Erasmus University Rotterdam
Erasmus School of Economics

Bayesian Optimization for Parameter Tuning of Recommendation Systems

MSc Econometrics and Management Sciences: Business Analytics and Quantitative Marketing

Author:
William Steenbergen

Academic Supervisor:
Dr. Michel van der Velden
Prof. Dr. Ilker Birbil

Practical Supervisors:
Dennie van den Biggelaar
Renée Lenders

August 6, 2019

Abstract

This research explores parameter optimization algorithms for recommender systems at retailers. Since evaluating the performance of a recommender is often expensive both in time and financial aspect, it is required that the optimization algorithm uses as little tests as possible. We tested several methods all in the field of Bayesian optimization: a Gaussian Process approach with a Matern and squared exponential covariance prior and a Tree Parzen Estimator approach with both a normal and uniform prior. The methods use an acquisition function, which determines the next parameter setting that will be evaluated. This research tested a new acquisition function (dynamic expected improvement) that is an adaptation to the popular expected improvement function. It was found that the TPE algorithm consistently finds better optimal values than the Gaussian Process approach. Moreover, the high impact of the prior selection is underlined by the results. It was also found that the dynamic expected improvement acquisition function performs worse than the regular expected improvement acquisition function, but could still have potential for applications with smooth objective functions.

Contents

1 Introduction

Large e-commerce web shops offer millions of products. For a customer, choosing from those products can be a challenging task. To aid customers in this task and in an attempt to cross-sell products, many companies developed recommender systems. These recommender systems aim to show products that fit the customers' needs and thereby increase the number of products sold.

There are many techniques for recommending products (see Ricci et al. (2011) for an overview), and it remains a challenge for companies to select the recommender that best fits their needs. Most recommender systems are based on the assumption that the 'best' products to recommend are products that are similar to products customers have already bought, or products that similar customers bought (Resnick and Varian, 1997). There is no widely accepted evaluation method that tests this assumption. Neither is there an accepted method that measures the performance of a recommender (Gunawardana and Shani, 2009), which makes it impossible to compare different recommenders. There are two reasons that make it hard to define a evaluation method:

First of all, it is not always clear what defines the 'best' recommender. Recommenders can have different purposes, and how well a recommendater performs depends on the purpose of the recommender. For example, a news website might have the function to keep the customer on the website for as long as possible. For an online retailer, a maximal conversion rate of the shown recommendations might be the primary objective. It is not uncommon that the creator of the recommendation system does not exactly know what the purpose of a recommender is, since the relationship of the recommender with the overall main goal of the company (usually to maximize profits) is not always straightforward.

Secondly, it is expensive to gather data about the performance of a recommender. It takes time to implement a recommender, and there is uncertainty about the performance of the recommender before testing. Testing a bad recommender has a negative impact on the customer relationship with the customers that are used for testing. Offline testing, using historical interactions of customers with a recommender, might be used to solve this problem. However, since historical data contains customer interactions with an *old* recommender, the data is inherently biased towards the old recommender in case customers positively interact with the old recommender.

Online testing, by channeling subsets of the customers to different recommenders, prevents this bias but also introduces a third problem: it is difficult and costly to build a pipeline that uses different recommenders for different subsets of the complete set of customers. Moreover, one needs a large number of data points to test a recommender, and since there is a finite number of customers interacting with the recommender, we cannot test a large number of different recommenders.

To mitigate these challenges, we need an algorithm that can determine the best recommender by testing as little different recommenders as possible. Moreover, this optimization algorithm should work for different objective functions,

such that it still works if the objective function is changed by the company due to a changing purpose of the recommender.

This aim of this research study is therefore to

find an optimization algorithm that finds the best recommender with a low number of testcases, where best can be user-defined.

As an approach to this goal, we need to solve multiple issues. First, we have to determine how to evaluate the performance of an algorithm, and how one can construct training and test sets in order to test the performance of a recommender. It is crucial that the algorithm can deal with a flexible objective function, and it should still work if the administrator of the website would change the objective function.

Next, it is important to find an optimization algorithm that uses few test-cases to find the optimal recommender. Moreover, we do not want to test very bad recommenders, since that would hurt customer relationships. So not only do we want to test as little recommenders as possible, we also want to minimize the number of bad recommenders we test.

Since it is relatively cheap to test a recommender on an offline data set, we can test the optimization algorithm in an offline setting. This raises the desire to validate that the performance of the optimization algorithm on the offline test set is generalizable to the online case, which forms another issue of this research.

This research tries to a achieve the main research goal and sub-goals by means of a case study in collaboration with a data science company named Building Blocks, who provided a case for an international online clothes retailer. Building Blocks already has a recommender in place for the retailer, and also has data available on historical interactions with this recommender. It is of interest for them to optimally tune the parameters of this recommender, which is essentially our main goal.

The rest of this paper is structured as follows. Firstly, the case is introduced by explaining the current recommender and setting of the case study. Next, an overview is given on relevant literature on this subject. The paper continues by describing the available data, and gives some insights by presenting descriptive statistics. Next, the used methods are discussed. We test four existing methods, and propose one new method that is an adapted version of one of the four methods. We continue with presenting the results, and end with a discussion and conclusion.

2 Current situation

This research was done in corporation with Building Blocks, who provided a case at an international online clothing retailer. This section introduces Building Blocks, explains what recommender they are currently using, and elaborates on the case from which the data is generated.

2.1 Building Blocks

Building Blocks is a data science company founded in 2013 in Tilburg, the Netherlands. Currently, Building Blocks has offices in Amsterdam and Tilburg. The team consists of approximately 40 data scientists and econometricians and is steadily growing. Building Blocks meets the growing demand of companies to understand their customer behavior by performing diagnostive, predictive and prescriptive analysis, and giving advise on how to act appropriately given this behavior. To do this, Building Blocks develops so-called 'blocks', which are flexible algorithms that can be applied to similar problems that different companies face. Some examples of developed 'blocks' are an algorithm that predicts whether customers churn and why they do so, an algorithm that predicts the probability of a customer buying a certain offer, or an algorithm that predicts the demand for a certain product. Building Blocks focuses mostly on the retail and insurance market.

2.2 Current recommender

Building Blocks currently has a recommender in place that has different parameters. To achieve the main research goal, we will test algorithms that evaluate this recommender with varying parameters. The recommender recommends products by using three methods, dependent on the customer it recommends for. See Figure 2 for a schematic overview of how the recommender works.

Method 1) is used for known customers, and it uses the purchase and click history of a customer to show similar products (content-based recommendation technique). Method 2) is used for unknown customers that are on a product page that more than X other customers clicked on. It uses the purchase and click history of customers that were on the same page and recommends commonly purchased products of those customers (collaborative filtering). Method 3) is used for unknown customers that are on a product page that less than X customers clicked on, and simply shows the most popular products. In this research, we focus only on known customers, so the first method.

Whether a customer is known is defined by asking the question: did a customer click on more than N products in between today and today $-$ M days. M and N are defined by the parameters of the recommender.

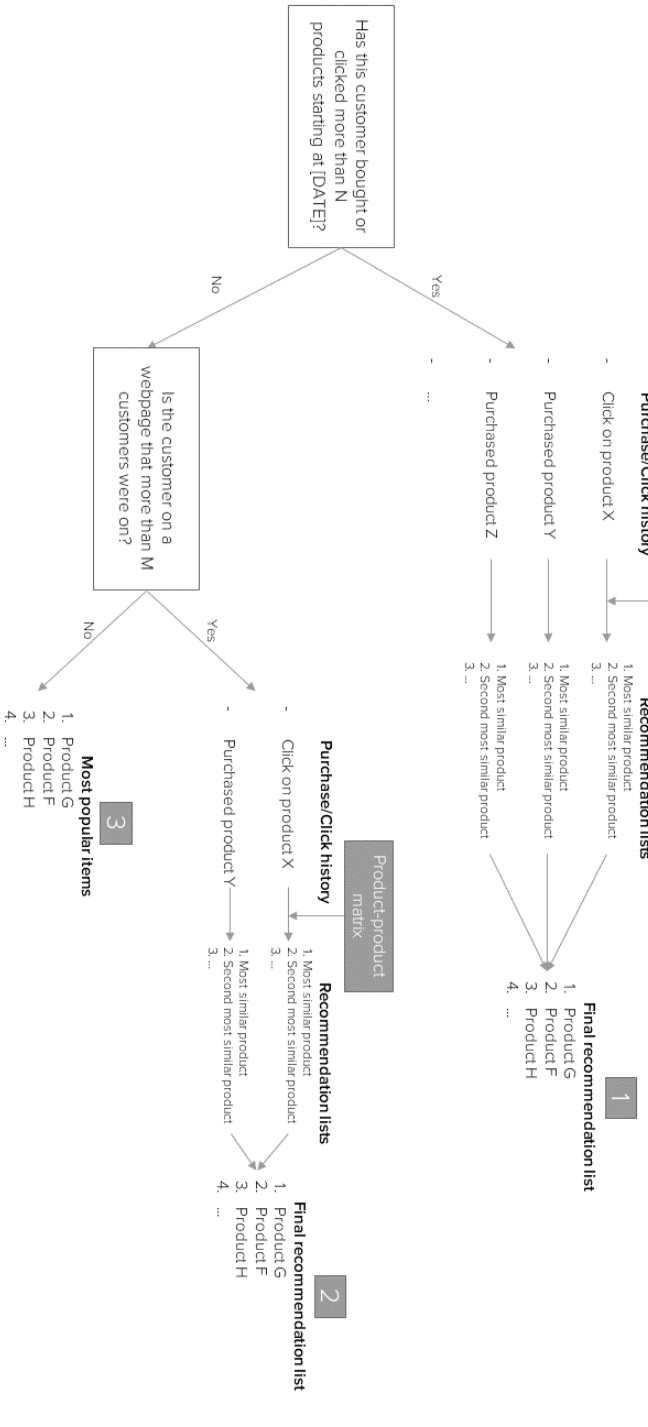

Figure 2: A schematic overview of how a recommendation is generated for a customer

Once a day, the recommender finds all known customers starting at a certain date and generates 25 products/recommendations for all these customers. The recommender does this as following. First, the recommender constructs a product-product matrix, which links a similarity score to all product-product pairs. This score is calculated as following:

$$\begin{gathered} Score(product1, product2) = \\ \alpha\, UserScore(product1, product2) + \beta\, SessionScore(product1, product2) \end{gathered} \tag{1}$$

α and β are parameters that can be set to define the importance of the scores. The $UserScore$ is calculated by looking at what the same user buys over history. If a user buys a t-shirt in the summer and a winter jacket in the winter, the similarity of those two items increases with 1. The $UserScore$ is calculated by adding the contributions of all users that happened after a specific date in history. The specific date is one of the parameters of the recommender. The $SessionScore$ is determined by what items are commonly bought within one session. A session consists of all consecutive clicks and events that a user makes without leaving the retailer's website. In the example of a winter and summer jacket, these products do not get a higher similarity score if not bought in the same session. The recommender also has a parameter that determines how far back we take data into account to calculate $SessionScore$.

For known customers we know what a customer previously bought/clicked, so we can use the similarity matrix to find the top N products that are most similar to already bought/clicked products and use those for N recommendations. All products that have been previously bought and clicked by a customer generate a ranked list of recommendations and their similarity scores, such that a customer with multiple clicks has multiple ranked lists of products and similarity scores. These ranked lists are aggregated by adding the similarity scores multiplied by a weight. The weights are introduced to make a distinction between product lists with similarity scores with respect to a purchased products and product lists with similarity scores with respect to clicked products. The weight ratio is a parameter of the recommender, and usually *purchased* products get a higher weight than from *clicked* products. The weighted average is then used to generate a final recommendation ranking that is used to recommend products.

Recommender 2) counts products that are bought by other customers that were on the same page, and weights purchases in the same manner higher as clicks. It then recommends the products with the highest weighted count. Recommender 3) simply uses the most popular products by counting all clicked and bought products and weighting them similarly as in recommender 1) and 2). Again, we only focus on the first method for known customers in this research, since this is the only method that allows offline testing.

The recommender *system* is defined as the combined mechanism of recommender 1), 2) and 3) and the parameters are defined in Table 1.

#	Parameter	Possible Settings	Current setting
1	α and β	[-∞, ∞]	$\alpha = 1$, $\beta = 0$
2	How many days are taken into account to determine $UserScore$	January 2018 - Now	(Now - 100 days) - Now
3	How many days are taken into account to determine $SessionScore$	January 2018 - Now	(Now - 30 days) - Now
4	How much historical data is taken into account to determine user-item recommendations	January 2018 - Now	(Now - 90 days) - Now
5	How much historical data is taken into account to determine which recommender is used	January 2018 - Now	(Now - 1 year) - Now
6	Weight purchase : click	[-∞, ∞] : [$-\infty, \infty$]	4:1
7	Minimum number of clicks by user to activate recommender 1)	[-∞, ∞]	30
8	Allow cross category recommendations	No/Yes	No
9	Allow recommendations that have already been viewed by the customer	No/Yes	No

Table 1: An overview of the switches in the recommender system and the settings that the data is based on.

The first parameter determines how we set up the product-product similarity matrix. We currently only use the $UserScore$. Parameter 2 determines how many days we take into account when calculating the $UserScore$. It sets how far we want to track users back to see what individual users commonly buy together. The third parameter does the same, but now for calculating the $SessionScore$. We currently look 30 days back in sessions to see what products are commonly bought together in sessions. Parameter 4 determines how many days back we want to take into account to construct the ranked lists of recommendations per customer. The fifth parameter selects how far we go back to determine whether a customer is known.

Parameter 6 selects how important a purchase is in comparison with a click when aggregating the ranked lists of recommendations. The seventh parameter determines the minimum number of clicks that defines a customer as known and decides (in combination with parameter 5) whether we use recommender 1), 2) or 3). Parameter 8 selects whether we allow to recommend a product that is in a different category as the product a customer is currently looking at. Finally, parameter 9 sets whether we allow recommendations that have already been viewed by a customer.

The basic assumption behind recommender 1) is that similarities on what customers already bought are highly correlated with the probability of a recommendation converting to a sell. Recommender 2) assumes that the similarity of products purchased by other customers has a high correlation with the probability of a recommendation converting to a sell. Recommender 3) assumes that the popularity of certain products is highly correlated with the probability of this recommendation converting to a sell. Even though these assumptions sound plausible, recommending in this manner does not necessarily maximize the expected profit from the recommendations, since it does not consider either a conversion rate or profit margins of the recommendation. It just shows products that customers might be interested in based on similarities.

Moreover, the parameters are now chosen arbitrarily, and it is not clear what

the effect of the parameters is on the performance of the recommender system. The goal of this research is to tune the parameters such that it optimizes certain evaluation criteria. These evaluation criteria can change for different web pages and clients, so it is important to make the parameter tuner flexible such that it can be used to optimize different evaluation criteria. If the algorithm is flexible enough, Building Blocks is able to scale the algorithm to different clients with little effort.

2.3 Case study

This research aims to develop such an optimization algorithm by means of a case study for an international online clothing retailer. The retailer mainly operates through their website, and it is therefore of crucial importance that customers view products they are interested in buying.

The core of the website consists of two different webpages: the product listing and the item page. The product listing shows a large list of products, and gives the customer the opportunity to view all products that the retailer offers. Even though the page shows all products, the ranking of the products can be regarded as a recommendation. Since this is often the first page a customer arrives at, the goal of this page is not only to make customers buy the recommended products, but also to strike the interest of the customer and convince him/her that the retailer suits their interests. Next to this, it is interesting to show a wide range of products such that we can learn more about the customers' preferences. When a customer clicks on a product in this product listing, he/she arrives at the item page.

The item page shows a product and its characteristics like its price, available colors, and size options. When the customer scrolls down on this page, he/she views a box with three recommendations. The customer can choose to view three new recommendations by clicking on an arrow. When a customer clicks on the recommendation a box pops up that displays the characteristics of a product and gives the user the option to put it in their basket.

As can be inferred from the previous paragraphs, the recommenders on the different web pages on the website have different functions. On the product listing page, the recommender has the function of showing products that strike the interest of the customer, but also products that allow for exploration of the customers' preferences. The recommender on the item page has the sole purpose of increasing revenue or profits. This means that the optimization algorithm should work with optimizing different criteria, depending on the purpose of the recommender that is optimized. The next section explains the current state of research on recommendation systems and its parameter tuning.

3 Literature overview

There is a long-lasting history of research on recommendation systems. Ricci et al. (2011) give an overview of recommendation techniques, and point out

that the most common recommendation methods can be divided in 6 types of algorithms: content-based, collaborative filtering, demographic, knowledge-based, community-based and hybrid recommender systems. The recommender used by Building Blocks is a hybrid between the content-based and collaborative filtering method. Since this research mainly focuses on parameter optimization of recommenders, this section does not focus on the recommendation method itself but instead on the evaluation and parameter tuning of the recommendation system.

3.1 Evaluation of recommendation systems

Since the evaluation criteria depend on the function of the recommendation system, there is no universally accepted method to evaluate recommendation systems (Gunawardana and Shani, 2009). However, there has been some research that focuses on developing a testing mechanism (e.g. Herlocker et al. (2004) and Gunawardana and Shani (2011)). Most researchers argue that online testing is the only method that gives truly unbiased results, but since it is so costly they all focus on a method that avoids this.

Gunawardana and Shani (2011) mention that it is of crucial importance that the test setting mimics online testing. They propose several methods of offline testing, which are all based on creating a train and test set from the available offline data set. The idea is that the test set closely resembles an online test, such that if the algorithm performs well on the test set, it is also likely to perform well in an online test setting.

Suppose one has a historical dataset that contains sessions from many customers, recommendations they viewed in this session, and whether they clicked or bought these recommendations. Gunawardana and Shani (2011) then propose 4 different methods to split this set in a test and train set. All methods hide choices from customers, and the hidden choices will form the test set.

Firstly, the ideal option would be to randomly sample test users, randomly sample a time just prior to a user action, hide all choices of all customers after this time, and then recommend something to this user. One would change the set of given information every time this is done, which becomes computationally expensive. Therefore, it might be better to sample a set of test users, then sample a single test time, and then hide all items after the sampled test time for all test users. This means that the sampled test time is not always right before a user takes action, and the recommender would not take into account new data generated in between the users' action and the sampled test time. A third option would be to sample a test time for every user that is just before an action, such that the test times are not the same. This would assume that the choice of a customer is not dependent on the absolute time of the decision, which would be a reasonable assumption in the situation of an online retailer. Finally, one can select N choices, and then hide N choices for all customers. This assumes that temporal aspects of user selections are unimportant.

Li et al. (2011) take a different approach, and describe the problem as an 'off-policy policy evaluation problem', which draws on the fact that we have a

data set where a certain policy (the old recommendation system) was used, and we want to find out the effect of a new policy. A simple solution to this problem could be to simulate data and test the new policy in this simulation. However, it is difficult to make the simulation unbiased, since using the data from the old policy to simulate would induce bias. Therefore, Li et al. (2011) propose a *replay* method to evaluate a new policy.

The replay method works on the assumptions that there is some random distribution D from which interactions with recommendations are drawn independently and identically distributed, and the old policy chose the recommendations uniformly at random. The first assumption might not hold, since customer's choices probably depend on previous actions. The second assumption presents an even bigger challenge. The recommender does not select recommendations uniformly at random, but bases them on customer characteristics. One could use rejection sampling to modify the data such that it becomes more random, but only if there is enough data available to allow this. In our case a recommender often recommends the same products, and some products are never recommended. If a recommender that we are interested in testing then would recommend a product that has never been recommended before, this creates a problem. Since there are many products in the store, the probability of this happening is far from zero. In this research we also explore the possibilities of rejection sampling to satisfy the assumptions.

If both assumptions are satisfied, the replay method works as following. First, it loops through all 'events' (in our case these are time-points where recommendations are shown) and selects T 'events' where the (resampled) data gives the same recommendation as the recommendation algorithm we are interested in testing. Since the (resampled) data gives recommendations uniformly at random, the probability of selection a recommendation is $\frac{1}{K}$, independent of everything else. One can then prove that evaluating the new recommender system against T real-world events from D is exactly the same as evaluating it against T selected 'events' by using the replay method, such that this method gives the same results as online testing.

3.2 Parameter tuning of recommendation systems

Parameter tuning of recommendation systems is a difficult task, since it is expensive to test multiple parameter sets. Moreover, the performance as a function of the parameters is a black box in the sense that we do not know anything about the function like the gradient or hessian. Therefore, we are limited in our choice to algorithms that do not use any of this information.

Often, parameter tuning is done by expert/human experience, without much quantitative reasoning (Snoek et al., 2012). Automatic methods are evolving, especially in machine learning applications that are characterized by long and costly train and test times. One method that seems to surpass other methods and also surpasses human/expert parameter tuning is Bayesian Optimization (Snoek et al. (2012), Brochu et al. (2010), Shahriari et al. (2016)). Bayesian optimization builds a model of the evaluation function by choosing a prior and

combining it with the data by using Bayes' rule to find a posterior function. This posterior function is then optimized. It chooses new datapoints by using an acquisition function, that uses information from the posterior function to decide which point is most interesting to try out.

Other optimization methods such as genetic algorithms or simulated annealing are also an option, but they need more function evaluations since they do not spent much computation time deciding what the next evaluated point will be, which makes them expensive to use.

4 Data

The data in this research are data about customers of an online retailer. The data contains sales, clicks and recommendation views (see Table 2 for all variables) from 16 months, from January 2018 to May 2019 and includes $1,688,218$ distinct customers, of which $374,863$ viewed at least one recommendation. For evaluating the recommender, we only used customers that viewed more than 5 recommendations, since there are not enough data to train and test on customers with less activity. This leaves us with $26,279$ distinct customers to evaluate on. We used all possible customers to create the product-product matrix. We assume that an optimal parameter setting for this smaller set will also be (close to) optimal for the larger set, since we do not expect customers that see recommendations more often to act differently on the recommendations than customers that do not see them often. Table 2 displays the variables that we have information on.

Variable	Description	Range
i	The ID of the customer	-
t	The date and time a session started	2019-01-01 to 2019-05-01
ID	The ID of the product a customer is looking at	-
H	The click number in the session of the customer	$[1, \infty]$
Event	Displays the event	View, Click, Add to cart, Remove from cart, Purchase
P, C, AC, RC	A multiset of product ID's a customer purchased (P), clicked on (C), added to cart (AC) or removed from cart (RC)	$P, C, AC, RC \in$ set of all product ID's
R	A set of product ID's that were recommended in case the event is a recommendation view	$R \in$ set of all product ID's

Table 2: The available variables in the data set

4.1 Descriptive statistics

First, we highlight some details about all customers in the data set and link it to potential hypotheses or considerations. We follow with a section on the customers that are used in this research (the $26,279$ customers) to evaluate the performance of different recommenders.

4.1.1 All customers

On average, customers that never saw a recommendation bought 0.088 products and put 0.28 products in the cart. It seems that customers who at least view one recommendation are more interesting for the company, since on average they bought 0.45 products and put 1.42 products in the cart per customer. This gives an indication that recommendations influence the clicking and purchasing behavior of customers, and underlines the importance of showing the right recommendations. The activity of these customers is used for creating the product-product matrix, but directly for evaluating different recommenders.

4.1.2 Customers used for evaluating

On average, customers that are directly used for evaluation of recommenders see 20.34 recommendations per session. After these recommendations, customers on average buy 0.32 products, of which 0.024 are products recommended in the same session. Customers on average click on 0.94 products after seeing a recommendation, of which 0.480 are recommended products. Finally, customers add on average 0.90 products (of which 10.7% are from recommendations) and remove 0.45 products to/from the cart. This shows that recommendations especially seem to influence the clicking behavior of customers. Since more than 50% of the products clicked after recommendations are recommended products, we could conclude that customers engage with the recommendations. However, considering the low percentage of products that were purchased after being viewed as a recommendation (7.5%), customers consider the recommendations not always good enough to also convert to a sell.

4.2 Data preparation

To measure a recommender's performance (see Section 5.2), we want to look at how recommendation views influence customer's decisions. Therefore, we want to score the recommendations based on the actions customers took after a recommendation view. To do this, we prepared the data such that every row displays a recommendation view and the impact of this view on a customer's behavior. This is displayed in Table 3.

i	$t_{session}$	$R_{i,t}$	$P_{i,t}$	$H^P_{i,t}$	$C_{i,t}$	$H^C_{i,t}$	$AC_{i,t}$	$H^{AC}_{i,t}$	$RC_{i,t}$	$H^{RC}_{i,t}$

Table 3: The prepared data set and a sample line. Note that most variables are (multi)sets.

In words, every row contains a moment that customer i sees recommendations $R_{i,t}$ at time t, in a session that started at $t_{session}$. This row also contains the product ID's that this customer bought after seeing these recommendations ($P_{i,t}$), and the corresponding hitnumbers of these purchases ($H^P_{i,t}$). The hitnumber is a variable that denotes the number of the click the event was for a

customer in one session. So the first click in a session has hitnumber 1, and the 10th click, perhaps a product that was added to the cart or a purchase, has hitnumber 10. Next to this, a row includes products that this customer clicked on ($C_{i,t}$), added to cart ($AC_{i,t}$) and removed from cart ($RC_{i,t}$) after seeing the recommendations and their corresponding hitnumbers respectively ($H^{C}_{i,t}$,$H^{AC}_{i,t}$ and $H^{RC}_{i,t}$). A (multi)set notation was chosen over matrix notation for easier interpretation and notation (Section 5.2).

Note that only products are included in a row if they are bought/clicked/added to cart after a recommendation has been seen (determined by their hitnumbers), but only if this happens in the same session. Since a customer can view multiple recommendations in a single session, there can be multiple rows with the same $t_{session}$. It follows from this that it can happen that different rows contain the same purchases. For example if someone sees a recommendation at hitnumber 3 and 5 and purchases a product as click 6, this product will be in the row of the recommendations viewed as hitnumber 3 but also in the row of the recommendations viewed as hitnumber 5. Since all the performance of all rows are measured independently from each other, this creates the problem that a purchase is counted towards the performance measurement multiple times. We deal with this potential problem in Section 5.2.

4.2.1 Data generation

For the case of online testing, it is interesting to know how much data are generated per day/month such that we can roughly predict for how long we should run the (expensive) online testing to generate enough data for us to have significant results. The most important measure is the average number of viewed recommendations per day. In the set of customers that are eligible for use in our research, we have 2512 recommendation views on average per day. This determines the number of observations we have and can evaluate a recommender on.

5 Methods

To find out whether there is a flexible optimization algorithm that finds a good hyper-parameter configuration with a low number of testcases, different methods are tested to compare which one works best. The different optimization methods that are tested all fall in the field of Bayesian optimization, but differ in the way they specify the prior. This research will test a Gaussian Process prior (GP) approach with a Matern kernel and an ARD kernel, and a Tree Parzen Estimator (TPE) approach with a lognormal and uniform prior. See Section 5.3 for an explanation of all methods.

5.1 Creating the test and train set

As explained in Section 3, there are three different methods of dividing the data into a train and test set, depending on three different assumptions. First, we want to stress that the most important decision factor is that the train/test split mimics the online testing case as much as possible. The first method chooses one test time and then generates recommendations for all customers using the information up to that time-point. This method assumes that information generated in between two recommendations is not important. This does not strictly hold in our data set, since clicks and purchases between recommendation views not only influence the product-product similarity matrix, but also the ranked lists of recommendations.

The second method samples a test time for every user and then generates a recommendation for that user using all the information up to that moment. assumes that the way a customer reacts to a recommendation is not dependent on the absolute time. The problem of this method is that we have to generate a product-product matrix for every interaction for every test user, which becomes computationally intensive. Moreover, it is not similar to the online testing case since currently Building Blocks only generates a product-product matrix once every day.

The third method hides N recommendations/interactions for all customers and generates recommendations based on the non-hidden recommendations, while choosing only one time point to generate the product-product matrix. This method does not use the time sequence of the recommendations and is more of a cross-sectional approach. It therefore assumes that temporal aspects and sequence of recommendations have no influence on customers' reactions on recommendations, which is unlikely to hold, since we expect that customers' reactions will depend on previously seen recommendations and clicked/bought products, so the sequence is certainly important.

The method that seems most similar to the online testing case without being computationally too expensive seems to be method 1), where we set one time-point for all customers that divides all customers in a train and test set. We choose the 1st of April 2019 as a time-point, and all interactions before this time-point belong to the training set, and after this time-point will be in the test set. We deviate from online testing since we generate the product-product matrix only once (at the 1st of April), and do not take into account information between this time-point and the time-point of the actual recommendation. For example if a customer sees a recommendation the 5th of April, we do not take into account any information between the first and 5th of April, while in the online case we would. By only testing customers that see recommendations shortly after the chosen time-point, we can minimize the difference and still come close to the online setting. We therefore chose one specific time-point (1st April 2019) for which we create the product-product matrix, and then only tested on customers who viewed recommendations before April 10 2019. Of this set of customers, we randomly chose 1000 customers due to computation time constraints. We chose 1000 customers randomly instead of the first 1000 customers with respect

to time since it more accurately models the online application if we have more computing power. The impact of this is discussed in Section 7.

In addition, the replay method, as was explained in Section 3, was considered. We should resample the data in such a way that the recommendations are uniformly random. However, since the products that that the retailer offers change often, and there is a huge number of different products sold (over $10,000$) this would mean that we have to resample a very high number of times to create a uniform distribution. Sampling such a large number of data points defeats its purpose by inducing bias and is also unpractical in terms of computation time.

5.2 Measuring recommender performance

Once we have a train and test set, we need to decide how to evaluate a recommender by defining the objective function. Part of the research goal states that the user, the retailer in this case, should be able to define the objective function. Therefore, the evaluation criteria should be defined in a flexible manner. We use the same notation as in Section 4, but we will elaborate more on the notation first.

We define an *event* as a moment where a customer sees recommendations and thus has the choice to click on it, put it in the cart, buy it, or ignore the recommendation. By definition, a customer views a recommendation if the recommendation is visible on the computer screen for at least 3 seconds.

We chose to use a more unconventional notation by using mathematical (multi)-sets instead of the more conventional matrices, because this is more convenient for this application, and for implementation. $R_{i,t}$ for $i = 1, 2, ..., N$ and $t = 1, 2, ..., T_i$ is the set of recommendations which customer i views at time index t. Note that T_i depends on person i, so not all individuals have an equal number of events. To give an example of $R_{i,t}$, if the recommender calculates that individual $i = 2$ at time $t = 3$ should be recommended products $\{4, 6, 8, 10\}$, then $R_{2,3} = \{4, 6, 8, 10\}$.

We define P_{i,τ_s,τ_e} as the (multi)set of products that customer i has bought in between τ_s and τ_e. It can be a *multi*-set since it is possible for a customer to buy the same item twice or more. C_{i,τ_s,τ_e} is defined as the (multi)set of products that customer i has clicked on between τ_s and τ_e. Index r indexes the product in the multisets, in order to loop through the set. It is convenient to set τ_s to the moment that the recommendation that is currently evaluated is viewed, so in further discussion we set $\tau_s = t$.

The evaluation function, and thus the objective function in the optimization problem, is an additive function that is constructed of multiple elements denoted by capital letters weighted by $\beta's$ that can be specified by the user:

$$\begin{aligned} y = &\beta_A A(\phi_A) + \beta_B B + \beta_C C(\phi_C) + \beta_D D(\phi_D) + \beta_E E(\phi_E) \\ &+ \beta_F F(\phi_F) + \beta_G G(\phi_G) + \beta_H H(\phi_H) + \beta_I I(\phi_I) \end{aligned} \tag{2}$$

The first element ($A(\phi_A)$) captures whether someone bought the recommended products, weighted by the profit we make on this, and by how fast

he/she bought something after seeing the recommendation:

$$A(\phi_A) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|P_{i,t,\tau_e,r}|} \underbrace{I[P_{i,t,\tau_e,r} \in R_{i,t}]}_{\text{Bought recommendation}} \underbrace{\pi^{\phi_A}_{P_{i,t,\tau_e,r}}}_{\text{Profit}} \underbrace{\frac{1}{\sqrt{H_{P_{i,t,\tau_e,r}} - H_{R_{i,t}}}}}_{\text{Timing relevancy}} \tag{3}$$

Where $I[\cdot]$ is the indicator function, $\pi_{P_{i,t,\tau_e,r}}$ is the profit of the recommended and purchased product $P_{i,t,\tau_e,r}$. ϕ_A is a weight that determines how important the profit weights are.

The first part of the equation makes sure that we only score products that are purchased after they are recommended. The company can set τ_{end} arbitrarily, but it is recommended to set it to the end of the session $(t+1)$, such that we only take into account products purchased in the same session as the viewed recommendations.

The second part weighs the product by its profit (dependent on how we set ϕ_A), such that products with a high profit are more important than product with a low profit. Note that ϕ_A is raised to the power of $\pi_{P_{i,t,\tau_e,r}}$, such that $\pi^{\phi_A}_{P_{i,t,\tau_e,r}}$ becomes 1 if weight ϕ_A is 0.

The last part of the equation is there to take into account that products that are bought right after viewing a recommendation are weighted more than products that are purchased many clicks after a recommendation. $H_{P_{i,t,\tau_e,r}}$ is the hitnumber that someone bought product $P_{i,t,\tau_e,r}$, and $H_{R_{i,t}}$ is the hitnumber that someone viewed the recommendation $R_{i,t}$. We take the inverse of the square root of the difference to give products that were bought earlier after the recommendation more weight than products bought long after the recommendation, also to account for the multi product problem as mentioned in Section 4.2.

The second element $(B(\phi_B))$ captures products bought by a customer that are not recommended, but are in the same category as one of the recommendations. We include this element in order to capture the extent of how much a recommendation inspires someone to buy something similar to the recommendation. If we define $P^c_{i,t,\tau_e,r}$ as the multiset of product categories of corresponding products $P_{i,t,\tau_e,r}$, and $R^c_{i,t}$ as the categories of the recommendations $R_{i,t}$, then we define $(B(\phi_B))$ as:

$$B(\phi_B) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|P^c_{i,t,\tau_e,r}|} \left(I[P^c_{i,t,\tau_e,r} \in R^c_{i,t}] \pi^{\phi_B}_{P_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{P_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \tag{4}$$

$B(\phi_B)$ works very similar to $A(\phi_A)$, but now comparing categories instead of the product themselves.

Thirdly, we want to score to what extent recommendations inspired customers to buy product in general. Therefore, $C(\phi_C)$ scores products that were

purchased but not recommended, and also in another category as the recommendations:

$$C(\phi_C) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|P^c_{i,t,\tau_e,r}|} \left(I[P^c_{i,t,\tau_e,r} \notin R^c_{i,t}] \pi^{\phi_C}_{P_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{P_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \quad (5)$$

A, B and C score all purchased products. However, we would also like to score clicked products and products that customers added to the cart. We do this in exactly the same manner as we scored purchased products which gives us D, E and F for clicked products and G, H and I for products added to cart. See the appendix for the complete formulas of D to I. These scores are essentially key performance indicators (KPI's).

We want to combine the KPI's in a single recommendation score (RS). Since this is the objective function, we will notate this as y from now on. We do this with an additive model that allows the company to define which KPI's they find most important:

$$\begin{aligned} y = RS = &\beta_A A(\phi_A) + \beta_B B + \beta_C C(\phi_C) + \beta_D D(\phi_D) + \beta_E E(\phi_E) \\ &+ \beta_F F(\phi_F) + \beta_G G(\phi_G) + \beta_H H(\phi_H) + \beta_I I(\phi_I) \end{aligned} \quad (6)$$

Where the β's are the weights the company can give to different KPI's depending on which they consider most important. Typically, a company would set $\beta_A > \beta_B > \beta_C$, to ensure that bought recommendations are more important than products bought in the same category of recommendations and products bought in a different category. The same holds for β_D and β_G for respectively clicks and products added to cart.

It is important to note that the score should be used to compare algorithms in relative sense. The absolute value of the score is not of interest since the weights are determined arbitrarily by the company.

5.3 Optimization methods

The methods that are used can all be categorized or are very similar to the field of Bayesian optimization, and the difference between them lies mainly in the prior that was chosen. Four different methods were chosen: Bayesian optimization with a Gaussian Process with two different priors and Bayesian optimization with a Tree Parzen Estimator algorithm (TPE) with two different priors. All methods are built in Python, or readily available in Python packages. We will first give an high-level overview of Bayesian optimization and then go into the different priors and algorithms that are used in this research.

5.3.1 Bayesian optimization and the acquisition function

Now the evaluation objective has been defined, it is clear what the objective function is to maximize. We denote the objective function as $y = f(\boldsymbol{x})$, where

Algorithm 1 The Bayesian optimization algorithm

Initialization
Choose prior $P(\boldsymbol{x})$
Choose acquisition function $Aq(\boldsymbol{x})$

for $i = 1$ to I **do**
Choose $\boldsymbol{x}_{i+1} = \arg\max_{x} Aq(\boldsymbol{x})$
Evaluate $y_{i+1} = f(\boldsymbol{x}_{i+1})$ and append to available data set D_i
Update posterior function $P(y_{i+1}|\boldsymbol{x}_{i+1}, D_i)$ by using Bayes' rule
end for
Return $\boldsymbol{x}$ with either the highest sampled $f(\boldsymbol{x})$ or the highest in the posterior function $f(y|\boldsymbol{x})$.

Algorithm 1: A simplified version of the Bayesian optimization algorithm. D_i contains all the sampled points $(\boldsymbol{x}_a, y_a)$ for $a = 1, 2, ..., i$.

$\boldsymbol{x}$ is the parameter settings, $\boldsymbol{x} = [Par2, Par4, Par6]$. In this research we only change parameters $2, 4$ and 6 from Table 1 for simplicity. We compare several optimization methods in terms of the number of function evaluations, the quality of the optimal solution and the stability and riskiness of the results. Due to the low number of needed function evaluations and its good performance in similar situations, we chose to test several types of Bayesian optimization methods as our proposal to achieve the main research goal.

Bayesian optimization works in an iterative manner (see Algorithm 1). To initialize, a prior must be chosen that models the evaluation function without any data ($P(\boldsymbol{x})$). Popular priors are a Gaussian Process (GP) (Brochu et al., 2010), Tree Parzen Estimator (TPE) (Snoek et al., 2015) or a regression forest (Hutter et al., 2011). The latter is not properly implemented yet so we will only focus on the first two algorithms in this research.

After the prior is chosen, we need an acquisition function that determines what the new recommender settings ($\boldsymbol{x}$) are which we want to test. The acquisition function is a function of the posterior function, which is a combination of the prior and the available data according to Bayes' rule. In literature the most widely used acquisition function is expected improvement (EI) and we use this too for this research. Expected improvement balances the uncertainty of the posterior function with the need to exploit and find the best solution:

$$EI_{y^*}(\boldsymbol{x}) = \int_{-\infty}^{\infty} \max(y^* - y, 0) p_M(y|\boldsymbol{x}) dy \tag{7}$$

where y^* is some value for the objective value which is used as relative point from which we define improvement. M is the model that describes how $\boldsymbol{x}$ impacts y: the posterior model under which the expected improvement is

measured.

In every iteration, we choose the point that shows the highest expected improvement, and then update the posterior function by combining the new data with the prior/old posterior. We then repeat by choosing the next point determined by the acquisition function that uses the new posterior. This research tests the performance of two different approaches to defining a prior, the Guassian Process approach and the Tree Parzen Estimator approach. For both approaches, two different priors are tested.

5.3.2 Gaussian process prior

The Gaussian process (GP) prior approach models $p(y|\boldsymbol{x})$ directly, by using a GP. A GP is like a usual function, but instead of returning a scalar it returns a mean and variance of a normal distribution (see Figure 3 for an example).

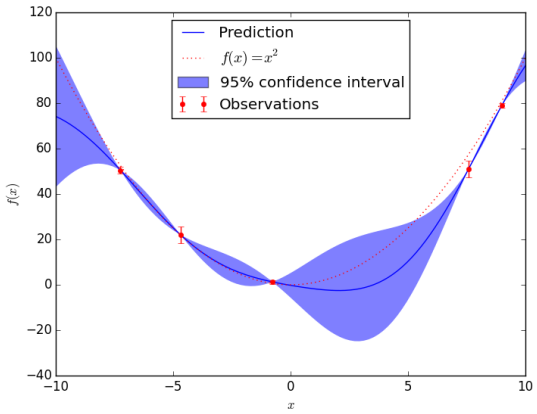

Figure 3: An example of a Gaussian Process

A GP therefore needs a mean and covariance function as prior. We choose a null function (function constant at 0) for the mean prior, since this seems to work relatively well in similar situations (Martinez-Cantin et al., 2009). For the covariance function, there are multiple options of which some include hyperparameters such as the Matern kernel (Matérn, 2013) and the squared exponential kernel with automatic relevance determination (ARD) hyperparameters (Rasmussen, 2003). Both options are tested to find out which works better in this situation.

The Matern kernel is defined as following:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \frac{1}{2^{\zeta-1}\Gamma(\zeta)}(2\sqrt{\zeta}||\boldsymbol{x} - \boldsymbol{x}'||)^{\zeta} H_{\zeta}(2\sqrt{\zeta}||\boldsymbol{x} - \boldsymbol{x}'||) \tag{8}$$

Where ζ is the hyperparameter that determines the smoothness of the function, $\Gamma(\cdot)$ is the Gamma function and $H_{\zeta}(\cdot)$ is the Besssel function of order ζ.

Matérn (2013) chooses $\zeta = 1.5$, and it is common in literature to leave it at this value, so we do too. The squared exponential kernel with ARD hyperparameters is defined as:

$$k(\boldsymbol{x}, \boldsymbol{x}') = exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}')^T diag(\boldsymbol{\theta})^{-2}(\boldsymbol{x} - \boldsymbol{x}')\right) \tag{9}$$

Where $diag(\boldsymbol{\theta})$ is a diagonal matrix with d entries $\boldsymbol{\theta}$ along the diagonal. Intuitively, if a particular θ_s has a small value, the kernel becomes independent of the s-th input.

The hyperparameters can be determined by trying the method on a few random samples and maximizing the log-likelihood of the evidence given $\boldsymbol{\theta}$ or ζ. To aid this, one can set a hyperprior which is often a log normal prior, then get some random samples, and then combine this into a posterior with Bayes' rule. We choose $\boldsymbol{\theta}$ such that it maximizes the log likelihood of the posterior function (ust as Rasmussen (2003)).

As acquisition function we choose the expected improvement which in the case of a Gaussian process prior can be written as (Brochu et al. (2010), Jones et al. (1998), Močkus (1975)):

$$EI_{y^*}(\boldsymbol{x}) = \begin{cases} (\mu(\boldsymbol{x}) - y^*)\Phi(Z) + \sigma(\boldsymbol{x})\phi(Z) & \text{if } \sigma(\boldsymbol{x}) > 0 \\ 0 & \text{if } \sigma(\boldsymbol{x}) = 0 \end{cases} \tag{10}$$

where

$$Z = \begin{cases} \frac{\mu(\boldsymbol{x}) - y^*}{\sigma(\boldsymbol{x})} \text{if } \sigma(\boldsymbol{x}) > 0 \\ 0 \qquad\qquad\qquad\qquad \text{if } \sigma(\boldsymbol{x}) = 0 \end{cases} \tag{11}$$

where $\mu(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ are the mean and the variance posterior function at $\boldsymbol{x}$, and Φ and ϕ are the CDF and PDF of a standard normal distribution respectively.

Equation 10 shows that the expected improvement consists of two parts. The first part ($(\mu(\boldsymbol{x}) - y^*)\Phi(Z)$) represents the average improvement, and thus can be seen as 'exploitation'. However, it is also important to explore the function and decrease uncertainty. The second part ($\sigma(\boldsymbol{x})\phi(Z)$) represents the variance of the posterior, and therefore ensures the acquisition function also takes into account exploration.

Since it is not time-intensive to evaluate $EI_{y^*}(\boldsymbol{x})$, we can optimize this rather easily. We optimize the acquisition function using DIRECT (Jones et al., 1993), a deterministic, derivative-free optimizer.

5.3.3 Gaussian process prior with adaptation

The current literature is mostly focused on improvements with respect to the kernel that is chosen for the variance prior. This research takes another direction by changing the acquisition function. We have such a low number function evaluations at our disposition that exploring the posterior function might not be as useful since we have little iterations to exploit the information gathered.

To solve this issue, we propose an adaptation to the expected improvement acquisition function, which we will call the 'Dynamic Expected Improvement' (DEI). In the the Gaussian prior setting, the DEI is defined as:

$$DEI_{y^*}(\boldsymbol{x}) = \begin{cases} (\frac{i}{I})(\mu(\boldsymbol{x}) - y^*)\Phi(Z) + (1 - \frac{i}{I})\sigma(\boldsymbol{x})\phi(Z) & \text{if } \sigma(\boldsymbol{x}) > 0 \\ 0 & \text{if } \sigma(\boldsymbol{x}) = 0 \end{cases} \tag{12}$$

where

$$Z = \begin{cases} \frac{\mu(\boldsymbol{x}) - y^*}{\sigma(\boldsymbol{x})} \text{if } \sigma(\boldsymbol{x}) > 0 \\ 0 \qquad\qquad\qquad\qquad \text{if } \sigma(\boldsymbol{x}) = 0 \end{cases} \tag{13}$$

where I is the total number of iterations of the Bayesian optimization algorithm, and i is the current iteration. The DEI is very similar to the EI, but we now weigh the exploitation and exploration part of the function by the iteration we are in, therefore being dynamic with respect to the iteration. We will illustrate the idea by walking through the iterations. At the first iteration of the Bayesian optimization algorithm, $i = 1$ and in this research $I = 20$. That means that the $(\mu(\boldsymbol{x}) - y^*)\Phi(Z)$ part (exploitation) gets a weight of $\frac{1}{20}$, and the $\sigma(\boldsymbol{x})\phi(Z)$ part (exploration) a weight of $\frac{19}{20}$. That means that at the start the algorithm will be almost solely exploring. In the final iteration, $i = 20$ and $I = 20$ which means the algorithm will solely focus on exploiting. In the middle, $i = 10$ and $I = 20$ which means that exploration and explotiation have the same weight, and the function returns effectively the same as EI.

This means that at the start the algorithm only chooses based on the variance function, and not take into account the quality of solutions. This means that the algorithm will search in the place we have the least information about and might thus find very bad solutions. This brings along risk at the start of finding very bad solutions, but this risk might be worth it if we can use the information to exploit in the end. We test the performance of this adaptation with a GP prior and Matern kernel, such that we can compare the performance with and without adaptation.

5.3.4 Tree parzen estimator prior

The TPE method takes a different approach. It does not directly model $p(y|\boldsymbol{x})$, but it instead models $p(\boldsymbol{x}|y)$ and $p(y)$ and uses Bayes' rule to calculate $p(y|\boldsymbol{x})$. For modeling $p(\boldsymbol{x}|y)$ and $p(y)$, it uses a 'tree parzen estimator', which uses a tree like method similar to decision trees. First, several points have to be sampled, and then these points are divided in two by choosing a boundary level y^* for the y values of these points:

$$p(\boldsymbol{x}|y) = \begin{cases} g(\boldsymbol{x}) & \text{if } \ f(\boldsymbol{x}) > y^* \\ l(\boldsymbol{x}) & \text{if } \ f(\boldsymbol{x}) \leq y^* \end{cases} \tag{14}$$

$l(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are modeled in the following way. First, the user chooses a prior for the hyper-parameters, that describes the expectation of what values

for the hyperparameters will perform well. The options for hyperparameters are theoretically unlimited, but in practise a uniform, Gaussian or log-uniform distribution is usually chosen (Bergstra et al., 2011). Regardless of the prior chosen, the TPE 'places' Gaussians on all the points that are sampled and satisfy $f(\boldsymbol{x}) < y^*$ when modeling $l(\boldsymbol{x})$ or satisfy $f(\boldsymbol{x}) \geq y^*$ when modeling $g(\boldsymbol{x})$. The width of the Gaussians are chosen by setting σ^2 to the greatest of the distances to the neighbor points. We then define $l(\boldsymbol{x})$ or $g(\boldsymbol{x})$ by taking an equally-weighted mixture of the Gaussians and the chosen prior. A downside of the TPE algorithm is that it does this for every hyperparameters, therefore losing ability to describe interaction effects between the hyperparameters. Figure 4 visualizes this process.

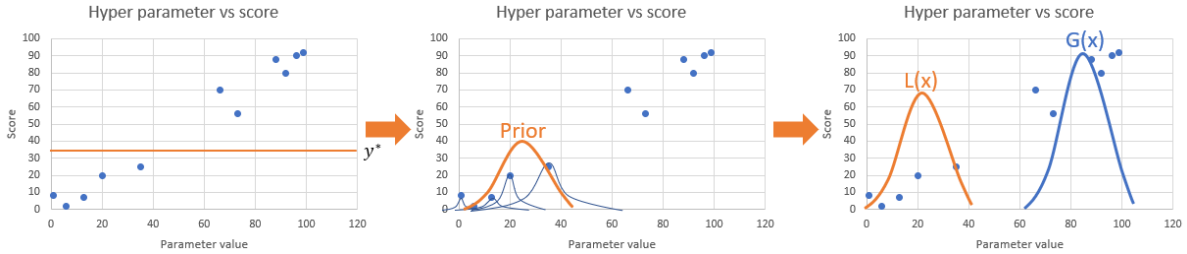

Figure 4: Example of the TPE algorithm with *dummy data*. The first graph shows the cutoff point y^*, the second shows the Gaussians 'placed' on the sampled points together with the prior and the third graph shows $l(\boldsymbol{x})$ and $g(\boldsymbol{x})$

So instead of modeling the complete evaluation function, TPE iteratively models two probability distributions of which one is used to find the maximum.

The acquisition function is again the expected improvement. Bergstra et al. (2011) show that, using Bayes' rule and some substitutions it can be writtenr as:

$$EI_{y^*}(\boldsymbol{x}) \propto \left(\gamma + \frac{l(\boldsymbol{x})}{g(\boldsymbol{x})}(1-\gamma)\right)^{-1} \tag{15}$$

where $\gamma = p(y < y^*)$, which is the quantile of the observed y function we use to model $l(\boldsymbol{x})$. This expected improvement function means that we want to sample points that have a high probability under $g(\boldsymbol{x})$ and a low probability under $l(\boldsymbol{x})$. This means that the TPE algorithm weighs exploitation more importantly than exploration.

In this research we tested two priors, a uniform and a log-normal prior. For the uniformative uniform prior, we choose $Par3 \sim Unif(10, 220)$, $Par4 \sim Unif(10, 190)$, and $Par6 \sim Unif(0, 9)$. For the normal prior, we used a discrete log-normal distribution to use that fact that from a business expertise it was expected to have lower values for the parameter (and non-zero), since recommendations with these settings are more tailored to recent activities

from customers. For the priors, we chose for $Par3 \sim lognormal(2.6, 1.2)$, for $Par4 \sim lognormal(2.6, 1.2)$, and for $Par6 \sim lognormal(2, 1)$.

5.4 Validation and generalization to online case

Finally, it is important to validate that good performance with offline testing can be generalized to good performance with online testing. The most crucial difference between the offline test set and the online set is that the offline set contains reactions from customers on one specific (old) recommender, and so we test the performance of a recommender of interest against behavior of people that did not actually see the recommender that we want to test. This is not a problem since we are only interested in the method and not the optimization result itself, but we must pay an effort to verify that the method works on both data sets.

We can never be certain that the algorithms work in the online case, since we can not test online. There are two reasons why the online case is different from the offline case: 1) in the offline case the customers have not seen the tested recommendations, and only respond to the old recommender and 2) in the offline case the train/test set is different than in the online case (see Section 5.1).

It would be interesting to examine whether the found optimal parameter settings in the offline case resemble the settings of the recommender that the test customers actually reacted to. This would indicate that customers change their behavior positively (in terms of our evaluation function) to the recommender, which is in line with our expectations. One method to examine the generalizability of the offline performance to online performance would be to model reactions of customers to a recommender, such that they 'react' to the recommender that is tested instead of the old one. This method would essentially model the online case. The problem is that this model will be based on the history of reactions, and is therefore still biased towards the old recommender. In existing literature (e.g. Li et al. (2011)) multiple parameter settings are used to develop the reaction model, making it possible to mitigate some of the bias. Since we only have one parameter setting that customers actually reacted to, it becomes extremely difficult to create a model. Therefore, this research did not model the reactions of customers but just used the existing reactions on one recommender.

To decrease the effect of the test and training set being different in off- and online case, as mentioned we choose to only use customers for the test set that have real reactions within 10 days of the chosen time-point. Choosing a smaller time frame decreases the amount of data, so we can alter this time frame and use the variance of the cross validation to determine whether the size of the data set does not harm the stability of the results.

Next to the problem of generalizing the offline results to the online case, it is important that the results in the offline case are stable, and are not subject to randomness induced by for example choosing a random starting point for the optimization algorithm.

To find out the degree to which the performance of a recommender is subject to the chosen starting point, we run the optimization algorithm multiple times, each time with a different starting point. We can use the variance in these results to determine how stable the solutions are.

We can find (an approximation of) the optimal solution by using a brute force method in the offline case. We can compare this optimal solution with the solution our algorithm found to assess the performance when the size of the data changes. For all results in this research, we chose to only optimize over 3 parameters: the amount of historical data taken into account to determine *sessionscore* (3#), the amount of historical data taken into account to determine user-item recommendations (4#) and the purchase versus click weight (#6) (see Table 1). The other parameters are kept at the current settings.

6 Results

To put the Bayesian optimization results in perspective, we first ran a brute force method that tried many parameter settings to find the optimal solution. The brute force method tried parameter 3# from 10 to 210 with steps of 50, parameter 4# from 10 to 190 with steps of 30, and parameter 6# with steps from 1 to 20 with steps of 3. It takes around 11 minutes to evaluate one recommender, so therefore we had to choose big step sizes. We choose

$$\boldsymbol{\beta} = [\beta_A, \beta_B, \beta_C, \beta_D, \beta_E, \beta_F, \beta_G, \beta_H, \beta_I] = [10, 7, 0, 5, 3, 0, 0, 0, 0]$$

$$\boldsymbol{\phi} = [\phi_A, \phi_B, \phi_C, \phi_D, \phi_E, \phi_F, \phi_G, \phi_H, \phi_I] = [1, 1, 1, 1, 1, 1, 1, 1, 1]$$

such that only recommendations that are bought or clicked on, or recommendations that are in the same category as bought or clicked products contribute to the score. $\boldsymbol{\beta}$ and $\boldsymbol{\phi}$ will remain these settings for the rest of the testing.

The results show that the best solution for the brute force method has parameters $\boldsymbol{x} = [Par3, Par4, Par6] = [110, 10, 3]$, with a score of 14.35. Remember from Section 5.2 that the higher the score the better. The results also show that parameter sets that satisfy $Par3 = 10$ significantly outperform the other parameters settings, except for when $Par6 = 0$. Furthermore, it seems that the lower $Par6$ the better the score, except for when it becomes 0. These two instances seem the only instances where there might be an interaction effect. We do not expect any other significant interaction effects by inspecting the brute force results. See the appendix for the top ten results.

From the results it also appears that a view weight of more than 9 the recommender score remains exactly the same if all other parameters are the same, which indicates that the recommendations do not change if you increase the view weight to more than 9. Therefore, we choose to decrease the boundary for this parameter to 9 for the optimization algorithms.

The following sections will describe the results from the different optimization methods used in terms of optimal solution, convergence speed and the variability of the performance due to a different starting point. The graphs shown

show the results of the different algorithms, all in the same manner. Every line represents a run with a different starting point. The dotted lines represent the optimal solution found by the run in the corresponding color. One can see a summary of all results in Table 4, and we will elaborate in the following sections.

Algorithm	Best sol.	Worst sol.	Mean best sol.	SD best sol.	Mean all sol.	SD all sol.
TPE + uniform	14.38	8.07	11.74	1.36	9.95	0.84
TPE + normal	18.57	7.86	17.59	0.59	12.39	2.85
GP + Matern	14.36	7.21	12.82	1.31	10.33	1.31
GP + sq. exp.	15.92	7.29	14.10	1.79	10.26	1.34
GP + Matern + DEI	13.34	6.53	11.79	0.80	9.96	1.02

Table 4: A summary of results on all algorithms tested

The best solution gives the best solution found in all iterations, and the worst presents the lowest objective value found in all iterations. The 'mean best sol.' displays the mean of the best solutions found per run. The 'SD best sol.' displays the standard deviation of the best solutions found per run. The mean and SD of all solutions provide these descriptives for all solution values found for all iterations and runs.

6.1 TPE and uniform distribution prior

The results of the TPE algorithm with using a uniform prior are shown in Figure 5. The results show that after 20 iterations the algorithm does not seem to converge. For example the orange run has a low objective value in the 20th iteration, which means that it is still exploring more than exploiting. The best solution value found seems to be a 'lucky' guess, and is clearly an outlier in the results.

The best solution found using a uniform prior is 14.38, given by parameters $[160, 152, 8.04]$ which gives almost exactly the brute force optimal objective value. The performance is relatively unstable, with the standard deviation of the optimal solutions being 1.36. The solutions found also seem to be dependent on the starting point. One of the runs even had its starting point already yielding the optimal solution.

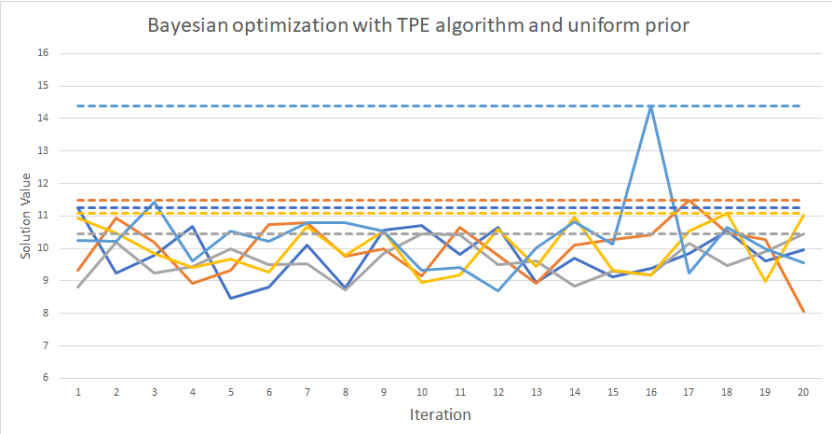

Figure 5: Results of the Bayesian optimization with the TPE algorithm and a uniform prior. Every line represents a run with a different starting point, and the dotted lines represent the optimal solution of the corresponding color.

Since it is costly to test bad performing recommenders, it is interesting to look further than only the best solution per run. We are also interested in finding out what the risk of the algorithm is. The average solution found was 9.95, which is a relatively bad solution value. The standard deviation of all solutions is 0.84, which is relatively low. Given that the lowest solution found is 8.07, this means that even though the solutions found on average might be bad, the risk of testing an extremely bad recommender seems low.

6.2 TPE and log-normal distribution prior

The results for the TPE algorithm with using a normal distribution prior are shown in Figure 6. The results show that, similar to the TPE results with a uniform prior, the algorithm seems to not converge within the 20 iterations. Many of the runs, even in the last iterations, show a low objective value, which means that the algorithm is still exploring at the last iteration.

All runs seem to perform very well, all performing better than the brute force method (!). The highest solution found was 18.57 with parameters $[5, 1, 1]$, which is a 29% improvement from the brute force method. Moreover, the high performance seems to be relatively stable, with a standard deviation of 0.59 of the optimal solutions. Moreover, starting points that are close to each other (orange line and dark blue line), do not necessarily have a similar performance. Where the orange run performs extremely well, the dark blue run performance as second worst from all runs, even though they start very similar.

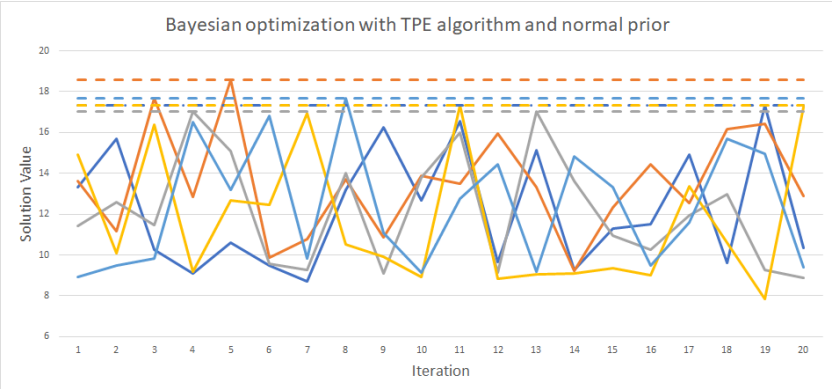

Figure 6: Results of the Bayesian optimization with the TPE algorithm and a normal prior. Every line represents a run with a different starting point, and the dotted lines represent the optimal solution of the corresponding color.

The mean solution of the algorithm is 12.39, which is relatively high. The standard deviation of the runs is 2.85, which is also relatively high. We could say that the TPE with normal prior performs well, but is also moderately risky since it deviates more in its solutions found. The worst solution found is with 7.86 decent, so the risk of the high standard deviation might be dissolved by the high average mean.

6.3 Gaussian process prior and Matern kernel

The results for the GP prior with a Matern kernel as prior for the covariance function are shown in Figure 7. The results show that regarding the optimal solutions, the optimization algorithm performs in between the TPE with uniform prior and TPE with normal prior. Three of the five runs find optimal solutions higher than 13, and the best optimal solution found has a value of 14.36 corresponding to solution $[73, 10, 0.75]$.

Similar to the other algorithms, the algorithm does not seem to converge. The yellow run illustrates the functionality of the algorithm. The run picks a risky point with a lot of uncertainty ('exploration') and then finds a relatively low objective value (the four downward peaks), it uses this information ('exploitation') and finds slowly finds a better solution with the newly gathered information. The GP algorithm seems to explore and exploit in a more extreme manner (ranging in solutions from around 7.2 to 14.4) than the TPE algorithm, making it more risky. Moreover the performance seems to be relatively instable, with a standard deviation of the optimal solution of 1.31. Therefore it seems that the starting point has an effect on the performance of the algorithm.

With respect to the risk of the algorithm, the mean of all solutions found is 10.33 and the standard deviation of the solutions is 1.31. The mean solution is in between the previous two algorithms, and the variance is moderately high. Even though the variance might not be extremely high, the algorithm does find

lower solutions than the other algorithms (going as down as 7.21), making it rather risky.

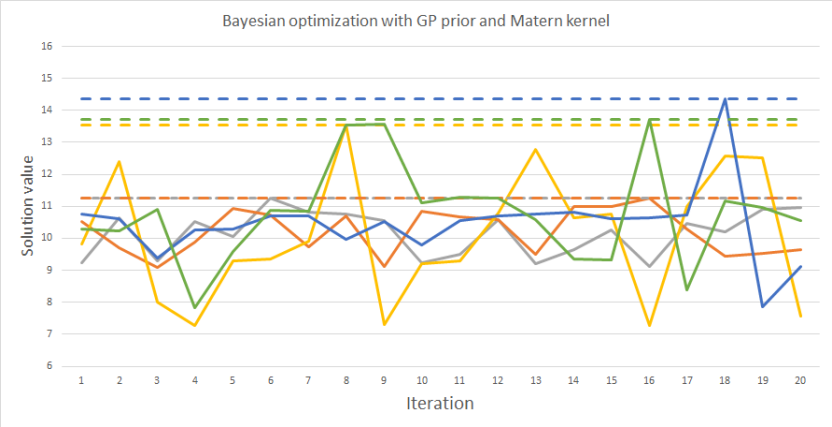

Figure 7: Results of the Bayesian optimization with GP prior and Matern kernel prior. Every line represents a run with a different starting point, and the dotted lines represent the optimal solution of the corresponding color.

6.4 Gaussian process prior with Matern kernel and DEI acquisition function

Figure 8 shows the results of the GP prior with Matern kernel with the Dynamic Expected Improvement function as proposed by this research. It is natural to compare this adapted version of the expected improvement function with the GP prior and Matern kernel with normal EI. Table 4 and the figure show that the adapted version performs worse in almost all aspects. The best solution found is the lowest so far (13.34), and the lowest objective function value found of all parameter settings tried is also lower than all the other algorithms (6.53), meaning that the algorithm relatively risky. Section 7 goes into why the algorithm seems to be performing so bad, and describes that there still might be potential in the proposed method.

Unexpectedly, the algorithm does seem to be more consistent, since it has a relatively low standard deviation of the best solutions found 0.80 and all solutions found 1.02. It also does not seem to converge, but as expected most best solutions are found at the end of the algorithm. Moreover, Figure 8 clearly shows that at the start of the algorithm there is more risk and the solutions go from very high to very low. At the end, most solutions seem to be more or less constant, and the algorithm is less risky.

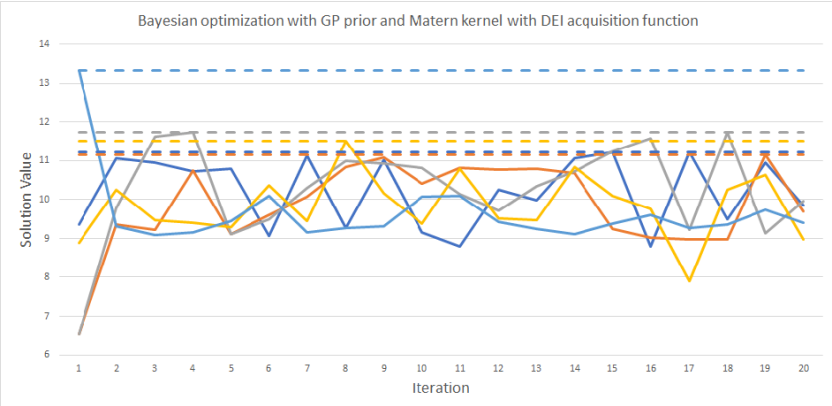

Figure 8: Results of the Bayesian optimization with GP prior and Matern kernel prior, with the dynamic expected improvement function. Every line represents a run with a different starting point, and the dotted lines represent the optimal solution of the corresponding color.

6.5 Gaussian process prior with squared exponential kernel

Figure 9 shows the results of the Gaussian prior with a squared exponential kernel. The graph shows that the GP prior with exponential kernel performs better with respect to the found optimal solutions than with a Matern kernel. All optimal solution but one are higher than 15, and the highest found is 15.92 with a solution that corresponds to $x = [68.51, 183.76, 1.56]$.

Given the other results, it was not unexpected that the algorithm also does not seem to converge. All runs do not necessarily give better results at the end of the algorithm, and seem to be exploring more than exploiting. Moreover, most optimal solutions are found between iteration 3 and 10. The optimal solutions found in almost all runs seem outliers, and it could be that all are 'lucky hits' in the sense that by trying to explore the algorithm stumbled on a good solution. This is all speculation though, and one could say the algorithm performs relatively in-stable with a high mean of optimal solutions of 14.10 but also a high standard deviation of 1.79.

To assess the complete risk picture, we find the mean and standard deviation of all solutions found. Our mean solution found is 10.26 and the standard deviation is 1.34. This seems to be very similar to the Matern kernel. Using a squared exponential kernel in this research gave better optimal solutions than the Matern kernel, while having the same risk situation.

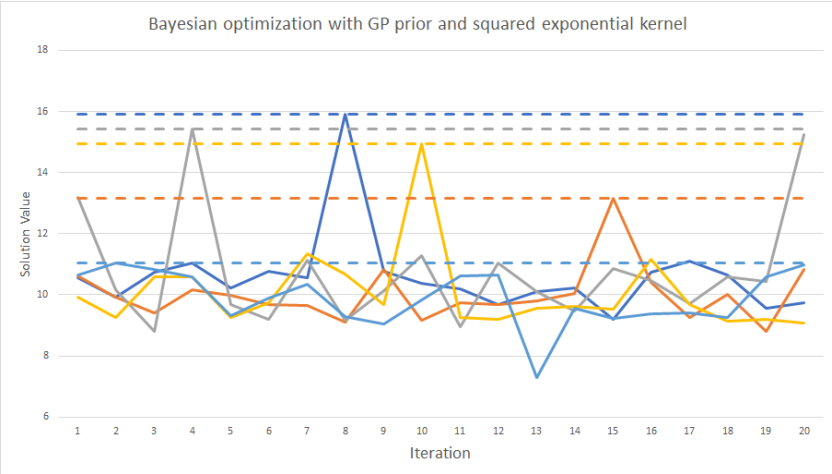

Figure 9: Results of the Bayesian optimization with GP prior and squared exponential kernel prior. Every line represents a run with a different starting point, and the dotted lines represent the optimal solution of the corresponding color.

7 Discussion

If one would just look at the graphs in the results section, he/she will conclude that the TPE algorithm with a normal prior is far superior over the other algorithms. However, this would be a too simple reasoning. The TPE algorithm with a normal prior indeed outperforms the other algorithms, but this might be due to the fact that it was given a stronger and more appropriate prior, that worked just in this particular case.

From the results, we can conclude that changing or including a prior greatly influences the performance of the algorithm. If the prior is an accurate representation of the 'real' objective function, it increases the performance greatly, and decreases the risk. In the case of a strong (like log normal), but inappropriate prior it might be that the algorithm is biased towards the prior in the extent that it will only find bad solutions. In practise we would suggest to only use a strong prior if one is certain that it is accurate to some extent.

One reason why the TPE algorithm seems to work better than the Gaussian Process approach is that the TPE algorithm by construction puts more weight on exploitation than exploration. Since we only have a low number of iterations, spending more iterations on exploitation instead of exploration might make sense, since we have less iterations to make use of the gathered information. This is especially the case when the prior reflects the 'real' objective function closely, since we have less risk of exploiting the function incorrectly.

When analyzing the brute force results, it seems that there are little interaction effects between the parameters: the marginal effect of changing one

parameter seems constant when we change the other parameters. This also contributes to the fact that the TPE algorithm performs well. The TPE method does not take into account interaction effects, and assumes a tree-like structure which seems to exactly correspond with the brute force results. The TPE method might not work so well in case there are interaction effects, but more research is needed to confirm this.

Even though the ultimate goal is to find a global optimum, with such a low number of iterations the customer is already happy with a high local optimum. All algorithms seem to not converge since they choose new points based on the expected improvement. Apparently 20 iterations do not generate enough certainty about the objective function so that the algorithms goes more into exploiting.

To solve this issue, we proposed to use the DEI acquisition function instead of the regular expected improvement. The results show that using this acquisition function does not generate better results. However, the results do show that there is potential. The main part where the algorithm performs very bad is in the first couple iterations. A solution of 6.53 was found twice in the first iteration. This is due to the fact that we put all weights on exploration instead of exploitation. Interestingly, the two solutions of 6.53 are identical, and both correspond to a parameter setting at the boundary of our solution space. After the first three initialization runs, it is not surprising that the point with the highest $\sigma(\boldsymbol{x})\phi(Z)$ is at the boundary of our solution space.

In the last iterations, most of the weight is on exploitation instead of exploration. One would expect that we would only find solutions higher than were already found in previous iterations. This is not the case, which might be due to the fact that $\mu(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ do not perfectly represent the 'real' objective function. According to $\mu(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ a point $\boldsymbol{x_k}$ might be a strong improvement over the previously best found solution, but when we evaluate the 'real' objective function at $\boldsymbol{x_k}$ we might find that the solution was not so good after all. It therefore becomes clear that using this acquisition function we did not explore enough to be able to exploit in the end. The brute force results shows that the application we tested the algorithms on is difficult in the sense that the objective function is not a smooth function. Since it is not smooth, the difference between our model and the 'real' objective function can differ significantly. If one then takes more risk by focusing on exploitation, we end up getting worse results. There might be potential for the DEI acquisitions with smooth objective functions, but this has to be researched more thoroughly.

It is important to look at the impact of decreasing the number of customers evaluated. For this research, we chose 1000 random customers. However, the data is sparse in the sense that most customers only click on one product without buying anything or putting anything in the cart. Therefore, most customers have a score of 0, while some customers buy a lot and generate a score of sometimes more than 1000. Therefore, it might be that a relatively small number of customers have a big impact on the score, since all scores are equally weighted. This might not be desirable since we want to recommend decent products for all customers instead of recommending extremely good products for just a small

number of customers. It could be an improvement to adjust the score of customers that buy or click above a certain threshold, to make sure these 'outliers' do not impact the score too much. We could also include a term in the score that allocates points to the number of customers that have a score that is not equal to 0, to create an incentive of recommending suitable products to as many distinct customers as possible.

To test whether 1000 customers are enough for testing, we also calculated the scores for $10,000$ customers and compared it with the score of 1000. Possibly due to the outliers mentioned in the previous paragraph, the score was very different. Since it is hard to determine whether the difference in score is due to the outliers and therefore definition of the score, or due to the 1000 customers not being a large enough sample to properly represent the complete sample, we recommend using more than $10,000$ customers to make sure that the results are representative for the full customer base. All algorithms in the results section are tested on the exact same set of customers, and even though the scores are different with different data sizes, the relative score of the algorithms remains the same, so the results are still likely to hold, although this is a point for further research.

Part of the research goal is that it should be possible for the user to define the objective function. The user can do this by changing $\boldsymbol{\beta}$ or $\boldsymbol{\phi}$. We have not explicitly tested whether the results change when the objective function is changed. However, since the objective function's uncertainty only comes from the separate elements (the capital letters) and not from $\boldsymbol{\beta}$ or $\boldsymbol{\phi}$, we can always calculate results with other $\boldsymbol{\beta}'s$ and $\boldsymbol{\phi}'s$ back to the settings used in this research, therefore making our results generalizable to all other $\boldsymbol{\beta}'s$ and $\boldsymbol{\phi}'s$.

Finally, it must be mentioned that the offline results are not necessarily representative for the online case, due to the reasons mentioned in Section 5.4. Interestingly enough, the optimal solutions the optimization algorithms found are not similar to the settings that were used to generate the data. This fact gives an indication that the recommender does not significantly impact the customer's behavior, and customers do not necessarily buy or click on products because they are inspired by recommendations. From a generalization perspective of this research this is positive, since it means that the customers' behavior would not change too much if he/she had viewed other recommendations. However, it might be that other (and therefore possibly better) recommenders might impact the behavior of customers much more, thereby distorting the generalizability of our results.

8 Conclusion

This research aimed to find an optimization algorithm that can optimize the hyperparameters of a recommendation system with the least amount of iterations and with little risk in the sense that little recommenders are testing that perform bad. It was found that the performance of the optimization algorithms highly depend on the chosen prior. If the user is certain that a prior accurately reflects

reality, the Bayesian optimization with a TPE algorithm and a log normal prior outperforms all the other tested algorithms. However, choosing a strong prior (like the log normal) is risky since a less accurate prior is expected to perform much worse.

The research also found that, in the case of a non-informative prior, the Gaussian algorithm seems to perform better than the TPE algorithm. Even though the TPE algorithm found one better solution, the Gaussian approach more consistently finds more solutions. It has to be mentioned that the Gaussian method is also more risky, so the user can choose between risk but a higher probability of finding a reasonably good solution or less risk but a smaller probability of finding a good solution.

Finally, we found that adapting the acquisition function to be more dynamic in this case performs worse than a regular acquisition function, mainly due not enough exploration to be able to exploit in the end. There might be potential in the DEI, but mainly for objective functions that are more simple in nature.

References

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554.

Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

Gunawardana, A. and Shani, G. (2009). A survey of accuracy evaluation metrics of recommendation tasks. *Journal of Machine Learning Research*, 10(Dec):2935–2962.

Gunawardana, A. and Shani, G. (2011). Evaluating recommendation systems. In *Recommender systems handbook*, pages 257–297. Springer.

Herlocker, J. L., Konstan, J. A., Terveen, L. G., and Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer.

Jones, D. R., Perttunen, C. D., and Stuckman, B. E. (1993). Lipschitzian optimization without the lipschitz constant. *Journal of optimization Theory and Applications*, 79(1):157–181.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492.

Li, L., Chu, W., Langford, J., and Wang, X. (2011). Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 297–306. ACM.

Martinez-Cantin, R., de Freitas, N., Brochu, E., Castellanos, J., and Doucet, A. (2009). A bayesian exploration-exploitation approach for optimal online sensing and planning with a visually guided mobile robot. *Autonomous Robots*, 27(2):93–103.

Matérn, B. (2013). *Spatial variation*, volume 36. Springer Science & Business Media.

Močkus, J. (1975). On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*, pages 400–404. Springer.

Rasmussen, C. E. (2003). Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer.

Resnick, P. and Varian, H. R. (1997). Recommender systems. *Communications of the ACM*, 40(3):56–59.

Ricci, F., Rokach, L., and Shapira, B. (2011). Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. (2016). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., and Adams, R. (2015). Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180.

A Measuring recommender performance

We define $D(\phi_D)$, $E(\phi_E)$, $F(\phi_F)$, $G(\phi_G)$, $H(\phi_H)$ and $I(\phi_I)$ as:

$$D(\phi_D) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|C_{i,t,\tau_e,r}|} \left(I[C_{i,t,\tau_e,r} \in R_{i,t}] \pi^{\phi_D}_{C_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{C_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \tag{16}$$

Where $C_{i,t,\tau_e,r}$ is product r that customer i bought in between time periods t and τ_e. $E(\phi_E)$ is very similar to $D(\phi_D)$, but now comparing categories instead of the product themselves.

$$E(\phi_E) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|C^c_{i,t,\tau_e,r}|} \left(I[C^c_{i,t,\tau_e,r} \in R^c_{i,t}] \pi^{\phi_E}_{C_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{C_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \tag{17}$$

And $F(\phi_F)$ for products clicked in other categories than the recommended categories.

$$F(\phi_F) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|C^c_{i,t,\tau_e,r}|} \left(I[C^c_{i,t,\tau_e,r} \notin R^c_{i,t}] \pi^{\phi_F}_{C_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{C_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \tag{18}$$

And we have the same concept for products added to cart, with the addition that we remove products from the set that are removed from the cart by the customer. So $IC_{i,t,\tau_e} = AC_{i,t,\tau_e} \notin RC_{i,t,\tau_e}$ where AC_{i,t,τ_e} is the multiset of products added to cart by customer i between time t and τ_e, and RC_{i,t,τ_e} is the multiset of products removed from the cart by customer i between time t and τ_t. We can then define:

$$G(\phi_G) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|IC_{i,t,\tau_e,r}|} \left(I[IC_{i,t,\tau_e,r} \in R_{i,t}] \pi^{\phi_G}_{IC_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{IC_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \tag{19}$$

Where $IC_{i,t,\tau_e,r}$ is product r that customer i put in cart and did not remove in between time periods t and τ_e. $H(\phi_H)$ is very similar to $G(\phi_G)$, but now comparing categories instead of the product themselves.

$$H(\phi_H) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|IC^c_{i,t,\tau_e,r}|} \left(I[IC^c_{i,t,\tau_e,r} \in R^c_{i,t}] \pi^{\phi_H}_{IC_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{IC_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \tag{20}$$

And $I(\phi_I)$ for products added to cart and not removed for other categories than the recommended categories.

$$I(\phi_I) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{r=1}^{|IC^c_{i,t,\tau_e,r}|} \left(I[IC^c_{i,t,\tau_e,r} \notin R^c_{i,t}] \pi^{\phi_I}_{IC_{i,t,\tau_e,r}} \frac{1}{\sqrt{H_{IC_{i,t,\tau_e,r}} - H_{R_{i,t}}}} \right) \tag{21}$$

B Brute force results

Daysitemitem	DaysUserEngagements	ViewWeight	Score
110	10	3	14.35
110	10	9	13.61
110	10	12	13.61
110	10	15	13.61
110	10	18	13.61
210	10	6	13.14
210	10	9	13.14
210	10	12	13.14
210	10	15	13.14
10	10	3	13.04

Table 5: Top 10 results brute force method