple items per order to make proper order decisions. Some platform components already supported this use case. For example, expense and profit calculation for merchants already considered orders of multiple items. However, there was no option to order more than one item from the producer. The original order request to get one random product was `POST /orders`. This request gets a new parameter that specifies the desired amount, e.g., `POST /orders?amount=14`. Accordingly, the producer returns an order with that many items. Ordering different product types with one order is not supported.

The option to order multiple items will not affect merchants' strategies. They can still order a single item whenever they need one. The next section introduces fixed order cost to discourage this behavior.

6.2 Fixed Order Costs

Fixed order cost is a fixed value that is added to the total costs of each order. Fixed order costs are comparable to shipping costs. The total order costs from an order can be calculated with (5.1). Merchants can reduce their fixed order costs by making few large orders. They can learn about the current fixed order costs per order from the product information from producer with the request `GET /products`. Each product type can have different fixed order costs.

Besides producer and merchants, the event aggregation service needs to know the total order cost to calculate merchants' profits and expenses. The event aggregation service used to calculate the total order cost from the amount of ordered items and the cost per item. The redundant order cost calculations may cause errors if the implementations are inconsistent. To prevent these errors, only the producer calculates the cost and adds this information to the order event. The event analysis service becomes simpler and changing the order cost formula requires only to update the producer.

Variable order costs were already present before this extension, even though it was only possible to order single items.

With orders of multiple items and fixed order cost in place, a good merchant strategy is to make one large order at the beginning. To make ordering decisions more interesting and natural, we introduce holding costs in the next section.

6.3 Holding Costs

Holding costs occur when merchants store items in their inventory. Each item that a merchant received from the producer and has not sold yet on the marketplace causes holding costs over time. Holding costs penalize merchants for overestimating demand and having excessive inventory levels. There is no inventory capacity limit but profit-oriented merchants have a practical inventory limit in order to avoid high holding costs.

A strategy to minimize holding costs is ordering often few items. This keeps the inventory level constantly low. Disadvantage of this strategy are high fixed order costs. In order to maximize profit, merchants must find a trade-off between low fixed order cost with few large orders and low holding costs with many small orders.

Real-world merchants are responsible for their inventory management and the resulting holding costs. However, in a Price Wars simulation it is not necessary to store physical items. Holding costs are only simulated. If merchants were responsible for their holding costs, they could cheat easily by reporting a lower amount. We decided to not trust merchants in that regard and calculate holding costs on the platform's services.

The marketplace manages the holding cost rate for each merchant. The holding cost rate indicates how much it costs to hold one item for a minute in the inventory. Merchants can request their current holding cost rate from the marketplace.

The actual holding cost calculation happens in Flink, the event aggregation service of the Price Wars platform. Flink receives events about inventory growth whenever a merchant gets items from the producer and about inventory reduction whenever a merchant sells items on the marketplace. Flink tracks the current inventory level of each merchant using information from these events. Additionally, Flink receives events about changes of holding cost rates from the marketplace. Having access to inventory levels over time and holding cost rates, it is possible to calculate holding costs in Flink.

The computation of holding costs is triggered every time a merchant's inventory level or holding cost rate changes. In that case, the holding costs since the last change are calculated. Figure 6.1 shows an example of this process. The holding costs of the interval between two consecutive events is the inventory level multiplied with the duration and the holding cost rate. Note that changes in inventory level or holding cost rate can happen at any time and are not periodically.

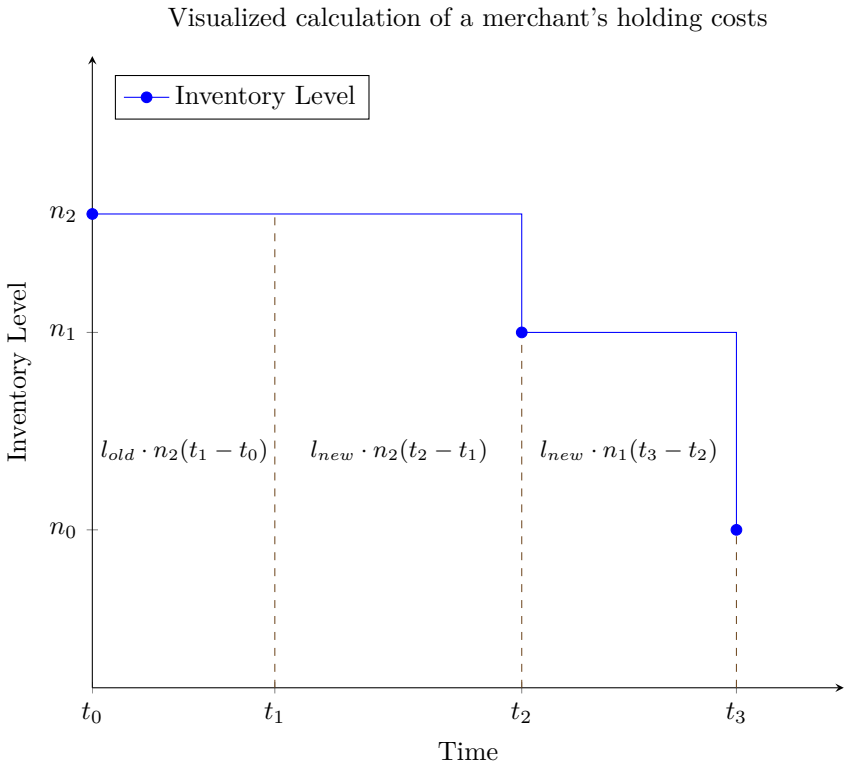

Figure 6.1. Each segment under the graph corresponds to one holding cost computation. At t_1 the holding cost rate changes from l_{old} to l_{new}. Whenever a merchant's inventory level or holding cost rate changes, the holding costs since the previous change are computed. Holding costs depend on the holding cost rate, the inventory level, and time since the last change.

6.4 Delivery Time

After a merchant orders items from the producer, some time should pass until the items arrive at the merchant. This is the delivery time from producer to merchant. Previously on the Price Wars platform, merchants received ordered products instantly. Merchants must think ahead of time while ordering to receive the products at the right time. If merchants underestimate demand, they risk stock-outs before the ordered products arrive. The addition of delivery time will result in a more realistic competition between merchants on the online marketplace.

Previously, merchants made a HTTP request to the producer for an order and the response contained the ordered items. To add a delivery time to this process, we considered three approaches: long-running HTTP requests, web sockets, and two separate HTTP requests.

Long-running HTTP requests are the simplest solution to implement delivery time. The producer creates the ordered items and waits a certain time to respond. However, this approach is problematic due to timeouts and may block execution of the producer or merchant. We decided against long-running requests.

Web sockets are a flexible alternative to HTTP requests and create a bi-directional connection. The merchant opens a web socket to the producer, then sends an order over the web socket. The producer can send the ordered items at any time as a response. When the merchant received the items, the web socket can be closed or kept open for additional orders. The consensus on the usage of web sockets is that they should not be used to replicate request-response communication. Thus web sockets are not the optimal solution to implement delivery time and we used the third approach, two separate requests.

The process of ordering and receiving products is split into two separate HTTP requests. The merchant creates a new order with an order request. The producer responds with an estimated time until the order is ready. If the delivery time is over, the merchant can make a request to receive the ordered items from the producer. The producer only returns the ordered items if the delivery time is over. An order can only be received once.

This solution requires an additional HTTP request compared to instant orders. The additional overhead is small and, compared to web sockets, it is easier to update existing merchants to the new ordering process. In contrast to the long-running request, this method is non-blocking. The known duration until the order is ready is advantageous for this approach. The merchant does not need to poll the producer until the order is ready and can instead wait the mentioned duration before sending the second request.

6.5 Inventory Visualization

The platform extensions create the inventory control problem for merchants. Merchants' inventory levels are more important than ever before because of this problem. In order to analyze how merchants control their inventory levels, we added an inventory chart to the web UI. Users have access to the new chart that shows inventory levels over time from all merchants. Figure 6.2 shows an example of the inventory chart. This chart is shown along other charts, which present metrics like prices, profit, and revenue. We used the highcharts[2] library to have the same look and feel as the existing charts.

[2] JavaScript chart library, Highcharts: `https://www.highcharts.com/`

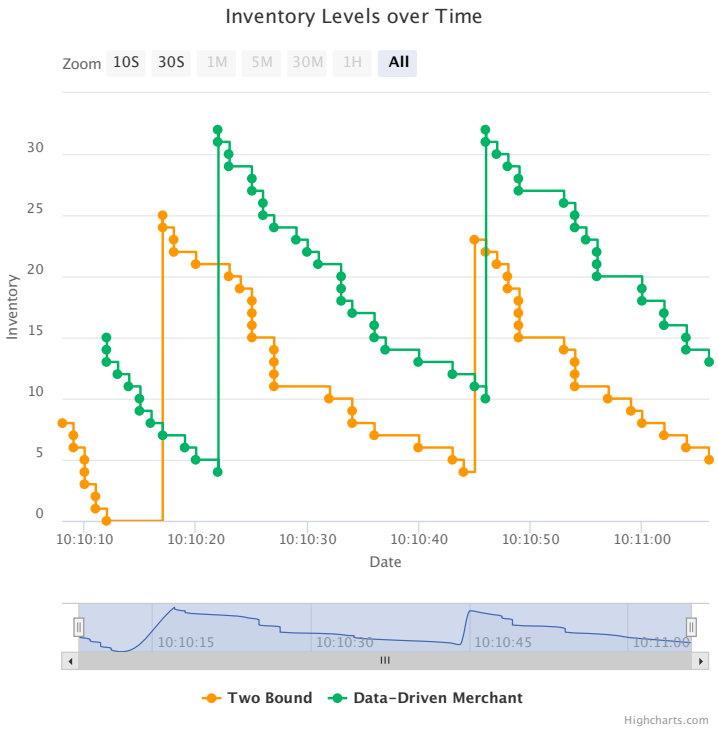

Figure 6.2. Example of the Inventory Chart in the web UI showing inventory levels over time of two competing merchants.

6.6 Benchmarking Tool

Getting comparable results over multiple simulations on the Price Wars platform was problematic while developing our merchant. The charts on the web UI are helpful in evaluating a single simulation run but it is not precise enough to compare multiple runs. It is not feasible to manually examine charts at an exact timing. A better way to compare simulation runs with each other is especially important in monopoly scenarios, when merchant with different configurations or strategies are compared.

We developed a benchmarking tool that runs simulations on the Price Wars platform with specified merchants for a certain duration. This tool allows the user to run multiple simulations for the same duration. With this tools, it is no longer necessary to read a chart at the right point in time.

After a run, the benchmark tool saves all events that happened on the platform from the event log. The events contain orders, sales, and inventory levels among others and are saved in JSON format. Events are analyzed to determine each

Merchant	Profit	Revenue	Holding Cost	Order Cost
Merchant A	534.17	3 547.00	212.83	2 800.00
Merchant B	315.12	4 063.80	363.80	3 385.00
Merchant C	492.50	3 844.80	152.30	3 200.00

Table 6.1. The benchmark tool generates a breakdown of expenses and revenues. Example results are shown from a five minute simulation.

merchant's profit, revenue, and expenses as shown in Table 6.1. These results are saves as a table in CSV format.

After a simulation, users can run additional analysis on the saved events to answer new questions, like: how many stock-outs had each merchant? Another benefit from using the benchmark tool is the automated setup and teardown of the platform. This process would need some manual steps otherwise like clearing the state from the previous simulation and starting the consumer.

The benchmark tool is a Python script. The command to run the tool is `python3 helper_scripts/benchmark.py`. The parameter `--duration` controls the length of the simulation in minutes. The user provides a list of merchant start commands to `--merchants`. The start commands are the same commands that would be used to start merchants manually. The parameter `--holding_cost` sets the same holding cost rate for all merchants. Results are written to the directory given to the `--output` parameter.

6.7 Miscellaneous

Besides already mentioned extensions, we made further improvements to the Price Wars platform. Here are listed noteworthy improvements.

We unified the implementations of example merchants to make their behavior consistent and reduce the amount of shared copied code. The new interface makes it easier to create a new merchant based on the example merchants.

We drastically reduced the time of the first-time setup of the Price Wars platform.

We added a new consumer behavior (cf. Section 5.4.5), which can be selected in the web UI. This behavior prefers the cheapest offer but also has a probability to buy more expensive products. This probability decreases with increasing price difference to the cheapest offer. Further, we fixed a bug that caused less consumers to arrive than configured.

7

Conclusion & Future Work

Making suitable decisions on competitive online marketplaces is crucial for a merchant's success. We presented different dynamic optimization models for four problem scenarios, which become increasingly realistic but also more complex. The final problem scenario considers ordering and pricing decisions on an online marketplace under competition. Pricing and ordering policies are created with a dynamic programming approach. The uncertain demand is estimated based on historical market data using demand learning with linear regression.

We implemented a merchant that uses the proposed optimization approaches. The merchant was tested and evaluated on the Price Wars platform. Computation time efficiency of the dynamic programming approach was improved until the merchant could react to new market situations within one second. This time can be further reduced with little impact on resulting policies. One advantage of our adaptive dynamic programming approach is that suitable ranges of potential pricing and ordering decisions are found over time and do not have to be configured by the user. Our merchant can cope with demand changes over time because of demand learning and the adaptable sets of pricing and ordering decisions.

We compared performance of different merchant strategies in duopoly and oligopoly simulations. As soon as a sufficient amount of sales observations is available, our merchant outperforms traditional rule-based merchants. In duopoly settings, our merchant typically undercuts the competitor if it is not profitable to have a much higher or lower price. Our merchant raises the price at some point, if both merchants undercut each other and the price decreases over time. The presented optimization approaches can be applied to real online marketplaces – such as Amazon – to potentially increase profits.

Formerly, it was not possible to simulate joint ordering and pricing problem scenarios on the Price Wars platform. We extended the platform by orders of

multiple items, fixed order costs, holding costs, and order delivery delay. These extensions allow testing and evaluating merchant strategies under more realistic conditions. Moreover, our merchant implementation can be used as a baseline to compare new ordering and pricing strategies.

There are three lines of work arising from this thesis. The dynamic pricing and inventory control problem can be extended to consider perishable products or multiple products with interdependent demand. Both problems require changes to the dynamic programming approach and the problem with multiple products with interdependent demand requires changes to the demand learning component. The price and order optimization could anticipate competitors' reactions. This can result in different policies compared to optimizations that do not consider competitors' reactions. Lastly, customer demand could depend on more dimensions than the price, like product quality or merchant rating. This requires new explanatory variables for market situations and maybe a more complex demand learning model.

References

[1] Adida, E., and Perakis, G. A robust optimization approach to dynamic pricing and inventory control with no backorders. *Mathematical Programming 107*, 1-2 (2006), 97–129.

[2] Adida, E., and Perakis, G. Dynamic pricing and inventory control: robust vs. stochastic uncertainty models - a computational study. *Annals of Operations Research 181*, 1 (2010), 125–157.

[3] Adida, E., and Perakis, G. Dynamic pricing and inventory control: Uncertainty and competition. *Operations Research 58*, 2 (2010), 289–302.

[4] Araman, V. F., and Caldentey, R. Revenue management with incomplete demand information. *Wiley Encyclopedia of Operations Research and Management Science* (2011).

[5] Azoury, K. S. Bayes solution to dynamic inventory models under unknown demand distribution. *Management Science 31*, 9 (1985), 1150–1160.

[6] Ban, G.-Y., and Rudin, C. The big data newsvendor: Practical insights from machine learning. *Forthcoming in Operations Research* (2018).

[7] Behnel, S., Bradshaw, R., Citro, C., Dalcín, L., Seljebotn, D. S., and Smith, K. Cython: The best of both worlds. *Computing in Science and Engineering 13*, 2 (2011), 31–39.

[8] Besbes, O., and Muharremoglu, A. On implications of demand censoring in the newsvendor problem. *Management Science 59*, 6 (2013), 1407–1424.

[9] Bisi, A., and Dada, M. Dynamic learning, pricing, and ordering by a censored newsvendor. *Naval Research Logistics 54*, 4 (2007), 448–461.

[10] Bitran, G. R., and Wadhwa, H. K. S. *A methodology for demand learning with an application to the optimal pricing of seasonal products.* Sloan School of Management, Massachusetts Institute of Technology, 1996.

[11] Boissier, M., Schlosser, R., Podlesny, N., Serth, S., Bornstein, M., Latt, J., Lindemann, J., Selke, J., and Uflacker, M. Data-

driven repricing strategies in competitive markets: An interactive simulation platform. In *Proceedings of the Conference on Recommender Systems* (2017), pp. 355–357.

[12] Broder, J., and Rusmevichientong, P. Dynamic pricing under a general parametric choice model. *Operations Research 60*, 4 (2012), 965–980.

[13] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[14] Carlson, J. L. *Redis in action*. Manning Publications Co., 2013.

[15] Chen, M., and Chen, Z.-L. Recent developments in dynamic pricing research: multiple products, competition, and limited demand information. *Production and Operations Management 24*, 5 (2015), 704–731.

[16] Chen, X. *Coordinating inventory control and pricing strategies*. PhD thesis, Massachusetts Institute of Technology, 2003.

[17] den Boer, A. V. Dynamic pricing and learning: historical origins, current research, and new directions. *Surveys in Operations Research and Management Science 20*, 1 (2015), 1–18.

[18] DiMicco, J. M., Maes, P., and Greenwald, A. Learning curve: A simulation-based approach to dynamic pricing. *Electronic Commerce Research 3*, 3-4 (2003), 245–276.

[19] Dubois, P. F., Hinsen, K., and Hugunin, J. Numerical Python. *Computers in Physics 10*, 3 (1996), 262–267.

[20] Elmaghraby, W., and Keskinocak, P. Dynamic pricing in the presence of inventory considerations: Research overview, current practices, and future directions. *Management Science 49*, 10 (2003), 1287–1309.

[21] Federgruen, A., and Heching, A. Combined pricing and inventory control under uncertainty. *Operations Research 47*, 3 (1999), 454–475.

[22] Gallego, G., and Van Ryzin, G. Optimal dynamic pricing of inventories with stochastic demand over finite horizons. *Management Science 40*, 8 (1994), 999–1020.

[23] Garg, N. *Apache Kafka*. Packt Publishing Ltd, 2013.

[24] Harris, F. W. How many parts to make at once. *Factory, the magazine of management 10*, 2 (1913), 135–136.

[25] Huh, W. T., Levi, R., Rusmevichientong, P., and Orlin, J. B. Adaptive data-driven inventory control with censored demand based on kaplan-meier estimator. *Operations Research 59*, 4 (2011), 929–941.

[26] Knöpfel, A., Gröne, B., and Tabeling, P. Fundamental modeling concepts. *Effective Communication of IT Systems, England* (2005).

[27] Le Guen, T. Data-driven pricing. Master's thesis, Massachusetts Institute of Technology, 2008.

[28] Levi, R., Roundy, R., and Shmoys, D. B. Provably near-optimal sampling-based policies for stochastic inventory control models. *Mathematics of Operations Research 32*, 4 (2007), 821–839.

[29] Martínez-de-Albéniz, V., and Talluri, K. T. Dynamic price competition with fixed capacities. *Management Science 57*, 6 (2011), 1078–1093.

[30] Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (Mar. 2014).

[31] Morris, J. A simulation-based approach to dynamic pricing. Master's thesis, Massachusetts Institute of Technology, 2001.

[32] Pedregosa, F., Varoquaux, G., Gramfort, A., et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[33] Phillips, R. L. *Pricing and revenue optimization*. Stanford University Press, 2005.

[34] Pinto, T., Vale, Z. A., Sousa, T. M., Praça, I., Santos, G., and Morais, H. Adaptive learning in agents behaviour: A framework for electricity markets simulation. *Integrated Computer-Aided Engineering 21*, 4 (2014), 399–415.

[35] Scarf, H. E. A survey of analytic techniques in inventory theory. *Multistage Inventory Models and Techniques 7* (1963), 185–225.

[36] Schlosser, R., and Boissier, M. Optimal price reaction strategies in the presence of active and passive competitors. In *International Conference on Operations Research and Enterprise Systems* (2017), pp. 47–56.

[37] Schlosser, R., and Richly, K. Dynamic pricing strategies in a finite horizon duopoly with partial information. In *International Conference on Operations Research and Enterprise Systems* (2018), pp. 21–30.

[38] Serth, S., Podlesny, N., Bornstein, M., Latt, J., Lindemann, J., Selke, J., Schlosser, R., Boissier, M., and Uflacker, M. An interactive platform to simulate dynamic pricing competition on online marketplaces. In *IEEE International Enterprise Distributed Object Computing Conference* (2017).

[39] Simchi-Levi, D., Chen, X., and Bramel, J. Integration of inventory and pricing. In *The Logic of Logistics*. Springer, 2014, pp. 177–209.

[40] Stonebraker, M., and Kemnitz, G. The postgres next generation database management system. *Communications of the ACM 34*, 10 (1991), 78–92.

[41] Talluri, K. T., and Van Ryzin, G. J. *The theory and practice of revenue management*, vol. 68. Springer Science & Business Media, 2004.

[42] Thomas, J. Price-production decisions with deterministic demand. *Management Science 16*, 11 (1970), 747–750.

[43] van Rossum, G. Python programming language. In *Proceedings of the USENIX Annual Technical Conference* (2007).

[44] Wagner, H. M., and Whitin, T. M. Dynamic version of the economic lot size model. *Management Science 5*, 1 (1958), 89–96.

[45] Wolff, R. W. Poisson arrivals see time averages. *Operations Research 30*, 2 (1982), 223–231.

[46] Yeoman, I., and McMahon-Beattie, U. *Revenue management: a practical pricing perspective*. Springer, 2010.

A

Notation Table

Symbol	Description
l	Holding costs per item and period.
c_{fix}	Fixed order costs.
c_{var}	Variable order costs.
$C(b)$	Total order costs for ordering b items.
a_{fix}	Fixed selling price.
δ	Discount factor for future profits.
t	Identifier variable for a time period.
N_t	Random inventory level at the start of period t.
b_t	Number of items ordered at start of period t.
B	Set of admissible order quantities.
$P(i)$	Probability to sell i items in one time period, given enough items are available.
$P(i, a, \vec{s}; \vec{\beta})$	Given a market situation $\vec{s}$ and weights $\vec{\beta}$, the probability to sell i items at price a in one time period, given enough items are available.
X_t	Random number of sold items from start of period $t-1$ to start of period t.
G_t	Random accumulated discounted profit from period t.
n	Inventory level.

$V_t(n)$	Value function: best expected discounted profit from period t with a start inventory of n items.
T	Number of periods of the limited time horizon.
N_{max}	Maximum inventory capacity.
b	Order quantity.
u	Start value for value function: $V_T(n) = u, 0 \leq n \leq N_{max}$.
$b^*(n)$	Optimal ordering decision with delayed orders for inventory level of n.
$b^*_{instant}(n)$	Same as $b^*(n)$ but for instantaneous orders.
A	Set of admissible pricing decision.
a_t	Offer price at time t.
$a^*(n)$	Optimal pricing decision for inventory level of n.
$\vec{s}$	Market situation, vector of merchant offers.
$\vec{x}(a, \vec{s})$	Explanatory variables for price decision a and market situation $\vec{s}$.
K	Number of competitor offers in market situation $\vec{s}$.
$\vec{p}(\vec{s})$	Vector of competitor prices in $\vec{s}$.
$\lambda(\vec{s}; \vec{\beta})$	Estimated mean sales in one period.
$\vec{\beta}$	Weights vector for linear regression.
$\vec{\beta}^*$	Optimal weights for specific training data.
M	Number of explanatory variables.
$\gamma(a)$	Probability for a single customer of buying an item. Used for binomial distribution.
$V_0^{old}(n)$	Result of value function from last computation.
d	Controls how many new decisions are added to adapted decision sets.

Table A.1: List of variables and parameters with description.

Declaration of Originality

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quoations are clearly marked. The master's thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

Potsdam, July 26, 2018

Carsten Walther