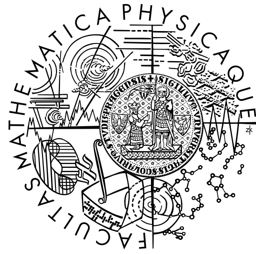

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Bc. Matěj Ščerba

Explaining Recommender Systems in Content-rich Domains

Department of Software Engineering

Supervisor of the master thesis: Mgr. Ladislav Peška, Ph.D.
Study programme: Computer Science - Artificial Intelligence
Study branch: IUIP

Prague 2025

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

This thesis is dedicated to everyone who has supported me throughout my studies, that are to be concluded by this thesis.
To my family, for their never ending support and unconditional love.
To my friends, for many great memories made along the way.
To my girlfriend, for her strength-giving love and motivation.
To my supervisor, for guiding me through the complex, and sometimes confusing, world of recommender systems.
To my colleagues, for sharing valuable feedback.
Thank you all for being a part of this chapter of my life and for helping me move to another.

Title: Explaining Recommender Systems in Content-rich Domains

Author: Bc. Matěj Ščerba

Department: Department of Software Engineering

Supervisor: Mgr. Ladislav Peška, Ph.D., Department of Software Engineering

Abstract: Services on the internet need to handle increasing volumes of data and so do the users. Information retrieval systems prevent overloading of users with extensive amount of information, the class of these systems used in ecommerce is called recommender systems. They select a subset of data to be presented to users. This selection is based on multiple criteria, which can be generalized as the estimated preferences of the users. Recommender systems are being used in all ecommerce domains, but are their capabilities fully exploited? This thesis analyzes the current usage of recommender systems in ecommerce and identifies problems in one of the domains, content-rich products. This thesis suggests possible ways to improve exploitation of the recommender systems in these domains and provides sample implementation of an interactive system rich with product and attribute explanation in the form of a web application. User study was conducted to evaluate the proposed solution, hopefully leading to better cooperation of users and recommender systems.

Keywords: explainability recommender systems content-rich domains

Contents

Introduction **3**

1 Problem **5**
1.1 Recommender systems 5
1.2 Customer behavior 6
1.3 Limitations of the current ecommerce systems 8
1.3.1 Finding relevant candidates 9
1.3.2 Selecting the best candidate 10
1.3.3 Utilizing recommender systems 11
1.3.4 Chatbots and virtual assistants 11
1.3.5 Summary 12

2 Application design **14**
2.1 Attributes 16
2.2 Products 16
2.2.1 Product groups 17
2.2.2 Filtered products 20

3 Data **22**
3.1 Requirements 22
3.2 Existing recommendation datasets 22
3.3 Scraping 23
3.3.1 Processing 23
3.4 Dataset 24
3.4.1 Products 24
3.4.2 Attributes 24
3.4.3 Images 26
3.4.4 Average ratings 26

4 AI models **27**
4.1 Recommender system 27
4.1.1 Diversification 28
4.2 Explanations generator 30
4.2.1 Unseen products versus all candidates 30
4.2.2 Candidate versus other candidates 32
4.3 Unseen statistics generator 33
4.4 Stopping criteria generator 33
4.4.1 Diversification 36

5 Implementation **38**
5.1 Repository 38
5.2 Server 38
5.2.1 Tech stack 39
5.2.2 Architecture 39
5.2.3 Configuration 40
5.3 GUI 40

5.3.1 Tech stack 40
5.3.2 Architecture 40
5.3.3 Configuration 41
5.4 Code maintenance and development 41

6 User study **42**
6.1 Goal 42
6.2 Setup 43
6.3 Results 45
6.3.1 Hypothesis 1 46
6.3.2 Hypothesis 2 47
6.3.3 Hypothesis 3 47
6.3.4 Hypothesis 4 48
6.3.5 Hypothesis 5 49
6.3.6 Comparison of UI variants 50
6.3.7 Open questions 51

7 Future work **53**
7.1 User interface 53
7.2 AI models 53
7.3 User study 53

Conclusion **54**

Bibliography **55**

Introduction

Internet helps us with more and more daily tasks, which means that the amount of information available is gradually increasing. The current amount of information is too large for people to process, selecting which movie to watch from more than 18,000 titles[1] available on Netflix is impossible without any filtering. Ecommerce, news, social media and many more domains needed to come up with a solution to this issue to succeed.

Consumer websites offering large amount of data usually present organized content. News, movies and products are split into categories to lower the number of relevant items. The number of items in a selected category can still be too large, Netflix offers over 1,000 comedy movies[2].

One way of narrowing down the amount of data for customers is by allowing them to apply filters. Users filter the products by restricting the values of attributes that the users think are relevant for them. Utilizing this feature effectively requires thorough understanding of the domain, such as knowledge of possible options, new trends or effect of attribute values on the item's quality.

Filtering is more suitable for content-rich domains, where each item has multiple attributes that have effect on the item's overall performance.

Applying filters on content-poor domains (for example movie streaming services) is a challenging task. Filtering movies by their length or actors starring is possible, but does not filter out bad movies confidently.

Filtering has lately been automized in the form of recommender systems. Recommender systems use past information about the current user and rank the items according to his/her modelled preferences. The best-ranked items are recommended to the user. We can view these systems as filters on complex attributes that estimate the user's satisfaction with each item.

Recommender systems have different goals in different domains. Social networks and streaming services aim to keep the users interested in the presented content to maximize their time spent on the website. Goal of recommender systems deployed on eshops is to increase their revenue.

Increasing revenue is a complex goal, which is why recommender systems are used for multiple tasks in ecommerce.

Customers are oftentimes greeted with personalized homepage, which displays products they are expected to be interested in. Customers can get to know the product catalog better, apart from faster access to the category they came to browse.

When customer browse the catalog in search for the product they want to purchase, recommender systems are designed to help them. Products in a category can be ordered by the customer's expected preference. When a customer is looking at a certain product, alternatives to that product are usually displayed.

Before proceeding to checkout, users are presented with more products to add to their current order. This can prolong their stay and possibly convince the

[1] According to information available at https://www.hollywoodreporter.com/tv/tv-news/netflix-releases-viewing-data-for-18000-tv-shows-movies-1235745500/, accessed July 9, 2024.

[2] According to information availbale at https://www.whats-on-netflix.com/library/comedy-movies-on-netflix/, accessed July 9, 2024.

customers to purchase more products than they initially intended.

These are a few examples where recommender systems are used in ecommerce. They have one thing in common; they select a subset of products[3] and present them to the customer as candidates he should consider purchasing.

There are domains where selecting the product to purchase takes majority of the time customers spend on the eshop. Selecting the product can be characterized as high-cost decision in these domains. These products are typically expensive, often used by customers and are not purchased frequently. Cars, computers or holiday trips are a few examples. These domains generally contain a lot of attributes, which is why customers tend to invest more time into selecting the product that suits them the most.

Recommender systems work the same way in these domains as in streaming services, even though selecting a movie to watch in the evening is a completely different situation than selecting a car to purchase.

Recent development of recommender systems focused on customers' trust, which is why modern recommender systems explain why certain products have been recommended to the users. Current recommender systems are capable of explaining why a certain product is good or bad in a user-friendly way.

If the recommender systems and their explanations do not persuade a customer to buy a recommended product right away, he/she must select the product by browsing the catalog. Selection of the best product can be split into two phases. Customers have to find all relevant products and then select the best product from them. Explanations described above help with the second phase.

Customers that want to find the best possible product have to look at a large part of the product catalog, it is probable that they inspect a lot of irrelevant products before eliminating them.

This thesis aims to propose a way to decrease the time needed to find the set of relevant products by designing an interactive system capable of telling the user when all of the relevant products have already been inspected.

[3]Ordering products in a category is the same as selection of a subset of products, because majority of customers do not go through the whole category.

1. Problem

Artificial intelligence gained significant attention in the recent years. It is capable of solving vast range of tasks. People found many tasks for artificial intelligence in ecommerce, one of which is product recommendation. Very similar, if not the same, interface of the product recommendations is used throughout various ecommerce domains and this interface is not ideal for some of the domains. Different domains bring out different decision making process in the customers, but the recommender systems do not reflect these differences.

This chapter describes recommender systems in section 1.1. Section 1.2 describes the behavior of customers when purchasing different types of products. Recommender systems should be designed to help customers in all domains, section 1.3 identifies domains where recommender systems are not implemented ideally.

1.1 Recommender systems

Number of products on eshops is gradually growing, it is impossible to analyze the whole product catalog when finding a product to purchase, for example Amazon offers over 350 million products[1].

Majority of eshops allow their customers to apply filters on the available products to lower the number of products to be displayed at once. Customers need to have decent understanding of the domain to apply the required filters, because customers are typically allowed to filter products by values of their attribute. It can be difficult to select the proper value of an attribute when purchasing a product in a domain not familiar to the customer. This is where recommender systems come in handy and make it easier for customers.

Recommender systems typically inspect a large number of products and present a small subset to the customer. This subset is expected to be interesting for the customer, the criteria that the products should satisfy can be defined in multiple ways. All these definitions have a common goal, which is to present content that the customer will eventually consume (purchase a recommended product, watch a recommended movie or read a recommended news article).

Classical implementation of recommender systems assign a score to each product, the higher the score, the more the customer is expected to like the product. This is why we can consider recommender systems as filters, they select a subset of k products based on multiple criteria, main of which is the score. The selection is not that simple in real world, because recommender system output can be post-processed to meet different criteria than pure score, for example scoring products by their similarity would select k very similar products, diversity is useful in some contexts, as Castells et al. [2022] shows.

Computation of the score can be implemented in various ways, which is why there are multiple categories of recommender systems. It is possible to categorize recommender systems in various different ways, some of them are described by Ricci et al. [2022]. We will show two of the most common classes of one of the

[1] According to information available at `https://www.mobiloud.com/blog/amazon-statistics`, accessed on July 9, 2024.

classification, content-based and collaborative-filtering methods. These types of recommender systems differ in the data they are using.

Content-based recommender systems consider attributes of products when performing recommendations. These systems know what products the customer liked in the past and recommend products similar to these products.

Collaborative filtering methods generate recommendations based on similar customers. The idea is that if customers rate some products similarly, they are likely to rate other products similarly as well.

Each of these (and more) approaches has advantages and drawbacks, which is why modern recommender systems are usually implemented as *hybrid models*. These are complex systems containing multiple classes of recommender systems. The system can aggregate recommendations performed by each model based on the current situation or use only the most suitable algorithm.

Recommender systems are used in multiple places on the eshop and each instance serves a different purpose, because the customer encounters it in different phase of his/her visit.

The first instance of recommender system that the customer encounters is *homepage personalization*. Homepage is a good place to present new products to customers, recommender algorithm can be applied to select the most interesting products for the current customer. Generally speaking, the homepage should present products that the customer is likely to be interested in during his/her visit, it might contain frequently purchased products or complementary products for his/her past purchases. The goal of personalized homepage is to navigate customers faster to the category of products they are interested in or familiarize them with different products available at the eshop.

During the customer's search for the product to be purchased, *product list ordering* is implemented to show the interesting products of a category first. After opening detail page of a product, *alternatives to the given products* are presented to the customer to save his/her time of searching for those products and provide initial comparison of those options.

When the customer selects a product to purchase, he/she proceeds to cart where the recommender system suggests *products to extend the order with*. It typically finds products that other customers frequently buy together.

We argue that customers would benefit from a combination of *product list ordering* and *alternatives to the given products*, a system that would guide the users more when browsing the product catalog. And focus on the browsing and exploration of the relevant area. Modifying the recommender system alone might not be enough, the user interface of the eshop would probably have to change as well, as we will see later in this thesis.

1.2 Customer behavior

Each customer behaves differently when it comes to deciding which product to purchase. The behavior depends on the customer's character as well as on the current domain of products.

Schwartz [2004] discusses the influence of growing number of options when people are making decisions. Ecommerce is a nice example of this phenomenon, Amazon, as mentioned earlier, offers more than 350 million products. Having

more options might seem good for people; everybody can choose what suits them the most, but the reality is different.

People have a various demands that the product they want should fulfill. More products lead to better satisfaction of these demands, increased competition of manufacturers and generally better quality of products.

The increase of number of products changed the way customers are thinking about the purchase they made, especially if they are not happy with the product. Unsatisfied customers used to blame the manufacturer in the past, if they were not happy with a product. A limited number of options was available to the customers, so selecting the best option was not difficult. If the product was not good enough, the customer would blame the manufacturer: "I wish they made this car bigger".

This is not the case nowadays. If a customer purchases a car and is not satisfied with its size, it is probable that there exists a bigger car that the customer could have purchased. The customer ends up blaming himself/herself: "I wish I bought a bigger car".

During the selection of the product, customers can build higher expectations then they initially had or even higher than any product can satisfy. As they inspect one product after another, they might remember all the interesting features some of them have and imagine an ideal product possessing all of those features, although that is not usually possible. Customers are simply overwhelmed by the number of products available. Questions like *Would product P be better than the one I purchased, because it possesses feature F?* pop up in the customer's head after the purchase. These doubts are present even if the customer selects the best possible product for them [Schwartz et al., 2002].

We should note that there is no guarantee that an ideal product for a customer exists. Customers typically have multiple requirements for the product. Let us look at an example; a customer wants to buy a family car. It needs to be large enough to fit the whole family in comfortably, it needs to have powerful enough engine to carry all their stuff with ease, it needs to have reasonable fuel consumption and it needs to come with a reasonable price tag. It is not possible to select a car that satisfies all of these requirements the best.

Different customers are influenced differently by the issues described above because they behave differently when purchasing a product, Schwartz [2004] identified two types of people: Satisficers and Maximizers.

The first type of people select a product to purchase as soon as they encounter any matching their initial criteria. These are called *Satisficers*, they are happy as soon as they find a product that is *good enough.*

The second type of people is called *Maximizers.* As their name suggests, they search for the best possible product. They invest more time and energy into the decision. Maximizers are typically more stressed about their decision and they show a higher level of doubt after they make their decision.

In reality, people are not pure Maximizers or Satisficers, they are a combination of both. Moyano-Díaz and Mendoza-Llanos [2021] also shows that the same people behave differently in different domains. A person can behave as Maximizer when buying shoes and as Satisficer when selecting a restaurant where to eat.

Although 90% of people are Satisficers [Schwartz, 2004], there are domains where both Satisficers and Maximizers behave more like Maximizers. Moyano-

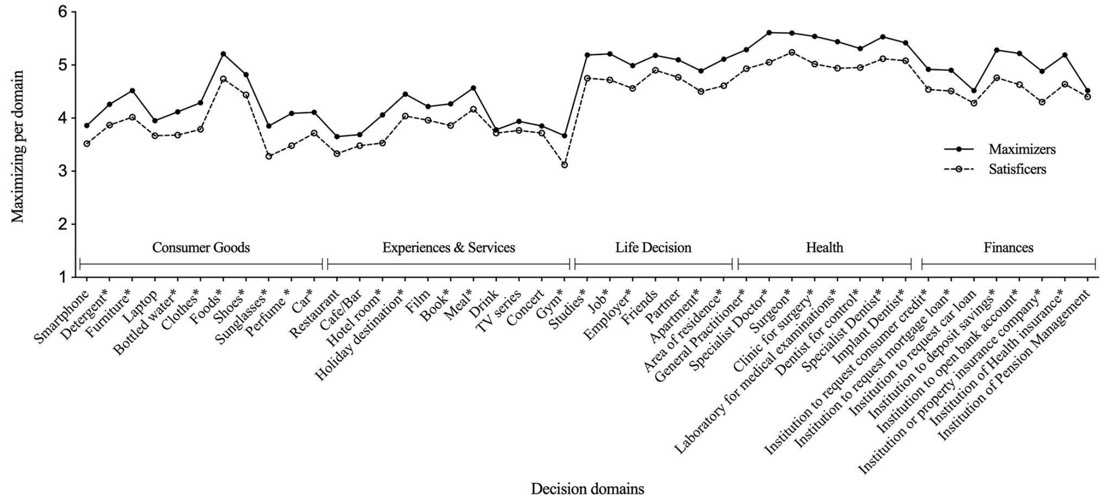

Figure 1.1: Maximizers' and satisficers' maximizing tendencies in various domains [Moyano-Díaz and Mendoza-Llanos, 2021].

Díaz and Mendoza-Llanos [2021] studied the extent of maximization that participants showed in various domains, which is shown in figure 1.1. The level of maximization is above half of the scale in majority of domains, even for Satisficers. This indicates, that people generally feel more like Maximizers.

Satisficers interact with eshop differently than Maximizers. Their typical visit of an eshop starts by looking for the category of products they wish to purchase, next they inspect a few products and as soon as they find one that is *good enough*, they purchase it.

Maximizers' process of purchasing a product is different. The step that differs is the inspection of products. They are trying to search for the best product possible. Maximizers inspect larger number of products, read reviews and compare products side by side. Let us assume how a Maximizer would buy a jacket in a physical store. He/she would walk through the store and pick up several jackets he/she likes before heading to fitting rooms. In the fitting rooms, he/she would select the jacket that suits him/her best. This process can be iterated several times. We can split this process into two phases: finding relevant candidates and selecting the best candidate.

1.3 Limitations of the current ecommerce systems

Current eshops are designed to suit Satisficers more, which is why Maximizers often reach out to third-party services and look for additional information elsewhere; this may include watching reviews of products or reading comparison articles.

We argue that eshops are capable of presenting all the information Maximizers need to make their decision and slight modification of the interface would make this process more comfortable.

Let us discuss the two phases mentioned at the end of section 1.2.

1.3.1 Finding relevant candidates

Browsing through the product catalog to find relevant candidates in current eshops is quite comfortable. Customers are allowed to view the category of products that is interesting to them, they can even apply filters to narrow the domain down. However, this process can be improved.

Applying filters

Applying filters, as mentioned earlier, can be tricky. Understanding of the product domain is required to set the filters properly, example of filter options in a complex domain is illustrated in figure 1.2. In addition to that, the filters are hard-limiting. Customer might want to filter products with value of an attribute that is higher than a specified lower bound, but other attributes can outweigh lower value of a product, especially in complex domains.

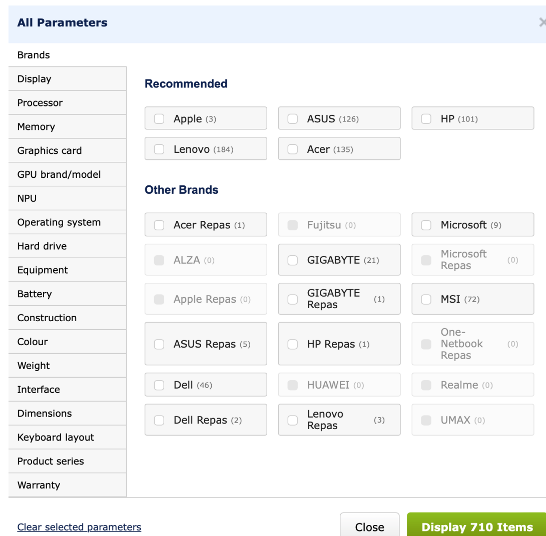

Figure 1.2: Filter options when purchasing a laptop on Alza.cz eshop.

Let us illustrate this problem with a concrete example. Imagine a person looking for a gaming laptop, the main requirement on the laptop is to be powerful enough. This is the reason why the customer applies filters on the number of processor cores, but this is not the only attribute, that affects the laptop's performance. For example the frequency of processor can outweigh slightly lower number of cores, or graphics card performance might be more important for the customer without him knowing it.

Keeping set of candidates

Current eshops offer limited methods of keeping lists of interesting products. This can be useful when browsing through a category and looking for an ideal product to purchase. Keeping track of possible candidates in current eshops is cumbersome; customers need to exploit cart or wishlist features, but these are not designed for this use case. Other possibility is opening detail pages of all considered products in individual tabs, but then it is not possible to display many products side by side.

In-person stores allow their customers to select arbitrary many products and inspect them side by side or try them on. This is an important process when searching for the best product offered by the store.

Ending the search

Selection of the candidate products performed by Maximizers is usually time-consuming. The reason for this is that the customer does not know when to finish looking for more candidates and focus on selection the best option from the filtered products. Maximizers experience fear of missing out, which is why they tend to explore greater part of product catalog than Satisficers. Even when they already selected all the relevant products, they fear that they are missing a better product they have not encountered yet.

The closest current eshops come to tackling this issue is by displaying number of products that have certain value of an attribute that can be filtered, this is shown in figure 1.2. This feature was primarily designed to help customers with applying filters, which can be challenging to define properly.

1.3.2 Selecting the best candidate

Picking the best option from a small set is not that challenging task, the reduced amount of data allows customers to inspect each product thoroughly or discuss the decision with an expert. Third party services allow customers to compare all attributes of a product, it often generates a summary containing an overall evaluation performed by an expert. Output of such a comparison of two graphics cards is illustrated in figure 1.3.

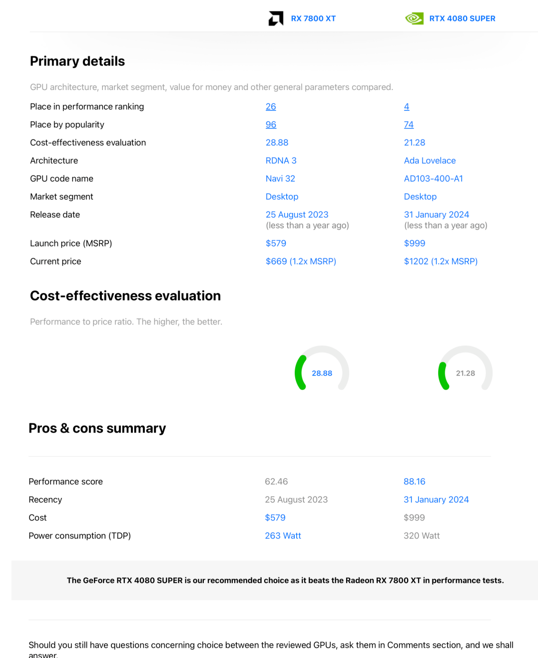

Figure 1.3: Comparison of two graphics cards performed on technical.city website including overall scores and price/value ratio (`https://technical.city/en/video/Radeon-RX-7800-XT-vs-GeForce-RTX-4080-SUPER`).

Some eshops offer simple version of this type of tools, but customers interested in this feature usually seek third party solutions, that are more capable.

1.3.3 Utilizing recommender systems

The role of shop assistants in brick-and-mortar stores is very similar to the role of recommender systems, they guide customers through the store and point out suitable products for them, they offer advice on which product to choose and they mention sales or accessories to increase the number of products sold. Yet there is one crucial difference between recommender systems and shop assistants; we trust shop assistants more with high-risk decisions.

Let us illustrate this with an example. Suppose you want to buy a second-hand car. This is going to be an expensive purchase, which you do not do very often. This means that you probably do not have thorough knowledge of the domain and the most recent development and trends. You use car very often and want to select as suitable car for you as possible.

We will consider two scenarios: in-person and online purchase.

If you want to purchase a car in a car dealership, you typically schedule an appointment with an employee. The meeting is lead by the employee asking you all kinds of questions regarding the intended use of the car. After he/she gathers enough information about you and your requirements, he/she can start offering cars for you. These recommendations are truly tailored, because you just revised your requirements and his/her expertise translated them into concrete product offers. The whole process is a dialogue that iteratively refines your requirements and the dealer's offers. If you have any questions regarding the purchase, he/she is (ideally) willing to provide accurate answers.

Online world is different. You visit a car dealership's website and are greeted with current sales, new products and/or recommended cars based on your past visits. Recommender systems are designed to help you similarly to dealers, but they lack the initial information dealer asks you in person. Recommender systems estimate preferences of customers based on the actions they perform on the eshop. When you are browsing a product catalog and are looking for a product, the most frequent actions you perform is open/close detail page of a product. Harvesting preference information solely from this type of actions is much more challenging for recommender system than if you tell the dealer that the car he recommended is a little too small.

Gathering explicit feedback data from the customer is very difficult for traditional recommender systems. Interactive elements would need to be presented to customers for example after analyzing a detail page of a product, which could be annoying. Recommender systems can not know whether customers want to discuss their preferences with them or browse the product catalog on their own. Shop assistants can estimate customers' mood from their behavior and body language.

1.3.4 Chatbots and virtual assistants

A growing number of eshops implement chatbots on their websites. Their role is to simulate a shopping assistant and help customers during purchasing process.

They provide more guidance than recommender systems, but the underlying AI's performance depends greatly on the complexity of the queries. Cherniak [2024] found out that chatbots are able to resolve 80% of routine tasks, but only 12% of queries in healthcare environment, because of their complexity. This is why chatbots are not reliable in complex product domains, yet.

Traditional recommender systems have a long-known problem with gaining trust of customers [O'Donovan and Smyth, 2005]. The models are typically implemented by the eshop and their goal is not transparent to the customers; they might be configured to help the customer as much as they can or they might be configured to maximize the eshop's revenue. Chatbots have the same problem, but there is one area of AI language models, that overcomes this drawback: *virtual assistants*.

Virtual assistants, especially the ones we have in our phones, are building trust of their users step by step. The tasks solvable by these models is quickly growing, few years ago, they could add a meeting to your calendar and now they are able to call a restaurant and make a reservation on your behalf.

The current trend shows that the capabilities of virtual assistants will continue to grow, while gradually gaining trust of their users. We believe it would be possible to ask the assistant to select a new laptop for you in the future.

The assistant could analyze the whole product domain without being overloaded by the amount of data and select the best laptop for you. Especially because the model knows a lot of information about you: what you do for living, how you use your devices, what is your budget and much more information that can help making a well-informed decision on your behalf.

However promising this sounds, it is not our goal to revolutionize the way people interact with their personal assistants and eshops, which is why we aim to help *people* when selecting a product in a complex domain.

1.3.5 Summary

Finding relevant candidates in high-risk domains is a time-consuming task, especially if customers want to find all relevant products offered by the eshop. Current approaches do not help customers with this task as much as human shop assistants do.

Recommender systems have the capabilities to help customers with the selection of relevant candidates, especially when it comes to suggesting alternatives to already selected products, but the interface and customer flow should be modified to suit this task better.

There are three main drawbacks of the current process of selection of relevant candidates.

Lack of explicit feedback data

Recommender systems can work with explicit and implicit feedback data from the customers. Explicit feedback can be seen as any rating given by the customer on individual product or a group of them. For example if a customer writes a review for a purchased product. The problem with explicit feedback data is its sparsity; there is not enough data to estimate preferences of customers, especially if they are new to the eshop and did not write any review. Implicit feedback data

contain any action a customer performs on an eshop, it includes clicks, adding products to cart, amount of time spent on a detail page of a product and much more. Preferences based on these data are estimates, they might not be accurate, but each eshop gathers lots of implicit feedback data, so sparsity is not an issue here. Combination of both these types of data leads to more accurate customer preferences estimates [Mandal and Maiti, 2018].

Lack of storage of candidate products

Current eshops do not possess the functionality to hold the set of candidate products when looking for the best product in the catalog. There is no analogy to taking several jackets to fitting rooms in the online world. When customer is looking for a product in a high-risk domain, he/she should be able to hold the set of products he/she is considering in a list directly in the eshop, not in his/her mind or multiple tabs in the browser.

Lack of cut-off alerts

Customers can waste a lot of time looking for relevant products in a large product catalog, especially if there are no more relevant products offered. Analogy to this functionality is a shop assistant telling you that there is no more jackets in stock to suit your needs, which saves your time of wandering through the aisles of store before finding out on your own.

2. Application design

Let us start with a general overview of the application before describing its individual features.

Our intention was to find a compromise between in-person shops and eshops, because both domains have their own benefits, this is why we needed to slightly modify the layout of traditional eshops. We decided to move eshops closer to in-person stores to simulate the following experience: Being guided through a store by a shopping assistant. We especially wanted the customers to be able to *hold* the products they consider buying (referred to as *candidates*), *being offered* alternatives to the candidates, with regards to products they *do not like* and *narrowing down* the domain to important attributes only.

Let us illustrate the usefulness of these features by providing concrete examples in an in-person clothing store.

A customer buying a jacket selects a few jackets before heading to a fitting room (he/she holds the candidates). The benefit is that when he/she considers a different jacket, he/she can immediately compare it side-by-side to any of the candidates.

He/she can ask the shopping assistant to see more of blue jackets, but not the dark ones, and it does not matter what material the jacket is made of. This situation illustrates the combination of the rest of the features (being offered alternatives, specify what products he/she does not like and narrowing down the domain).

Figure 2.1 shows the proposed layout of the application. Customers can select attributes they are interested in in the left menu. Their current candidates are at the top of the main section of the screen. Under them is the section explaining the rest of the product catalog, simulating a shopping assistant saying *we have one more jacket that you might like.* The rest of the page is filled with alternatives to the current candidates and the discarded products. We do not want to hide these because the customers might change their preferences or misclick.

We designed the application to help customers select the best possible product for them with emphasis on selection of relevant candidates, which is why our application does not implement standard features available in traditional eshops (such as adding a product to cart or opening its detail page). Our application is more of a *proof of concept* simulating only a specific part of the shopping process.

The intended use case of the application starts when a customer selects a category he/she wants to purchase a product in. The customer is presented with an unorganized layout of the application containing only the menu for selecting important attributes and a single list of products. We did not want to estimate customer preferences before he/she does any action in the current session. Only after a customer selects a candidate is he/she presented with an organized layout of the application, which is shown in figure 2.1.

Selecting important attributes narrows down the domain for the customer both on frontend (the customer sees values of important attributes only) and on backend (the underlying models ignore unimportant attributes).

As soon as a customer selects at least one candidate a recommender system can start recommending alternatives to the current customer. Detailed properties of

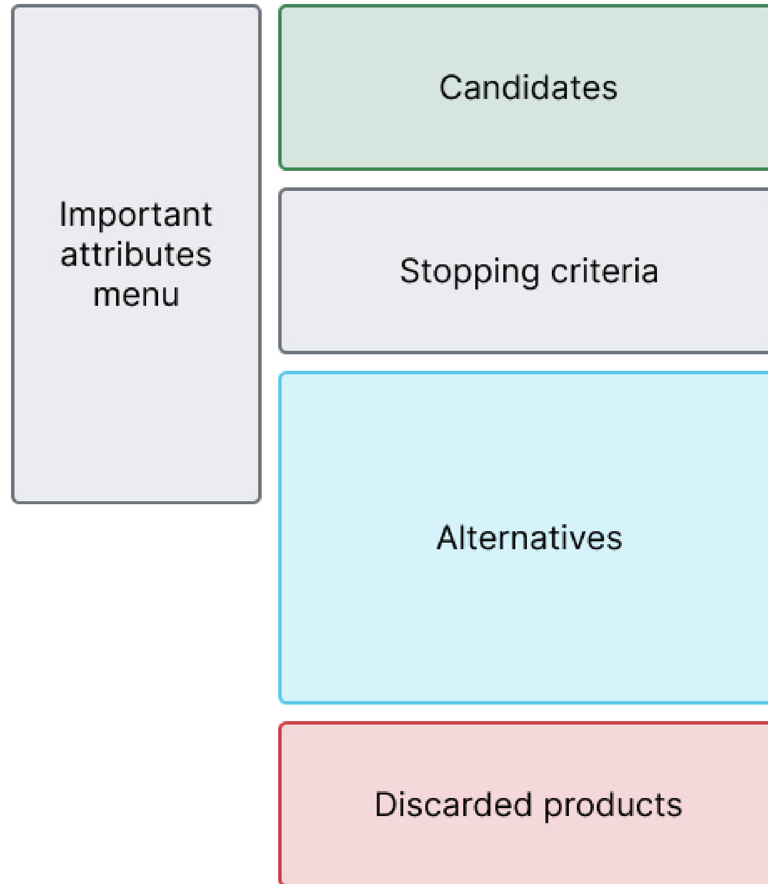

Figure 2.1: Proposed layout of the application, left column contains menu for selecting important attributes, main section contains the products of the category and stopping criteria.

the recommender system are described in section 4.1. The recommender system is designed to simulate a shopping assistant showing a customer more jackets he/she might like. The recommender system can also utilize the currently discarded products to account for negative feedback provided by the customer. The nature of the application allows the customer to provide explicit feedback data (whether he/she likes a given product or not), the recommender system does not need to estimate the customer's preferences from implicit actions such as opening a product detail or time spent looking at a product, it can directly utilize the explicit feedback, which is more accurate, as Mandal and Maiti [2018] shows.

All products are enhanced with explanations provided by explanations generator, which is described in section 4.2. It explains individual attributes and also products as a whole, its goal is to provide more information about the domain to the customer. Some customers are not familiar with the domain, explanations can help them when deciding which product is better.

We wanted to provide information about the unseen products to the customer that tells him/her how many *relevant* products are still in the product catalog. Our intention was to be able to tell the customer when to stop looking for more products that might be interesting to him/her and focus on selecting the best candidate.

The rest of this chapter describes individual sections of our application in

more detail.

2.1 Attributes

We begin our description with attributes, because they narrow down the domain and enable customers to compare different products, especially in content-rich domains.

Attributes in this context define different properties of all products, such as number of processor cores of a laptop or display size of a mobile phone. More technical information about the attributes is provided in chapter 3.

Narrowing down the domain was an important feature to us, because we did not want to overload the customer with irrelevant information. Content-rich domains can have high tens of attributes, narrowing the domain to approximately five attributes makes it easier for a customer to compare two products and he/she does not have to consider other attributes, that are not important for him/her[1].

There is a possibility to generate the set of important attributes automatically from the customers' data, because a lot of researchers, including Karimzadeh et al. [2024], in ecommerce focus on assigning importance to individual attributes, the importance is considered in the systems for multiple reasons including personalized interface (only the most important attributes are shown to the customer), explanations (description of why a product was recommended by means of its most important attributes) and recommendations (which attribute values should affect the recommendation result when processing the products). Importance in this context can have multiple definitions such as how much is a customer's decision of which product to buy influenced by an attribute or how strongly should a recommender system model reward or penalize an attribute to provide the best recommendations.

We decided not to implement such a complicated feature and kept the categorization of attributes entirely in the hands of the customers, the reason for this simplification is that we wanted to focus on different features of our application. Each customer can mark each attribute as important or unimportant by using the important attributes menu in the left column of the page layout depicted in figure 2.1, which effectively creates personalized product domain both on frontend (what attributes the customer sees) and on backend (what attributes the whole application considers).

2.2 Products

We can now move on to products. In our understanding, products are defined by values of the important attributes only, even though each product probably has much more attributes. The product space the application and a customer works in is defined by the important attributes. Each product is represented by values of the important attributes, we can interpret the product space as a vector space of dimension $d = |I|$, where I is the set of important attributes. A product can

[1] We understand that the element of serendipity (unexpected, surprising situation) is important in recommender systems, but we decided that narrowing down the domain strictly is more suitable rather than generating content for surprising, unimportant attributes.

be represented as a vector in that space, but some attributes are not numerical, so some elements of the vector can be for example textual.

Each product has additional data assigned to it, such as its name, rating and image, which the application does not consider when processing the products. This data is intended only for the customer to improve his/her experience with the system. Figure 2.2 illustrates how a product is displayed in the application.

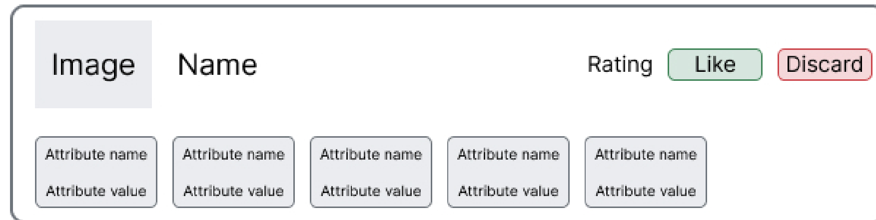

Figure 2.2: Proposed design of a product item, only important attributes are displayed in the list of attributes in the bottom section.

Interface of traditional eshops allows us to browse products of different attribute domains in a single list. Our proposed application can handle this as well, but we decided to simplify our situation by displaying only products of a single category in a single page. Products of one category are similar enough that we could assign the same attributes to all of them, even though values of several attributes of some products can be empty.

2.2.1 Product groups

The core feature of our application is the possibility to organize products into groups. Customer can mark any product as a candidate if he/she likes it or discard it if he/she does not like it. These actions generate explicit data that are processed by the application and the rest of the products is further organized for the customer based on his/her preferences estimated by the system. The part of the product catalog that was not yet inspected is presented to the customer in a non-overwhelming way. Only a small subset of alternatives to the current candidates is presented and the rest of the catalog is aggregated into easy to understand information displayed in the stopping criteria section shown in figure 2.1.

There are three main groups of products:

- candidates,
- discarded products and
- unseen products.

Let us consider product groups as sets, then the following holds:

$$C \cup U \cup D = P$$

C is the set of candidates, U is unseen products, D is the discarded products and P is the set of all products. This means that each product is included in some product group, in other words, the groups cover the whole product category.

Also, all the product groups are distinct, meaning no product is in multiple groups:

$$\forall G, G' \in \{C, U, D\} : G \cap G' = \emptyset$$

Our application does not allow customers to display the whole set of unseen products. We restricted the application this way to enforce customers to utilize the recommender system and its provided alternatives when exploring the product catalog. However, it is possible to display a subset of the unseen products. There are two types of these subsets:

- alternatives and
- filtered products.

All possible actions and transitions of products between different groups is depicted in figure 2.3. The figure shows also the two types of subsets of unseen products, because only these subsets are displayed to the customers.

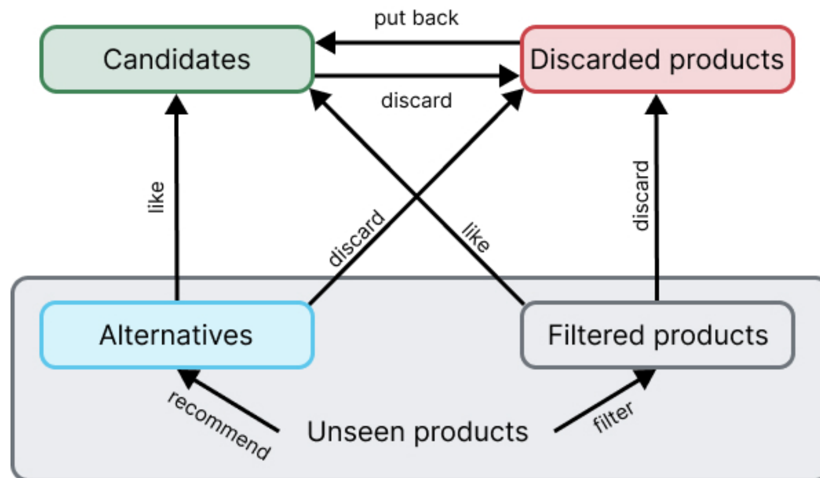

Figure 2.3: Product groups and transitions between them.

Candidates

The group of products that is in the topmost section of the page contains the candidate products. These are the products a customer has already seen and marked them as candidates (he/she liked them and considers purchasing them).

The purpose of this group of products is to create a storage for all products a customer has shortlisted. It is easier for customer to make a decision in this small group of products rather than in the whole product catalog.

We wanted to provide some form of support to customers when selecting the final product inside this set, which is why we generate explanations of individual attributes of all products. These explanations are designed to highlight differences of candidates to help the customer when choosing the best product for him/her. More information about these explanations is provided in section 4.2.

A customer can perform only one action with a candidate product; discard it if he/she decides to remove a certain product from candidates. This action moves the product to discarded products.

Discarded products

If a customer does not like a product, he/she moves it into a set of discarded products. This group of products serves a similar purpose as a *bin* on your computer, you delete the files you do not need anymore. It sometimes happen that the a customer changes his/her mind and he/she decides to mark a product as a candidate. We allow this action.

We do not provide any explanation of these products, because they are typically not interesting for customers.

Alternatives

Exploration of the product catalog is not performed by full-text search, filtering of attributes or ordering of a product list. It is guided by a recommender system that considers customer's previous actions (what products he/she added to candidates or discarded) and provides a small set of alternatives to the current candidates. The recommender system has to ensure that the exploration will be strong enough because there is no different way to explore the product catalog. More about the implemented recommender systems and their properties is described in section 4.1.

The alternatives are recommended to a customer, we have no idea whether he/she will like or dislike them, which is why we allow the customer to add each alternative to candidates or discard it. Each of these actions can potentially refine the recommender system to provide different recommendations, which should ideally guide the customer towards relevant products.

A customer is expected to potentially add an alternative to candidates, this is the reason why we generate explanations for the whole products and their individual attributes. These explanations are meant to help the customer when comparing an alternative to his/her candidates. Section 4.2 describes the explanations in greater detail.

No alternative was earlier interacted with, these products come from the group of unseen products.

Unseen products

One of the goals of our application is to tell customers when to stop looking for more products and focus on selecting the best candidate. Providing information about the rest of the product catalog appeared to be a natural approach to implementing this functionality. It is not possible to display all information about the rest of the product catalog, because there can be hundreds or even thousands of products a customer has not yet seen. We decided to aggregate and personalize the information for individual customers. These information is presented in a section called *stopping criteria*.

We implemented two different variants of generation of these information: *unseen statistics* and *stopping criteria*. We describe both of these approaches briefly in the rest of this section, more detailed information is provided in section 4.3 and section 4.4.

Unseen statistics represent our initial approach. We generate relevant sets of values of individual important attributes and display the number of unseen

products that have a relevant value of a certain important attribute, a customer can open a list containing all these (filtered) products. The proposed visualization of unseen statistics is provided in figure 2.4, there is one unseen statistics item for each important attribute.

Figure 2.4: Proposed visualization of unseen statistics. It displays statistics for two important attributes: price and color. Price is numerical, so the relevant (green) values are displayed as a range. Color is categorical, which is why the relevant values are displayed as a set of values.

The variant described above is rather simple, which is why we developed a more complex variant, *stopping criteria*. We wanted to combine multiple important attributes to generate rules inspired by Market Basket Analysis' Association rules.

The Association rules are in a form $A \wedge B \wedge C \implies D$, meaning if products A, B and C occur in a transaction, then also product D occurs in a transaction. The main use case of these rules is to analyze what items customers buy together.

The stopping criteria variant displays a small number of rules modeling the customer's preferences. The rules are in a form $A \wedge B \wedge C \implies D$ (same as Association rules), but A, B, C and D are sets of values of attributes. This allows us to generate rules such as "Out of products with *red color*, *price between 13,000 and 16,500 CZK* and *6 inch screen size* you prefer products with *512 or 1024 GB storage capacity*". It is possible that the body of the rule is empty, then the rule is represented as D, or "You prefer products with *512 or 1,024 GB storage capacity*".

Stopping criteria also display the number of unseen products satisfying each rule. A product satisfies a rule if all its attributes contained in the rule have the values specified by the rule. Only *red* products with *6 inch screen*, *512 or 1,024 GB storage capacity* and *price between 13,000 and 16,500 CZK* satisfy the rule mentioned in the previous paragraph.

Figure 2.5 shows the proposed visualization of the stopping criteria variant.

2.2.2 Filtered products

Unseen statistics and stopping criteria generate filters that can be applied to the unseen products. A customer can display a list of products satisfying these filters, each product in this list can be added to

Unseen statistics and stopping criteria work with filters, we allow customers to display all products satisfying unseen statistics items and stopping criteria

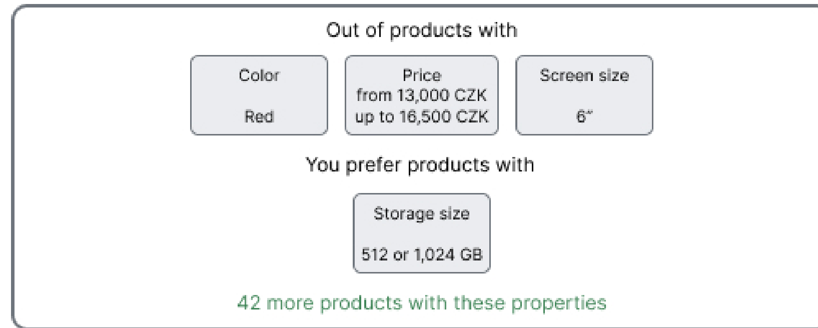

Figure 2.5: A stopping criteria rule representing "Out of products with *red color*, *price between 13,000 and 16,500 CZK* and *6 inch screen size* you prefer products with *512 or 1,024 GB storage capacity*".

items (satisfying any combination of attribute filters). Customers can mark filtered products as candidates or discard them, which means their role is similar to alternatives; they guide the customer when exploring the unseen part of product catalog. Additionally, filtered products are enhanced with the same type of explanations.

3. Data

After we came up with a proposed solution, we needed to gather appropriate data to run our application on. These data needed to satisfy multiple properties described in this chapter for us to be able to present the results in a human-understandable fashion.

3.1 Requirements

We identified the main problem discussed in this thesis in high-risk domains, these domains contain products that customers do not buy very often, but they are rather expensive and used very frequently. Some examples include cars, electronics or holiday trips. We preferred to stay in a similar domain when evaluating the impact of the proposed solution.

Our use case is not the same as the use case of traditional recommender systems. Recommender systems usually need a dataset with user feedback data to build not only content-based, but also collaborative filtering or hybrid model. Our application is based on attributes and content-based approaches, which is why we did not need user feedback data.

Our application was also subject to a user study, which is why we needed all of the data to be human-understandable.

An ideal candidate for our use case was defined as a human-understandable dataset in a content-rich, high-risk domain designed for recommender system (including user feedback data), but a simple product catalog in a similar domain was sufficient enough.

3.2 Existing recommendation datasets

There is a large number of public datasets for recommender systems, but most of them are either hashed or incomplete.

As mentioned earlier, we wanted to use a dataset in a content-rich, high-risk ecommerce domain. There are two well-known datasets that we considered using.

Retailrocket recommender system dataset[1] is a dataset containing data from a real-world ecommerce site. It contains information about the product catalog (category structure and item properties) and user behavior data. The values are not transformed in any way, but they are hashed, which is a serious drawback for our use case. This drawback can be partially bypassed, because numerical values are simply prepended with `n`, but textual values are very difficult, if not impossible, to unhash. The category structure is hashed as well, which means that it is very difficult to select a suitable category for us.

Amazon product reviews dataset[2] is a dataset containing over 500 million reviews of almost 50 million products. The review data is not transformed and human-understandable, but there is not enough data about the products, which was one of our main requirements. The reviews contain only the IDs of products,

[1] Available at `https://www.kaggle.com/datasets/retailrocket/ecommerce-dataset`.
[2] Available at `https://cseweb.ucsd.edu/jmcauley/datasets.html#amazon_reviews`.

so we would need to scrape the properties of products if we wanted to use this dataset.

One promising dataset was provided to us by Recombee, a Czech-based company focusing on recommender systems. This dataset was provided under a non-disclosure agreement, so we can not discuss a any details. The data in this dataset was rich and human-understandable, but the domain of the products was not high-risk. It contained products that people buy frequently, but it is not expensive and so the level of regret is low when you buy a suboptimal product.

Due to the reasons mentioned above, none of the datasets were suitable for our application.

3.3 Scraping

The requirements on the data enabled us to step aside from conventional recommender systems datasets. The main advantage of our use case is the fact that we do need any user feedback data. Our decision was to scrape the data, we selected one of the largest eshops in the Czech Republic, Alza.cz.

The data was scraped in November 2023, the resulting data consists of four categories:

- computers,
- laptops,
- tablets and
- mobile phones.

We scraped the data by parsing the source code of each product's page. Source code of each page was downloaded using Selenium WebDriver. Simple tools such as curl or Python's Requests package were not sufficient because of dynamic content on the website. The downloaded source codes were parsed using Python's beautifulsoup4 package.

We downloaded *image*, *name*, *attributes* and *rating information* for each product.

3.3.1 Processing

The data we downloaded needed to be converted to a format suitable for our application. This process was time-consuming, but not very interesting, which is why we leave this section rather brief.

We wanted to keep the technical requirements of our application low to provide a pleasant experience to the customers, which is why we reduced the resolution of the downloaded images.

Although we scraped the data from the English version of the website, information about some products was not translated to English from the original Czech version of the website. We created a script, that translated the names and values of all attributes to English using manually generated rules. The product names were not translated, so some of them still contain Czech words.

Attribute values required special attention because we downloaded the data directly from the source code of the page; they were textual, had different units, formats and some of them even contained pop-up text descriptions.

All these modifications were performed by Python scripts with manually defined rules specifying how to convert each value of each attribute.

3.4 Dataset

Let us describe the properties of the dataset we prepared for our application. We downloaded data for four categories of products, we consider data of a single category as a separate dataset in this section, because products of different categories have different attributes.

A dataset consists of four parts:

- products,
- attributes,
- images and
- average ratings.

3.4.1 Products

This part of our dataset contains information about each product. A product is represented as a set of values of all its attributes, which is why we store all products of a single category in a single CSV file.

Columns of this file are labeled by names of individual attributes, each row represents a single product and contains values of all attributes defined for the given product category. Some attribute values might be missing, because not all products of a single category have the same set of attributes defined in the real world, for example *capacity of SSD card* is not defined for a mobile phone without and SSD card slot.

First several columns of this file is not the attributes of products, but information about them that is used by the application and/or is displayed to the customers. These columns are *ID*, *name*, *rating*, *number of ratings* and *availability*, these are not considered as *attributes* in the context of this thesis.

3.4.2 Attributes

We focus on product domains with a large number of attributes, as table 3.1 shows.

Not all attributes are the same and we need to account for their differences when working with them to provide the best possible experience for the customers. Let us classify the attributes as *numerical* and *categorical* and describe each type of attributes separately. We store all attributes of a category and their properties in a single JSON file as a list of objects containing union of all properties of each attribute type and *name*, *type* (numerical or categorical) and *group* of the attribute (attributes describing a display of a laptop are grouped to *display* group).

Table 3.1: Number of products and attributes in our dataset.

Category	Products	Attributes
Computers	958	76
Laptops	2421	80
Mobile phones	1199	86
Tablets	325	43

Numerical

Values of numerical attributes are represented as numbers. The application can compare these values directly, compute their differences and other mathematical operation, some of which are described in chapter 4.

Some numerical attributes have a unit that all values have been converted to, the rest of the numerical attributes do not need units. This allows us to save only the number as the value to a corresponding product and do not perform any conversions during runtime of the application.

We wanted to help customers when deciding which value of an attribute is better, which is why we introduce a property *order*. This value can be *ascending* or *descending* and tells the application which way the performance of products grows, in other words, whether a higher value of an attribute is better or worse.

Some numerical attributes behave similarly to categorical attributes, even though they are represented as numbers. An example of this attribute is *size of operational RAM*, because there are common values that this attribute can have: *8 GB*, *16 GB*, *32 GB* and so on. It is not common for this attribute to have a value *14.37 GB*, so we introduce a boolean property of attribute called *continuous*, which specifies whether the attribute can have any value (is continuous) or whether there is some "standardized" set of values[3].

We introduce two more properties for numerical attributes, these are used when generating ranges of values.

Step specifies the size of the interval that we generate when we want to filter products by a continuous numerical attribute. If we want to generate a range containing a given value, we start at the interval $[0, step]$ and work our way up or down by *step* until we reach an interval containing the given value.

Round decimals specifies how many decimal places to round values to. We have a different approach to generating continuous numerical attribute ranges, which utilizes cumulative density function and is described in section 4.4. This approach generates the interval from the data, which can have very specific values, showing customers a price range *from 14,847 up to 16,539 CZK* is not as nice as rounding the range to *from 14,000 up to 17,000 CZK*[4]. The purpose of this property is to present *nicer* values to the customers.

[3] Attributes with different value of *continuous* differ in the filters they generate. Continuous attributes generate ranges of values, non-continuous generate sets of options. These filters are used in stopping criteria and unseen statistics. This distinction was initially reasonable, but we decided to direct our effort on different features and leave this improvement for future work

[4] Note that we do not round the values to the nearest thousands, but we expand the range in both directions. It could happen that we would lose the value we want to generate the interval around if we used standard *rounding* approach.

Categorical

Categorical attributes do not have a specified representation. The values can be textual, numerical, boolean, their combinations or complex objects. An example of a categorical attribute is *color*, the values of these attributes are typically not ordered.

There is only one easily implementable relation of values of categorical attributes: equality. Other relations would need additional data, for example an ordering of values generated by an expert.

Categorical attributes have only one additional property: *is list*. This property specifies whether an attribute has values that are lists of multiple values. The attribute mentioned above, *color*, can have a list of values. Imagine a laptop that has two colors: black and white. The *color* of this laptop would be represented as *[black, white]*.

It is possible to design more complex operations than the the ones with standard categorical attributes, some examples can contain *value a is contained in value b* and standard set operations including union or intersection. Although we did not implement advanced handling of list attributes, we prepared the interface for future improvements of the application. List attributes are not common in our dataset, so we handle them as standard categorical attributes and this property is not set for any of the attributes.

This means that value *[black, white]* does not satisfy filter *[black]*, but not even *[black, white, blue]*. It only satisfies the same value: *[black, white]*. Implementation of this feature would not improve the application in a significant way for our purpose.

3.4.3 Images

Each product has one image assigned to it, this makes the customer's experience better because he/she can visualize the currently inspected product. Each image is named in the format *id.jpg*, where *id* is the ID of the product assigned to it in the products part of our dataset, described in section 3.4.1.

3.4.4 Average ratings

We wanted to show customers which values of attributes are better even if they do not have defined order. Therefore, we decided to use rating of other users for this. Users of the Alza.cz eshop assigned ratings to individual products, but we wanted to assign ratings to all *(attribute, value)* pair.

The average rating $r_{a,v}$ of attribute a with value v was computed as $r_{a,v} = \frac{\sum_{p \in P_{a,v}} r_p \cdot c_p}{\sum_{p \in P_{a,v}} c_p}$, where $P_{a,v}$ is a set of products where attribute a has value v, r_p is rating of product p and c_p is number of ratings of product p.

This way, we can compare values of attributes even if we do not have the data of their actual influence on the product performance, better-rated *(attribute, value)* pairs are considered better in this context. Naturally, customers are informed that these comparisons are based on ratings of other customers and not their actual performance.

4. AI models

Our application has several AI features including recommendation, explanations and stopping criteria. Each feature is handled by a specialized model. In this chapter we describe how all these models work.

4.1 Recommender system

First described model takes care of providing alternatives to already selected candidates.

We implemented two similar recommender system models, both are content-based and use only the current application's state as data. Only the current set of candidates and discarded products is considered as customer's feedback data, which is why we can categorize these models as session-based.

Customer preferences are modeled as a vector in product space. The product space in which these models live is different to the one described in section 2.2. The product space in this context is defined as a vector space of dimension $d = \sum_{a \in I} |V_a|$, where I is the set of all important attributes and V_a is the set of all values that attribute a has.

The idea behind this product space is to assign preference to each attribute value based on candidates and discarded products, inspiration for the final implementation was provided by Balog et al. [2019]. The inspiration was based on pairwise interactions in the customer model presented in the paper, these interactions are able to model the following situation: "A customer likes A especially if B". These interactions could easily be transformed to suit our use case, if we considered A and B to be *(attribute, value)* pairs.

We wanted to provide similar form of explanations and decided it is a good idea to keep the recommender system as close to the explanations generator as possible. Although it is not an uncommon practice to implement a different form of explanations generator (for example content-based explanations with collaborative-based recommendations), we were skeptical to doing so in content-rich domain. We wanted the base of recommender system to be similar to the base of explanations generator to produce related results. Explanations generator is described in more detail in section 4.2.

Each product is represented as a binary vector $\in \{0, 1\}^d$ in the product space defined above. The vector representing product p has the value 1 at a position (a, v)[1] if and only if product p has value v of attribute a.

Let us formalize the customer model. Here, the two implemented models differ. As mentioned earlier, both models represent customers as single vectors $\in \mathbb{R}^d$ in the product space.

The first model assigns the following value $r_{a,v}$ to the customer model at position (a, v):

[1] The position is described as a tuple in this case, but all values of all important attributes are flattened to conform to the product space, so the real position of vector is a single number. We used tuple to emphasize the meaning of a position in a vector.

$$r_{a,v} = \frac{|C_{a,v}|}{|C|} \tag{4.1}$$

where C is the set of all candidates and $C_{a,v}$ is the set of all candidates that have value v of attribute a.

The second model is similar, but considers the discarded products as well. The value $r_{av,}$ in the second model is defined as follows:

$$r_{a,v} = \frac{|C_{a,v}|}{|C|} - \frac{|D_{a,v}|}{|D|} \tag{4.2}$$

where D is the set of all discarded products and $D_{a,v}$ is the set of all discarded products that have value v of attribute a.

The customer model benefits values of attributes that the customer adds to candidates and (in the case of second model) penalizes values of attributes that the customer discards.

The final score s_p is computed as a sum of a pointwise multiplication of a product p and the customer model c (we can consider the pointwise multiplication as applying a product mask to the customer model), formally:

$$s_p = sum(p \cdot c)$$

where sum is the sum of the vector's coefficients ($v \in \mathbb{R}^d : sum(v) = \sum_{i=1}^{d} v_i$, where v_i is the i-th component of vector v) and $\cdot$ is a dot product of two vectors.

Each customer's model represents his/her preference of all *attribute, value* pairs and the product's score sums the preferences of all *attribute, value* pairs the product has.

Our application works with different attribute types and recommender system has to be able to handle all of them. Multiple approaches to handling for example continuous numerical attributes exist nowadays, these approaches include binning or modifications of the formulae to consider differences and other operations with different values of continuous attributes.

We initially did not modify the algorithm to account for different types of attributes than discrete, where the one-hot encoding we mentioned above makes sense. The amount of data we use in our application allowed us to use the one-hot encoding for continuous attributes as well, which is sufficient for the requirements we pose on our recommender system. Improvement of these formulae or some form of pre-processing of attributes of different types is mentioned in chapter 7.

4.1.1 Diversification

One of the known problems of content-based recommender systems is over-exploitation, they tend to recommend similar set of products if no post-processing is applied. We needed to overcome this problem, because recommendation of alternatives is meant to allow the customer explore the product domain. We implemented post-processing to the data provided by the recommender system to increase the diversity of the generated set of alternatives.

This process is the same for both models. We select the product with the highest score and the rest of alternatives is selected with respect to the following procedure.

All products are separated into bins according to their score bounded by whole numbers, a product p is in bin b if and only if $l_b < s_p \leq u_b$, where l_b is the lower bound of bin b, s_p is the score of product p and u_b is the upper bound of bin b. We then select the bin from which to select a product. Each bin is selected with different probability, bins with higher scores are selected more probably, the weight w_b of selecting a bin b is $w_b = u_b + 1 - min\{u_c : c \in B\}$, where u_b is the upper bound of bin b, B is the set of all bins and u_c is the upper bound of bin c. This ensures that the bin with the worst score has weight 1. A bin b is selected with probability $p_b = \frac{w_b}{\sum_{c \in B} w_c}$. After a bin is selected, a product from that bin is selected uniformly randomly.

The complete algorithm recommending alternatives to a customer is illustrated in algorithm 4.1. The entrypoint of the recommender system is the function *Predict*, which assigns scores to all unseen products. These scores with the product IDs are then passed to the function *PostProcess*, which performs the diversification and returns the resulting list of alternative product IDs. The function *BuildCustomerModel* generates the customer model according to equation (4.1) or equation (4.2) depending on the selected recommender system model.

Algorithm 4.1 Algorithm recommending alternatives.

```
 1: function Predict(candidates, discarded, unseen)
 2:     customer ← BuildCustomerModel(candidates, discarded)
 3:     Initialize an empty array scores
 4:     for product ∈ unseen do
 5:         score ← sum(customer · product)
 6:         Append (product, score) to scores
 7:     end for
 8:     alternatives ← PostProcess(scores ordered by score from highest to
    lowest)
 9:     return alternatives
10: end function
11: function PostProcess(scores)
12:     Initialize empty array alternatives
13:     Append first product from scores to alternatives and remove it from
    scores
14:     Separate products from scores into bins by their score
15:     while There is not enough alternatives do
16:         bin ← random bin b with probability p_b
17:         product ← uniformly random product from bin
18:         Append product to alternatives
19:         remove product from scores
20:     end while
21:     return alternatives
22: end function
```

4.2 Explanations generator

Our application enhances products with explanations, we implemented a model capable of doing so. This model is capable of explaining values of individual attributes as well as the whole products, the goal of explanations was to provide more information about the products and their attributes to the customers, especially if they are not very familiar with the current domain.

We could consider the implementation of the explanations generator as hybrid, because it uses both the product and customer data. We mentioned in the previous section that we wanted to use similar models for recommending products and for generating explanations. This is why we use the customer data to explain individual attributes and their values of the products, we try to enhance the content of the products, not to persuade the customers to interact with a given product in any way because of preferences of other customers. Each product is explained individually, the model generates all explanations for a given product which include the whole product's explanation and individual explanations of all its important attributes.

It is possible to separate the products into different groups and each group has a different purpose, the explanations we generate need to be designed to support the purpose of the product group they are displayed in. We can classify the explanations into two categories, *unseen products versus all candidates* and *candidate versus other candidates*, that we describe separately.

4.2.1 Unseen products versus all candidates

The first type of explanations is designed to help a customer when exploring the product domain, we generate this type of explanations for every unseen product, that is being presented to the customer. It is not considered whether the product is among alternatives or filtered products, the customer is expected to decide whether he/she wants to add that product to candidates or discard it.

The customer discards a product if he/she decides the product does not fit into his/her current set of candidates. This is the idea we had when designing this type of explanations: we compare an unseen product with the current set of candidates.

Attributes

Let us start by describing the explanations of individual attributes, because they are simpler than the explanations of the whole products. We show the explanations before describing the process of generating them. The form of explanations differ based on the attribute they describe, we will start with *ordered* attributes:

- worse than all candidates,
- relevant value or
- better than all candidates.

The meaning of relevant value in this context is described further in this section. *Unordered* attributes have explanations of a slightly different form, which indicates the usage of data of other customers:

- lower-rated than all candidates,
- no explanation or
- higher-rated than all candidates.

The colors used in these lists are the same as when the explanations are presented to the customers; the boxes containing the attribute names and their values are colored as displayed in figure 4.1. This makes the layout of the application cleaner, because we can hide the text of the explanation into a popover and a customer understands the nature of the explanation immediately.

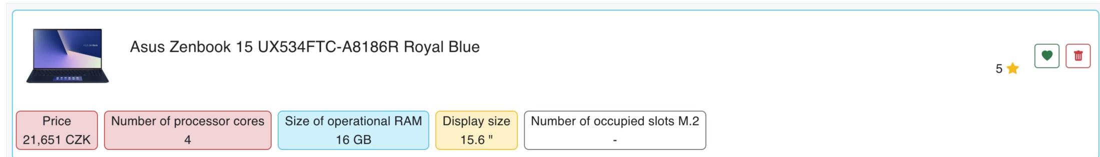

Figure 4.1: A product with color-coded attributes and their values. *Price* and *number of processor cores* is worse than all candidates, *size of operational RAM* is better and *display size* is lower-rated than all candidates. *Number of occupied slots M.2* is not explained.

The process of obtaining both forms of the explanations above is very similar. We generate a relevant range of values of an attribute based on the current set of candidates. The relevant range of values of attribute a is simply an interval $[\min\{p_a : p \in C\}, \max\{p_a : p \in C\}]$, where p_a is the value of attribute a product p has and C is the set of candidates. The model is able to identify which value is *better* or *worse* than all of the candidates if the value lies outside of the interval. The attribute's *order* property is used to determine if lower or higher values are better.

If an attribute is not ordered, the model generates explanations based on average ratings of individual *(attribute, value)* pairs, described in section 3.4.4. The *relevant* range is represented as an interval $[\min\{r_{p_a} : p \in C\}, \max\{r_{p_a} : p \in C\}]$, where r_{p_a} is the average rating of attribute a and its value that product p has.

Products

We implemented explanations of the whole products, which were meant to aggregate the explanations of individual attributes and provide additional information to the customers about the product domain. The following forms of explanations can be generated for the whole products:

- better than all candidates,
- no explanation or
- worse than all candidates.

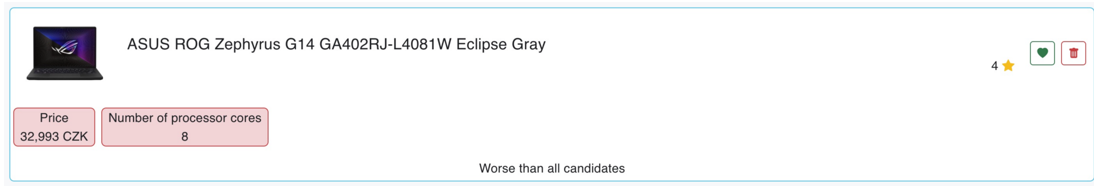

Figure 4.2: A product with values of all its important attributes worse than all candidates.

These forms of explanations are not color-coded, because we display the text of the explanation under the line with product's attributes, so customers see the text immediately. A product with this explanation is shown in figure 4.2.

As the textual representation of the explanations indicates, a specific setup of important attributes is required. These explanations are displayed only if all important attributes are *ordered numerical*. A product is explained as *better* or *worse* than all candidates if and only if all its important attribute values are *better* or *worse* than values of all candidates.

4.2.2 Candidate versus other candidates

The second type of explanations is similar to the first, we compare a product to candidates. However, in this case, the compared product is a member of candidates and we compare it to the rest of this set. The idea behind this implementation comes from the purpose of these explanations: we want to highlight the differences of individual candidates, because the customer is expected to select the best product from the set of candidates.

Our model does not implement explanations of the whole products when comparing a candidate to other candidates, so we will focus only on explanations of the individual attributes. Similarly to the first type of explanations, different forms of explanations are generated for different attributes. The model generated the following explanations for ordered attributes:

- worst of all candidates,
- no explanation or
- best of all candidates.

Unordered attributes are explained by one of the following explanation:

- lowest-rated of all candidates,
- no explanation or
- highest-rated of all candidates.

The process of obtaining the explanations is similar to the previous type; we generate a range of values of an attribute from all candidates and compare the value of an attribute of a given candidate. An attribute and its value is marked

as *worst*, *best*, *lowest-rated* or *highest-rated* only if the value is a unique maximum or minimum of the generated range. In mathematical terms, a candidate's value p_{a_v} of attribute a is marked as *best* of all candidates if and only if it holds that $\forall q \in C \setminus \{p\} : q_{a_v} \prec p_{a_v}$, where $a \prec b$ represents *a is worse than b*. Direction of *worse* or *better* values is again identified by the attribute's *order* property.

4.3 Unseen statistics generator

We developed two approaches to presenting information about the unseen product to the customers, unseen statistics is the more simple one, which is why we describe it first, stopping criteria follows.

Unseen statistics generator works similarly to explanations of individual attributes described in the previous section. It identifies relevant values of all important attributes: ranges of numerical attributes and subsets of values of categorical attributes. The differences in the visualization of these approaches is presented in figure 2.4.

The generation of the relevant range of numerical attributes is identical to attribute explanations described in section 4.2. We consider a single interval of numerical attribute values to be relevant for the customer. This is a simplification of the real world, where it is possible that a customer would be interested in two distinct intervals. For example a customer could be looking for one of two laptops: either a small one, for which he/she would purchase an external monitor, or a large one, which he/she will use without an external monitor. It did not seem necessary to implement this feature as a part of our solution.

Identification of the relevant values of individual attributes allows us to model the preferences of the customers and we can apply this model to the unseen products to provide personalized information about them. The information provided to the customers is composed of two components: identification of the customer's model and the amount of unseen products complying to said model.

Let us illustrate this with a concrete example. Suppose a customer is looking for a laptop and he/she has selected several notebooks with a 13 and 14-inch display as his/her candidates. An unseen statistics item is generated, which identifies the relevant values range (an interval from 13 to 14 inches) and a number of unseen products with *display size* lying in this interval. The customer can display the whole list of unseen products satisfying the filter.

This model is rather simple, because it considers only a single attribute when it is generated. We considered combining multiple unseen statistics items to generate more complex information, which we achieved in the form of *stopping criteria* described below.

4.4 Stopping criteria generator

Stopping criteria generator generates several[2] items (rules) in the form $A \wedge B \implies C$, where A, B, and C are filters of individual attributes. We want to generate the rules that describe the customer's preferences the best but keep a reasonable

[2]The default number of rules to be generated is 5, but this value can be configured.

level of diversity in the provided set. A brief introduction to the rules is provided in section 2.2.1.

Let us describe the metric we used to rank individual rules. We identified several requirements on the metric r:

- $|C_1| > |C_2|, |D_1| = |D_2| \implies |r_1| > |r_2|$
- $|C_1| = |C_2|, |D_1| > |D_2| \implies |r_1| < |r_2|$
- $|C_1| = |C|, |D_1| = 0 \implies$ maximum value
- $|C_1| = 0, |D_1| = |D| \implies$ minimum value

In the above list of requirements, C_i is the set of candidates satisfying rule i, D_i is the set of discarded products supporting rule i and r_i is the score of rule i. C is the set of all candidates and D is the set of all discarded products. A product supports a rule if all its *(attribute, value)* pairs correspond to the attribute values of the product.

To put the requirements into words: we wanted to prefer filters frequent among candidates, but penalize the ones frequent among discarded products.

We defined the final metric as $r_{S\cup\{a\}} = \frac{|C_{S\cup\{a\}}|-|D_{S\cup\{a\}}|}{|C_S|}$, where S is the set of filters, a is a filter, C_S is the set of candidates that satisfy S and D_S is the set of discarded products that satisfy S. This metric satisfies all of our requirements defined above.

Because we were inspired by Association rules, we considered using existing algorithms used to generate them, such as the Apriori algorithm [Agrawal et al., 1996]. This algorithm generates more and more complex association rules, but cuts off rules that have low support (are not frequent enough in the data). Our metric did not allow us to use this algorithm. Consider a customer with 100 candidate jackets and no discarded ones. 90 of them have are red and 10 of them have green color, all of the green jackets have buttons, 50 of the red jackets have buttons and 40 zipper. What rule is *stronger*? “Out of the red jackets, you prefer the ones with buttons” or *Out of the green jackets, you prefer the ones with buttons*? According to our metric, it is the second one ($r_1 = 0.56$, $r_2 = 1$), but if we used the Apriori algorithm, it could cut off the generation of supersets of $S = \{a\}$, where a is the green jackets, because its support is only 0.1.

Apriori algorithm works because the support of a set is always higher than or equal to the support of all its supersets. We do not have this guarantee, so we can not cut off the generation of more complex rules, however, our approach is similar to the Apriori algorithm, which can have exponential time complexity with respect to the unique number of items in the database in the worst case. This is why we analyze the time complexity of our algorithm below.

We developed algorithm 4.2, which generates the stopping criteria rules. It generates rules for all combinations of filters that can be generated from attributes and values of the candidates. The entrypoint of the algorithm is the function *Generate*, which collects support set of each filter (from candidates and discarded products). These support sets are used to compute the metric of each rule defined above, we exploit the fact that $C_{S\cup T} = C_S \cap C_T$ and do not need to collect the support set for each rule.

Algorithm 4.2 Algorithm generating stopping criteria rules.

```
 1: function Generate(candidates, discarded, attributes)
 2:     filters, support_sets ← CollectSupportSets(candidates, discarded, attributes)
 3:     Initialize an empty array rules
 4:     for S_size = 1, 2, ..., N do
 5:         for S_attrs ∈ {S : S ⊆ attributes, |S| = S_size − 1} do
 6:             for a_attr ∈ {a : a ∈ attributes, a ∉ S_attrs} do
 7:                 Initialize an empty array S_all
 8:                 Initialize an empty array a_all
 9:                 for product ∈ candidates do
10:                     Add filters[product_{S_attrs}] to S_all
11:                     Add filters[product_{a_attr}] to a_all
12:                 end for
13:                 for (S, a) ∈ S_all × a_all do
14:                     r_{S∪{a}} ← ComputeMetric(S, a)
15:                     Add rule S ∪ {a} to rules
16:                 end for
17:             end for
18:         end for
19:     end for
20:     return rules
21: end function
22: function CollectSupportSets(candidates, discarded, attributes)
23:     Initialize an empty mapping filters
24:     Initialize an empty mapping support_sets
25:     for attribute ∈ attributes do
26:         for product ∈ candiates do
27:             if product_{attribute} ∉ filters then
28:                 filter ← GenerateFilter(product, attribute)
29:                 Add filter to filters
30:                 support_set ← GetSupportSets(filter, candidates, discarded)
31:                 Add support_set to support_sets
32:             end if
33:         end for
34:     end for
35:     return filters, support_sets
36: end function
```

The function *CollectSupportSets* collects the support sets of all the filters available in the candidates set. We generate a filter for a given attribute and product in the following way.

If the attribute is continuous numerical, we find the median value of the candidates and generate an interval around this value containing $c\%$ of lower and $c\%$ of higher values, where c is a configured constant. This way, we generate only one filter for a continuous numerical attribute.

Otherwise, we add only the product value of the attribute to the filter options. No other value satisfies the filter than the one a given product has. This way we generate the same number of filters as there is distinct values of a given attributes in the whole set of candidates (no more than $|C|$).

The function *GetSupportSets* selects the product IDs of all candidates and discarded products satisfying a given filter.

The number of filters generated by the function *CollectSupportSets* is $\mathcal{O}(|C| \cdot |A|)$, where A is the set of important attributes. Assuming each of the functions *GenerateFilter* and *GetSupportSets* has time complexity $\mathcal{O}(|C|)$, the time complexity of the whole function *CollectSupportSets* is $\mathcal{O}(|C|^2 \cdot |A|)$.

After the support sets are collected, we generate rules by combining the individual filters. We generate filters up to length (number of filters) N, which is constant. This means that we generate $\mathcal{O}(f^N)$ rules, where f is the number of filters obtained from function *CollectSupportSets*, meaning we generate $\mathcal{O}((|C| \cdot |A|)^N)$ rules.

We compute the metric for every rule we generate, the metric is computed in constant time if we know the support set of a rule. We do not know the support set, but need to construct it from the support sets of individual filters. Intersection of two sets S and T has time complexity $\mathcal{O}(\min\{|S|, |T|\})$, we know that the sets are subsets of C or D. $\min\{|C|, |D|\} \leq |C|$, so the final time complexity of computing the metric of one rule is $\mathcal{O}(|C|^N)$.

The total time complexity of the function *Generate* is $\mathcal{O}((|C| \cdot |A|)^N \cdot |C|^N) = \mathcal{O}((|C| \cdot |A|)^{N+1})$, which is not exponential, but polynomial with the exponent being the maximum complexity of a stopping criteria rule.

4.4.1 Diversification

Similarly to the recommender system, we apply diversification post-processing to the generated rules. We first order the rules by two properties: score and complexity. A score is the value of the metric r and complexity is the length of the rule (number of its filters). Score is the primary property for sorting and $-$complexity secondary. This means that simpler rules are preferred, if multiple rules have the same score, this helps with exploration of more products in the catalog, because simpler rules are satisfied by more products.

Sorting the rules has time complexity $\mathcal{O}(n \cdot \log(n)) = \mathcal{O}(|C| \cdot |A| \cdot \log(|C| \cdot |A|))$, where n is the number of rules.

The post-processing algorithm is shown in algorithm 4.3. We return a diverse subset of ordered rules *rules*, the subset is built rule by rule. We take the best rule we have not yet processed and compute the maximum similarity it has with any of the already used rules. The score of the current rule is penalized if the similarity is high. The score of each rule can only drop, which means that we can

cut off the for loop on lines 6-17 as soon as the score of the current rule drops below the penalized score of the best rule we have found so far.

Algorithm 4.3 Algorithm post-processing stopping criteria rules.

```
 1: function PostProcess(rules)
 2:     Initialize an empty array result
 3:     for i = 1, 2, ·, M do
 4:         Initialize best_rule with empty value
 5:         Initialize best_score with empty value
 6:         for rule ∈ rules \ result do
 7:             if rule_score < best_score then
 8:                 Add best_rule to result
 9:                 continue with outer For loop
10:             end if
11:             similarity ← MaxSimilarity(rule, result)
12:             rule_new_score ← rule_score · (1 − similarity)
13:             if rule_new_score > best_score then
14:                 best_rule ← rule
15:                 best_score ← rule_score
16:             end if
17:         end for
18:     end for
19:     return result
20: end function
```

We use Jaccard similarity to compute the similarity of two rules. We consider each rule as a set of all its filters; the similarity of rules a and b is $s = \frac{|F_a \cap F_b|}{|F_a \cup F_b|}$. The time complexity of computing *MaxSimilarity* is $\mathcal{O}(|M| \cdot |N|)$, where N is the maximum complexity of a rule, so the total time complexity of the function *PostProcess* is $\mathcal{O}(|M|^2 \cdot |N|)$.

The time complexity of algorithm 4.4 is $\mathcal{O}((|C| \cdot |A|)^{N+1}) + \mathcal{O}(|C| \cdot |A| \cdot \log(|C| \cdot |A|)) + \mathcal{O}(|M|^2 \cdot |N|)$, which can be simplified to:

$$\mathcal{O}((|C| \cdot |A|)^{N+1} + |M|^2 \cdot |N|)$$

Because $N \geq 1$. The time complexity of the whole algorithm is polynomial, which makes it reasonably fast for our application.

Algorithm 4.4 The whole algorithm generating stopping criteria rules.

```
1: function StoppingCriteria(candidates, discarded, attributes)
2:     rules ← Generate(candidates, discarded, attributes)
3:     Sort rules by (score, −complexity)
4:     result ← PostProcess(rules)
5:     return result
6: end function
```

5. Implementation

We implemented a web application consisting of two parts: *server* and *GUI*. Our goal was to develop a working instance that implements all the functionalities proposed in chapter 2 while following the latest practices and trends. We focused on keeping the application easy to maintain and extend with new features.

Each part of the application is deployed as its own Docker container, which communicate with each other through a REST API.

5.1 Repository

The source code for this entire application is present in a public GitHub repository available at `https://github.com/matejscerba/Thesis`. We also have a running instance; its current URL is written in the README file of the repository, along with the technical information needed to maintain and extend the code, which is why we do not want to go into too much detail. In the remainder of this chapter we will provide general information about the technologies, architecture, and development process used.

5.2 Server

The first part of our application is the back-end application *server*. This application handles all requests sent to it from the front-end application (*GUI*), it performs all complex computations in real time and contains the implementation of all AI models described in chapter 4.

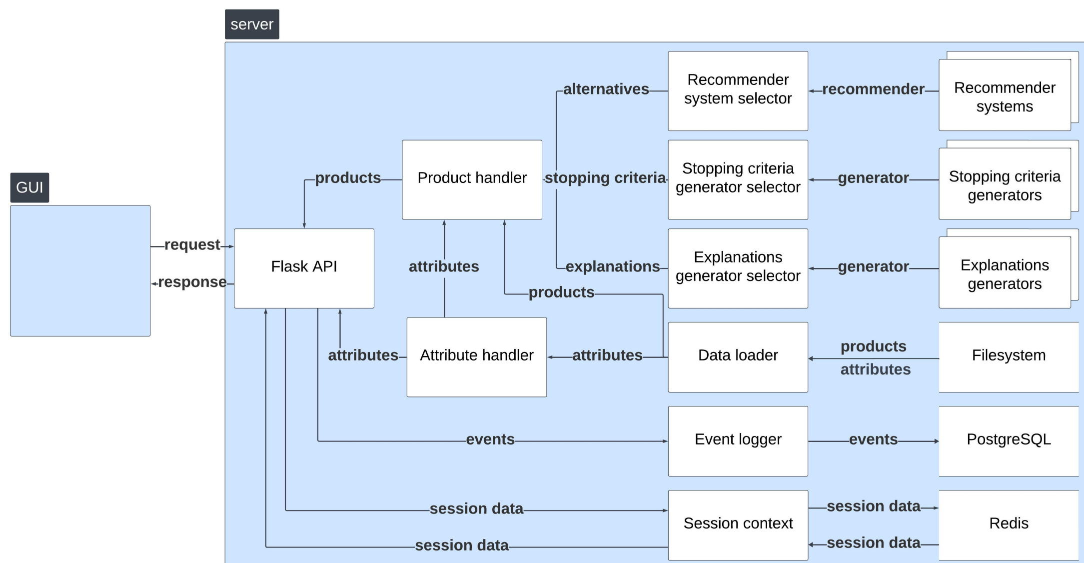

Figure 5.1: Diagram showing architecture of the application.

5.2.1 Tech stack

The server is a web application written in Python, using Flask framework, which allows it to be accessible over HTTP. The application handles most of its data as objects; we use Pydantic package to handle their definitions and serialization. The product data are stored in the form of JSON and CSV files[1]. The CSV files are processed using Pandas package. The server uses two storage solutions, a PostgreSQL database and Redis. PostgreSQL database is used for logging user events and mapping of IDs in the case of user study setup, while Redis is used for storing session data of all users.

5.2.2 Architecture

The source code of the server is object-oriented, the processing of the data is performed in Python classes. We can see the architecture in figure 5.1. Let us describe how a HTTP request is processed in the server.

A request is handled by Flask and passed to a view function that processes its content, passes data to, and receives data from other components of the application and returns an HTTP response afterwards. *Product handler* handles all operations with products, including organization of the category, working with the attributes and triggering recommendations, explanations, stopping criteria and unseen statistics generations[2].

We implemented three different AI model selectors, which are capable of selecting the configured implementation of the given AI model and use it when processing a request. These models include the *recommender system*, *explanations generator* and *stopping criteria generator*.

Attribute handler handles all operations with attributes, it can, for example, parse the raw data provided by *Data loader*, which loads the product and attribute data from the filesystem.

Event logger is designed to log each event performed in GUI into a PostgreSQL database. These events include, for example, *adding a product to candidates*, *opening a stopping criteria item* or *submitting a user study questionnaire*. We have two tables in the database, one for logging events (named `events`) and one for mapping between session IDs and Prolific[3] IDs (named `prolific`).

All user information is stored in *Redis* and it is managed by *Session context*. The application is stateless, however, we store some data when a request is being processed to be able to access them anywhere in the application (for example when logging an event; we need to know which user performed the action).

[1] Production system would benefit from storing the data in an external database, but it was sufficient for us to keep the data in files, because our dataset was static.

[2] Note that unseen statistics generator is not a standalone model. It is included in the *Product handler*, because the unseen statistics was the initial implementation of aggregating information about the unseen products and we had no need to separate it into a standalone model initially.

[3] Prolific is a platform providing participants to a user study, it is described later in the text.

5.2.3 Configuration

It is possible to select which AI model implementation to use, their parameters, what mode to run the application in, set connections to the storages, and configure of the server itself using the environment variables; details are described in the README file of the repository. We will describe the most interesting configuration because of how it is handled in the application: selection of an AI model and the difference between *production* and *user study* mode.

The selection of an AI model starts by reading the environment variable containing its name. Suppose that we are selecting a recommender system, and the application reads an environment variable `RECOMMENDER_MODEL`. When selecting a model, the selector checks all the available implementations[4] (subclasses of an abstract superclass) and uses the one with the same name as the environment variable has. That model is passed to *Product handler*, that uses the model to process the data according to its needs.

We support two modes the application can run in (represented as `AppFlowType` enumeration in the code): *production* and *user study*. The production mode displays a menu containing all available product categories, a customer can select and display one category. He/she can browse the category without selecting any product as a final choice.

The user study mode displays the tutorial as the homepage, which describes all the information a participant needs in the user study. We support Prolific integration in the user study mode. Prolific users are redirected to our user study, so we need to collect their ID and store it in the database. Each participant is then presented with several questionnaires and product categories, where he/she simulates real world-like selection of a product and provides feedback.

5.3 GUI

The second part of our application is *GUI*, a front-end application. This application presents all data to the customers, who do not directly interact with any other part of our system.

5.3.1 Tech stack

This application is built using React framework and its source code is written in TypeScript to take advantage of type checking. React Router handles the routing inside the application. We use React Bootstrap framework to handle styling using CSS classes. The dockerized application is served using Nginx server.

5.3.2 Architecture

The navigation of GUI is handled by React Router, which specifies which components should be rendered at what page. The components organized into folders, each folder contains components related to each other (folder `menu` contains all

[4]The looping through all the implementations is performed only if no prior selection was performed. The selection is static (based on the environment variable), so it is sufficient to perform it only once.

components rendering menu of a product, and so on). We used this layout to keep the code clean and organized.

We use multiple contexts that hold the state of the application and pass it to multiple components. There is for example a context holding the whole state of the organization of the category: the set of candidates, discarded products, the alternatives provided by the server and all methods used to modify the state (adding a product to candidates or discarding it).

The auxiliary functions are defined in their own folder that contains several files working with the objects handled by GUI: attributes, products, stopping criteria, API calls, or configuration of the whole application. The types representing all these objects are also defined in a specific folder.

5.3.3 Configuration

GUI does not have any form of configuration, it loads all the necessary information from the server. The GUI only needs to know the URL of the server, which is hardcoded in the current solution.

5.4 Code maintenance and development

The GUI dependencies are managed by npm and the server dependencies are managed by Poetry.

The source code of both applications is statically checked, the GUI checks the code using Eslint and formats it with Prettier supporting TypeScript, the server checks the code using Flake8 and Mypy and formats it with Black. This means that we are using type hints in Python.

Types are also documented directly in the code. The JSDoc notation is used in the GUI and the reStructuredText (reST) notation is used in the server.

Pre-commit hooks are configured to run all the static code checks and formatters, so no unchecked code reaches the repository. After a commit is pushed to the repository, GitHub actions run the same checks for both applications and try the docker compose command to see if the container can be created and started properly.

6. User study

One of our goals was to evaluate the impact of our application. We would eventually like our application to be deployed to a production system where real customers would utilize the features we developed and purchased products they wish. It is not possible (or at least not a good idea) to deploy a new technology to a production system without any testing.

We created a user study to evaluate our application in a controlled environment. We deployed the application to a publicly available website and invited several participants to access the website and complete our user study. We addressed several people who we personally know to enter the study, but the number of submissions was low, which is why we published our study on Prolific, a web platform that provides participants to user studies.

The description of our goals and the list of hypotheses we aim to test is provided in section 6.1. The setup of our application and the study is described in detail in section 6.2, participants were asked to browse two different categories with two different user interface (UI) variants: *unseen statistics* and *stopping criteria*. And finally, the results of our user study and its conclusion are provided in section 6.3.

6.1 Goal

Our high-level goal was to show that the application we implemented helps our customers in terms of the problem components identified at the beginning of this text, in chapter 1. Let us review these components:

- lack of explicit feedback data,
- lack of storage of candidate products and
- lack of cut-off alerts.

It is apparent that the actions customers perform in our application generate explicit feedback data, but it is important to ask whether the actions generating explicit data are not annoying. It would be possible to design an application where a customer would be asked to rate a product he/she just viewed with a pop-up window, but this interaction could very likely be considered annoying and the customer would likely never return to an eshop with such a feature.

We specifically developed the possibility to organize the products into different groups. One of the groups was designed to hold every candidate a customer encounters when browsing a product catalog; this problem component is apparently resolved as well.

The most complicated component to evaluate is the lack of cut-off alerts, because our application does not generate alerts in a form similar to a pop-up text "stop looking further and focus on candidates", but rather provides information about unseen products and allows customers to interpret it on their own. This is why our user study was designed to evaluate whether the information provided could be considered as cut-off alerts.

We decided to compare the two UI variants we implemented: *unseen statistics* and *stopping criteria*, while expecting the stopping criteria to be evaluated as better by the participants. Unseen statistics were developed initially and we decided to improve them after a preliminary testing performed by a few of our close acquaintances, who pointed out that the unseen statistics were rather simple, we agreed with this statement and decided to overcome this drawback by developing the final version of stopping criteria.

We formulate five hypotheses to test in the user study.

1. Stopping criteria make users more confident that the product they selected was the best option for them than unseen statistics.
2. It is easier to find the final product with stopping criteria than with unseen statistics.
3. Stopping criteria is better for telling users when to stop searching for more products than unseen statistics.
4. Users would more likely want to see stopping criteria in production systems.
5. Users familiar with the product domain find the stopping criteria easier to use.

6.2 Setup

When a participant enters the user study, he/she is greeted with a tutorial describing the application and the user study. This tutorial contains explanations of all the features available in the application and what the participant is expected to do. We wanted to simulate the process of selecting the best product possible from the product domain; what would a maximizer do. All participants were told to select the best product possible from the data multiple times in the tutorial.

As stated earlier, all participants were presented two different categories of products and two different UI variants. We uniformly randomly selected two of the available categories and the order of the UI variants. This form of generation of the setup was supposed to cancel out any bias that could emerge from the order of the UI variants. The time spent in each category was biased by their order, as we can see in section 6.3.

After a participant had read the whole tutorial, an initial questionnaire was presented to him/her. This questionnaire was designed to collect demographic information about the participant how he is familiar with and his/her latest interaction with the product categories to be presented further during the user study. The initial questionnaire consisted of the following questions:

1. *My age:*
2. *My gender:*
3. *My occupation:*
4. *I am familiar with the first domain of products.*

5. *Last time I bought a product from the first domain was:*
6. *I am familiar with the second domain of products.*
7. *Last time I bought a product from the first domain was:*
8. *There are 7 days in a week. [attention check]*

Participants could select a 10-year wide interval when entering their age, we provided three options for gender: *male*, *female* and *other*. Occupation was the only open question of this questionnaire (a participant could write any text in the provided input). Responses to the questions about participant's familiarity with the domains were selected from a 7-point likert scale, containing options *strongly disagree*, *disagree*, *partially disagree*, *neutral*, *partially agree*, *agree* and *strongly agree*. Options for the latest interaction with the product category were *several days ago*, *several weeks ago*, *several months ago*, *several years ago* and *never*. The last question was an attention check, more about these questions is described in section 6.3. Attention checks were all questions on the 7-point likert scale.

Submitting the initial questionnaire lead the participant to the first category of products. The layout was identical to the one described in chapter 2, but each participant had to select one product as his/her final choice. We forced the participants to utilize the set of candidates, because only a candidate could be marked as a final choice. Each participant had to agree to a pop-up alert asking whether he/she is reasonably sure there is no better product for him/her in the rest of the product catalog.

Each category of products was followed by a step questionnaire; this questionnaire contained questions regarding the prior category of products and its UI variant:

1. *I am confident that the product I selected was the best possible option for me.*
2. *The final product was easy to find.*
3. *The system helped me decide when to stop searching for better products.*
4. *I got a lot of recommendations for [skates / hotel rooms] on the previous page. [attention check]*
5. *Working with the system was easy.*
6. *I found [unseen statistics / stopping criteria] useful.*
7. *I would like to see [unseen statistics / stopping criteria] on production systems (other eshops).*

All of these questions were 7-point likert scale. Question 4 was an attention check, because our data did not contain any skates or hotel rooms. We asked for skates in the first step and for hotel rooms in the second step. The last two questions differed on whether the previous variant of the UI was *unseen statistics* or *stopping criteria*.

After a participant had selected a product from both categories and submitted both step questionnaires, he/she proceeded to the last step, an overall questionnaire containing the following questions:

1. *Which UI variant was more helpful in finding good candidates quickly?*
2. *Which UI variant was easier to understand?*
3. *The color of the clear sky is gray. [attention check]*
4. *What makes one UI variant better than the other?*
5. *What other product domains could benefit from similar interfaces?*

The first two questions could be answered by selecting an option from a 7-point UI variant comparison scale. The possible options were *definitely unseen statistics*, *unseen statistics*, *rather unseen statistics*, *both were equal*, *rather stopping criteria*, *stopping criteria* and *definitely stopping criteria*. This question contained screenshots of both UI variants to help participants select the appropriate option. Question 3 was an attention check and the last two questions were open.

6.3 Results

The user study took place in December 2024 and January 2025, we gathered data from 70 participants. We did not consider responses from all participants, because some of them did not pass the attention checks. We marked a submission as valid only if the participant passed at least 3 of the 4 attention checks. The attention check participants failed the most was the one after the first category of products: *I got a lot of recommendations for skates on the previous page.* It seems as if participants did not understand the question properly, because 24 of all 70 participants failed that question. This is why we decided to allow *neutral* answer to the step questionnaires attention check questions. Other attention checks were strict; we did not consider *neutral* as a correct answer.

The process described above invalidates the submissions of 7 participants, bringing the number of valid participants to 63. When analyzing the data, we noticed that a lot of participants spent less time selecting a product in their second category. When we compared the time spent in the first category ($M = 313.564, SD = 168.113$) in seconds with the time spent in the second category ($M = 231.717, SD = 148.101$) by conducting a paired t-test, we obtained the result $t(52) = 3.902, p = 0.0003$, which is statistically significant. We also decided to filter out the submissions in which a participant spent less than one minute in at least one category.

The final number of valid participants dropped to 49 after the latest filter was applied. There were 51% male participants, 49% female, the majority (63%) was 21 to 30 years old and 16% of the participants were 31 to 40 years of age. 12% of participants did not answer the question about their age.

Let us analyze the responses directly connected to our hypotheses one by one before presenting other interesting results and the conclusion of this user study, we chose $\alpha = 0.05$ as the significance level in this study.

6.3.1 Hypothesis 1

Stopping criteria make users more confident that the product they selected was the best option for them than unseen statistics.

The goal of our application was to help maximizers when selecting a product from a catalog. Achieving high level of confidence in the final selection is a good step towards helping them, especially when the selection is done in a short amount of time.

Question 1 of the step questionnaire (*I am confident that the product I selected was the best possible option for me.*) was designed to test this hypothesis.

The responses to this question were overall positive, most participants agreed that they are confident that their selection was the best option for them, their responses are visualized in figure 6.1.

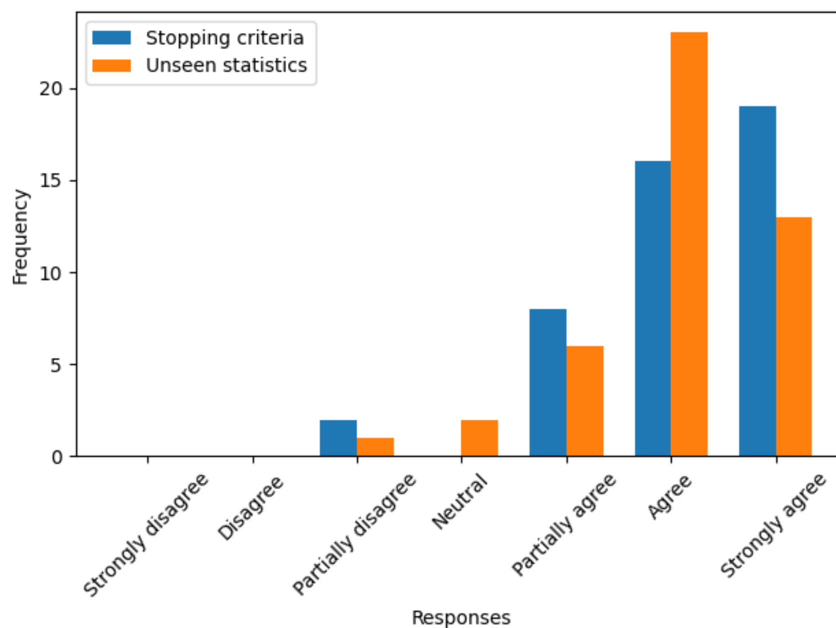

Figure 6.1: Responses to the question *I am confident that the product I selected was the best possible option for me.* of all participants.

Because we were comparing two different UI variants by using a single group of participants, we conducted a paired t-test. We transformed the likert responses to an integer from the interval $[-3, 3]$, where *strongly disagree* was represented as -3, *neutral* as 0 and *strongly agree* as 3.

The analysis of our data compared the scores for stopping criteria ($M = 2.111, SD = 1.005$) and unseen statistics ($M = 2.000, SD = 0.905$). The t-test did not reveal a statistically significant between the two variant with result $t(44) = 0.696, p = 0.490$. This result suggest that the differences in scores is likely not caused by true preference of one UI variant over another, but rather by random variation in the data.

6.3.2 Hypothesis 2

It is easier to find the final product with stopping criteria than with unseen statistics.

We assumed that the more complex model of customer preferences would be more suitable for finding appropriate candidates; let us analyze the data to check whether our assumption was correct.

We formulated question 2 of the step questionnaire (*The final product was easy to find.*) to gather data related to this hypothesis.

The responses were generally positive, but the data surprised us because participants considered finding the final product with unseen statistics easier. Maybe the simplicity of the rules generated in unseen statistics allows higher level of exploration, which persuaded participants to provide these responses, which can be seen in figure 6.2.

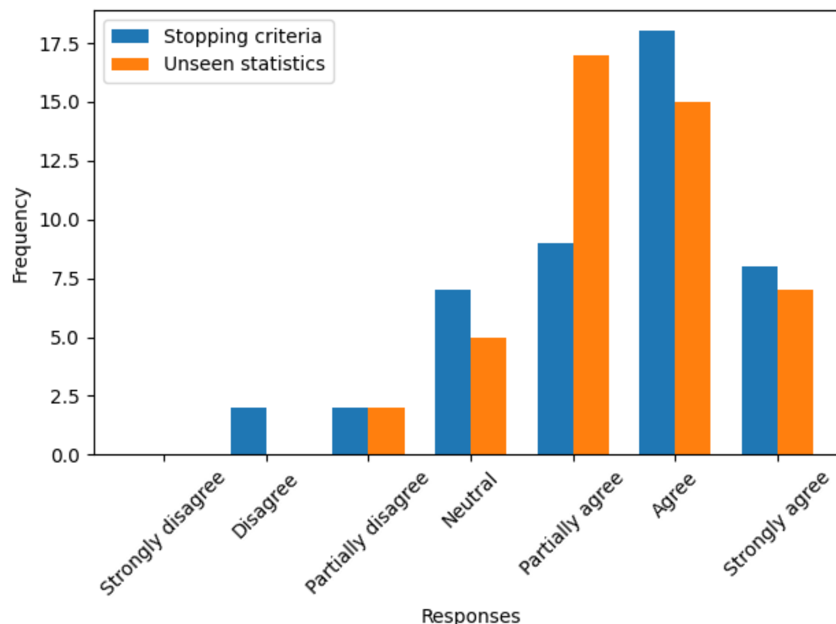

Figure 6.2: Responses to the question *The final product was easy to find.* of all participants.

We also performed a paired t-test to check whether the result was statistically significant. We compared the scores for the stopping criteria ($M = 1.370, SD = 1.306$) and unseen statistics ($M = 1.435, SD = 1.025$) and obtained statistically insignificant results from the test: $t(45) = -0.308, p = 0.759$, so we cannot draw any conclusions from this test alone.

6.3.3 Hypothesis 3

Stopping criteria is better for telling users when to stop searching for more products than unseen statistics.

This hypothesis was aimed at testing the interpretation of the information presented by the application as cut-off alerts. We expected stopping criteria to be interpreted more likely as cut-off alerts than unseen statistics.

Question 3 of the step questionnaire (*The system helped me decide when to stop searching for better products.*) was supposed to collect the data from the participants.

We collected generally positive answers, the participants stated that both UI variants helped them stop looking for more potential candidates, these answers are presented in figure 6.3.

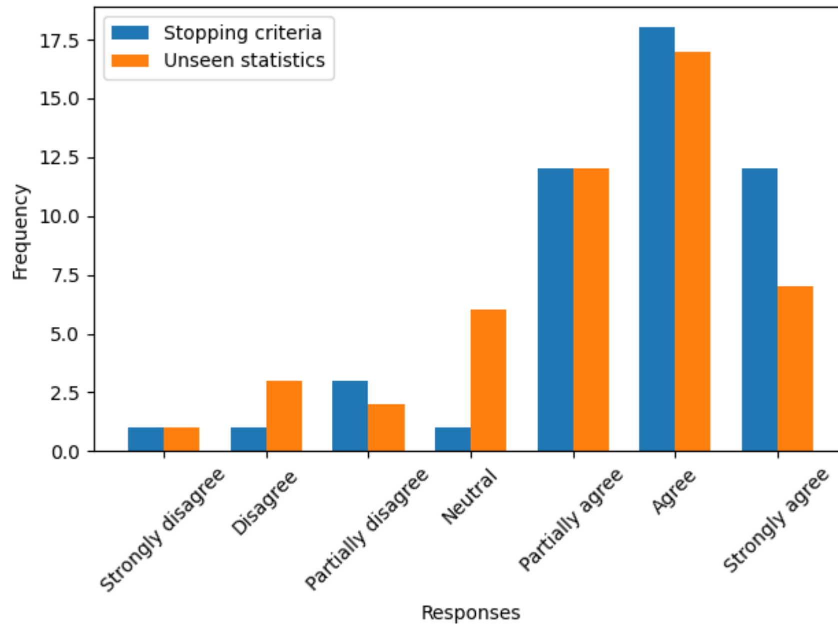

Figure 6.3: Responses to the question *The system helped me decide when to stop searching for better products.* of all participants.

The setup of the test of this hypothesis was the same as the previous hypotheses, so we conducted a paired t-test. We compared the scores for the stopping criteria ($M = 1.583, SD = 1.366$) and unseen statistics ($M = 1.167, SD = 1.463$) with the result $t(47) = 2.480, p = 0.017$, which is statistically significant. This result shows that participants interpret the stopping criteria more likely as cut-off alerts, telling them when to stop searching for better products.

6.3.4 Hypothesis 4

Users would more likely want to see stopping criteria on production systems.

Our assumption is that participants will see more value in stopping criteria than in unseen statistics and thus would rather like to see stopping criteria than unseen statistics on production system.

We added question 7 of the step questionnaire (*I would like to see [unseen statistics / stopping criteria] on production systems (other eshops).*) to collect data to test this hypothesis.

The collected data were also overall positive, which suggests that both UI variants could add value to current eshop solutions. Feedback from all participants is presented in figure 6.4.

We conducted a paired t-test with result $t(48) = 2.121, p = 0.039$ when comparing scores for stopping criteria ($M = 1.286, SD = 1.541$) and unseen statistics

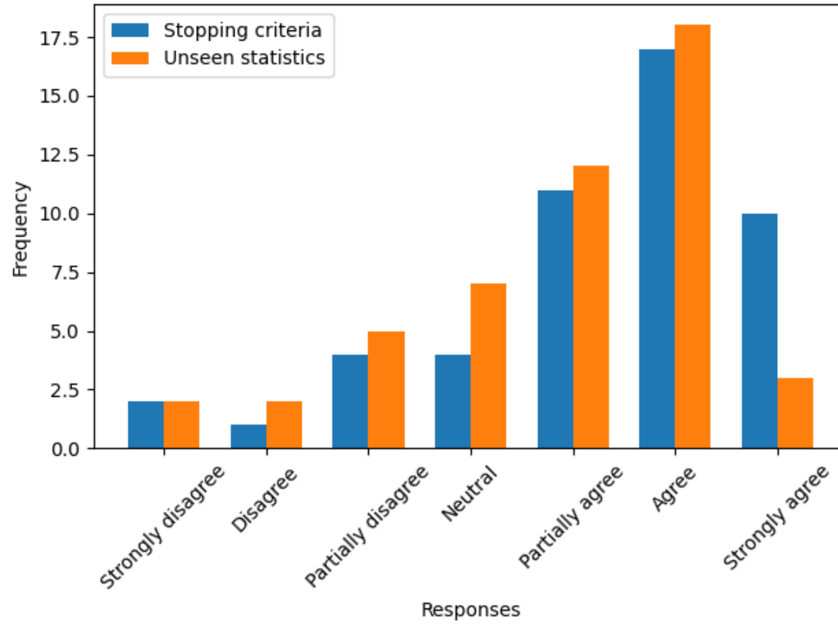

Figure 6.4: Responses to the question *I would like to see [unseen statistics / stopping criteria] on production systems (other eshops).* of all participants.

$(M = 0.857, SD = 1.486)$, this result is statistically significant. We see that participants would be more likely to see stopping criteria on production systems than unseen statistics, proving our hypothesis.

6.3.5 Hypothesis 5

Users that are familiar with the product domain find the stopping criteria easier to use.

This hypothesis suggests that we want to compare two groups of participants. We divided groups of participants into groups *familiar with stopping criteria* (SF) and *familiar with unseen statistics* UF according to which UI variant was used in the product domain the participant was more familiar with. For example, if a participant was familiar more with the domain of laptops than with the domain of mobile phones, he would be included in the *familiar with stopping criteria* if laptops were displayed with stopping criteria.

There were 16 participants in group SF and 12 in group UF, some of the participants were not included in any of these groups, because they were familiar with both categories the same.

Because we have two independent groups of participants, we needed to perform a different statistical test than paired t-test. We selected Fisher's exact test because we did not get enough data for a Chi-squared test.

We obtained a contingency table for the Fisher's exact test by computing the number of participants of each group that differentiated each group into two subgroups based on which UI variant they marked as easier to work with. We used question 5 from each step questionnaire (*Working with the system was easy.*) and then question 2 from the overall questionnaire (*Which UI variant was easier to understand?*). One subgroup preferred stopping criteria (labeled SE), the

other unseen statistics (labeled UE). Participants who did not prefer one variant over another were not considered.

We obtained two contingency tables: table 6.1 and table 6.2.

	SE	UE
SF	3	4
UF	4	2

Table 6.1: Contingency table for Fisher's exact test of hypothesis 5. SF is group of participants familiar more with stopping criteria domain, SF is group of participants familiar more with unseen statistics domain. SE is the group of participants considering stopping criteria easier to work with, UE consider unseen statistics easier, with respect to their response to question 5 of the step questionnaire.

	SE	UE
SF	7	6
UF	8	2

Table 6.2: Contingency table for Fisher's exact test of hypothesis 5. SF is group of participants familiar more with stopping criteria domain, SF is group of participants familiar more with unseen statistics domain. SE is the group of participants considering stopping criteria easier to work with, UE consider unseen statistics easier, with respect to their response to question 2 of the overall questionnaire.

Both contingency tables look interesting; it looks like participants less familiar with stopping criteria domain considered stopping criteria easier to work with, which goes against our assumption, but the p-value of the first test was $p = 0.592$ and the p-value of the second test was $p = 0.379$, which are not statistically significant.

6.3.6 Comparison of UI variants

We obtained interesting results when considering other questions that compare the two UI variants, and we will present the statistically significant results below.

Usefulness

We asked participants whether a given UI variant was useful to them (question 6 of the step questionnaire), and we conducted a paired t-test comparing scores for stopping criteria ($M = 1.286, SD = 1.541$) and unseen statistics ($M = 0.612, SD = 1.777$), the result was $t(48) = 3.049, p = 0.004$, which is statistically significant for our selected significance level $\alpha = 0.05$, showing that participants consider stopping criteria more useful than unseen statistics.

The general usefulness was high from the data we obtained. We visualized the responses of the respondents in figure 6.5.

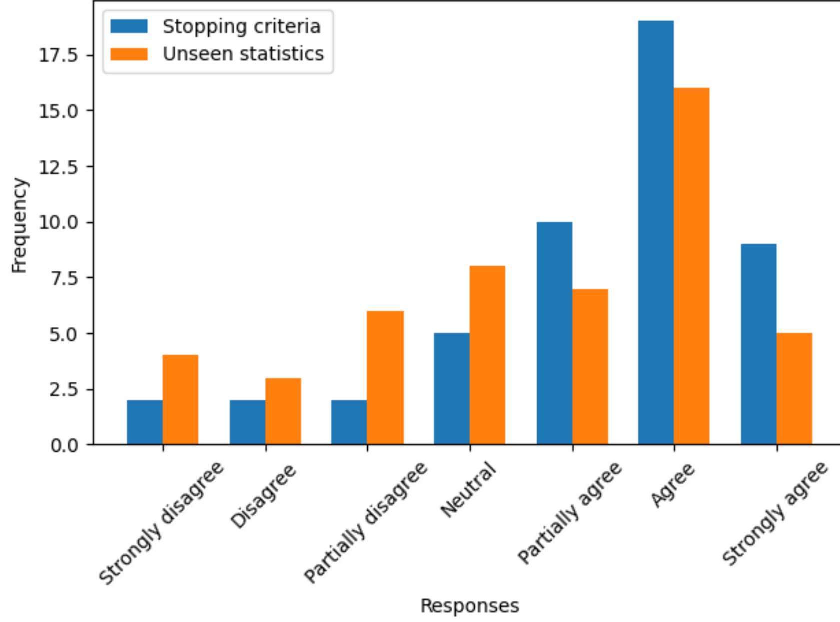

Figure 6.5: Responses to the question *I found [unseen statistics / stopping criteria] useful.* of all participants.

Helpfulness

We asked participants to select the UI variant that was more helpful in quickly finding good candidates, the responses of all participants are visualized in figure 6.6.

We performed a one-sample t-test, whether the observed preferences differ significantly from a balanced preferences. The responses were transformed into integers in an interval $[-3, 3]$, where -3 represented the response *definitely unseen statistics*, 0 *both were equal* and 3 represented *definitely stopping criteria*. We obtained data with mean $M = 0.854$ and standard deviation $SD = 1.726$ that produce a result $t(40) = 3.167$ with p-value $p = 0.003$, which is statistically significant and shows that participants consider stopping criteria better to find good candidates quickly.

6.3.7 Open questions

We analyzed the open-end questions using a publicly available LLM model, ChatGPT. Let us present the outcome of this analysis.

Considering the question *What makes one UI variant better than the other?*, the recurring topic of the answers included "easy to read and understand", "visual aspect of stopping criteria", or "the amount of information displayed". Some of the respondents even included the name of the UI variant they were describing, and several participants stated that they prefer unseen statistics, because the visualization is better than the one of stopping criteria (sliders show the range where the values lie, while stopping criteria only show a textual representation of the interval). Some answers also mentioned that stopping criteria are (in the author's point of view) more suitable for technically advanced users, while unseen

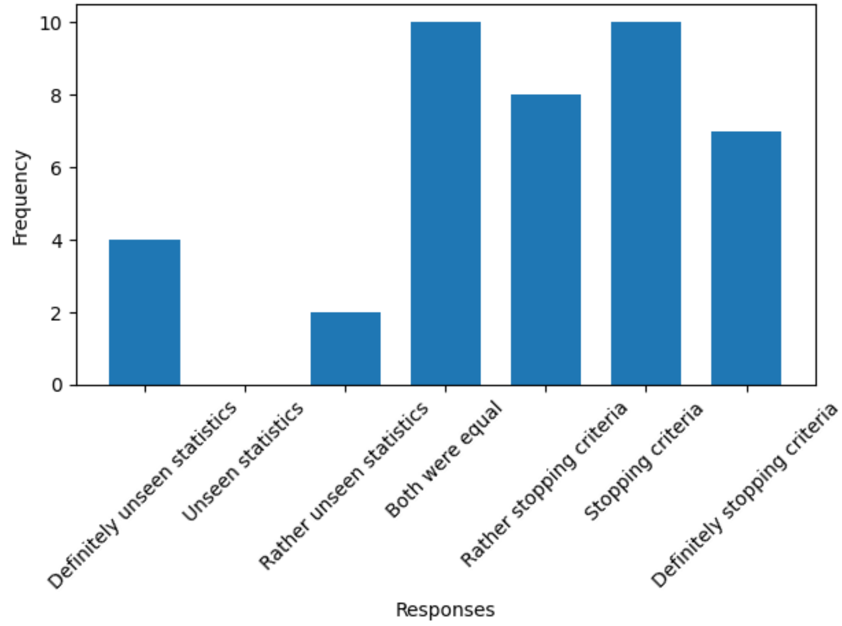

Figure 6.6: Responses to the question *Which UI variant was more helpful in finding good candidates quickly?* of all participants.

statistics are easier to understand for users without a strong technical background.

We wanted to support our claim that content-rich domains, especially the ones with high-risk, high-reward decisions, would be suitable for the type of UI we implemented. The respondents supported this claim by responding with electronics, cars, or smart home devices the most. Several respondents mentioned clothing, grocery stores, and even pets.

7. Future work

The goal of this thesis is to provide an alternative system to the traditional eshops. We believe a system with similar properties is missing in the ecommerce environment, but the proposed solution should be improved to become production-ready. However, the current state of the application meets our goals and expectations we had at the beginning.

7.1 User interface

The proposed system was developed without any collaboration with a UI or UX expert, which was noticed by several participants of the user study. They preferred visual aspects of unseen statistics over stopping criteria. We wanted to make the stopping criteria as good as possible, but we lack expertise in some areas.

We had several ideas how to improve the UI ourselves, but implementing these should be supported by a user study or A/B testing. We did not have resources and motivation to conduct these experiments, which is why we leave *improvement of the UI* as a part of this chapter.

7.2 AI models

The recommender system implemented with the proposed solution is purely content-based. This type of system was selected because of the dataset that was used in this thesis, it did not contain a lot of user feedback data.

Also, the implemented recommender systems are very simple. Implementing support for all attribute types such as lists or continuous numerical attributes could improve its performance and the satisfaction of customers.

Explanations generator could probably use some improvement even more. It would be nice to have an explanations generator capable of explaining a complex product domain to a customer who is not familiar with it. Enhancing explanations with a chatbot is one of our ideas we would like to explore.

7.3 User study

The results of our user study showed great potential of our solution, but there is still a lot of questions needed to be answered before considering deploying a similar system to production.

The layout of the application we proposed differs from traditional eshops, but we did not compare our solution to the traditional layout. The responses to the question *I would like to see [unseen statistics / stopping criteria] on production systems (other eshops).* seem promising, but we would need more data to fine-tune the details of the deployment.

Conclusion

As mentioned in section 6.1, it is apparent that we solved the problems with the lack of storage of candidate products and explicit feedback data, so we will focus on the last component: alerting customers to stop looking for more products.

The results of the user study were generally strongly positive and even though we did not get statistically significant results for all of our hypotheses, we obtained interesting information when analyzing the data.

The level of confidence in selecting the best product from the catalog was generally positive, but did not depend on the current UI variant.

It is interesting that we did not get a statistically significant result when comparing the responses to the question *The final product was easy to find.*, but the responses to the question *Which UI variant was more helpful in finding good candidates quickly?* generated results with strong statistical significance showing a strong preference for stopping criteria. It seems that filtering products using more complex (and thus accurate) rules is better when looking for relevant products, even if the level of exploration of stopping criteria is lower than the one of unseen statistics.

We were content to see that the participants supported our hypotheses concerning interpretability as cut-off alerts and that the participants would like to see this type of UI in production systems. This shows that we solved the problem we defined in chapter 1 and that there is potential to improve and deploy a similar solution in the real world.

The data did not confirm our last hypothesis, which can be considered a promising result. We thought that the stopping criteria would be preferred by participants familiar with the domain, because they are more complicated than the unseen statistics, but the fact that there was no clear preference shows that the UI is not too complicated for participants unfamiliar with the domain. It looks more like participants unfamiliar with the product domain preferred the stopping criteria more than unseen statistics, but we did not obtain enough data to draw reasonable conclusions. We believe this is caused by the complexity of the rules that provide useful information to the participants and explain more about the product domain.

The rest of the results showed that participants consider the stopping criteria to be more useful. It would be nice to analyze what exactly makes them useful and how. The open-end questions confirmed our assumption of the domains our solution would be suitable for.

The data we obtained in the user study support not only that our application solved the problem summarized in section 1.3.5, but also support the definition of the problem itself and the identification if the domain where the problem is the most noticeable.

Bibliography

Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.

Krisztian Balog, Filip Radlinski, and Shushan Arakelyan. Transparent, scrutable and explainable user models for personalized recommendation. pages 265–274, 07 2019. ISBN 978-1-4503-6172-9. doi: 10.1145/3331184.3331211.

Pablo Castells, Neil Hurley, and Saúl Vargas. *Novelty and Diversity in Recommender Systems*, pages 603–646. Springer US, New York, NY, 2022. ISBN 978-1-0716-2197-4. doi: 10.1007/978-1-0716-2197-4_16. URL `https://doi.org/10.1007/978-1-0716-2197-4_16`.

Kateryna Cherniak. BEST Chatbot Statistics for 2024 — Master of Code Global — masterofcode.com. `https://masterofcode.com/blog/chatbot-statistics`, 2024. [Accessed 09-07-2024].

Aigin Karimzadeh, Amir Zakery, Mohammadreza Mohammadi, and Ali Yavari. An explainable machine learning-based approach for analyzing customers' online data to identify the importance of product attributes, 2024. URL `https://arxiv.org/abs/2402.05949`.

Supriyo Mandal and Abyayananda Maiti. Explicit feedbacks meet with implicit feedbacks : A combined approach for recommendation system, 2018. URL `https://arxiv.org/abs/1810.12770`.

Emilio Moyano-Díaz and Rodolfo Mendoza-Llanos. Yes! maximizers maximize almost everything: The decision-making style is consistent in different decision domains. *Frontiers in Psychology*, 12, 2021. ISSN 1664-1078. doi: 10.3389/fpsyg.2021.663064. URL `https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2021.663064`.

John O'Donovan and Barry Smyth. Trust in recommender systems. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, IUI '05, page 167–174, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581138946. doi: 10.1145/1040830.1040870. URL `https://doi.org/10.1145/1040830.1040870`.

Francesco Ricci, Lior Rokach, and Bracha Shapira. *Recommender Systems: Techniques, Applications, and Challenges*, pages 1–35. Springer US, New York, NY, 2022. ISBN 978-1-0716-2197-4. doi: 10.1007/978-1-0716-2197-4_1. URL `https://doi.org/10.1007/978-1-0716-2197-4_1`.

Barry Schwartz. The paradox of choice: Why more is less. 2004.

Barry Schwartz, Andrew Ward, John Monterosso, Sonja Lyubomirsky, Katherine White, and Darrin R Lehman. Maximizing versus satisficing: happiness is a matter of choice. *Journal of personality and social psychology*, 83(5):1178, 2002.