NTNU
Norwegian University of
Science and Technology

Instant, Personalized Search Recommendation

Juul Arthur R Rudihagen

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Herindrasana Ramampiaro, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Personalization has proven to be a useful method in helping users find relevant items and documents. More and more digital stores implement some form of personalization to help users find products. Even though there are a lot of work in the field of personalization and recommender systems, search engines are only personalized to a limited degree. In this work, we investigate how to develop a personalized search engine, which gives instant product suggestions to the user based on user preferences and queries. As part of this, we implement and test our methods based on Bayesian classification and Kullback-Leibler divergence. In addition, we compare these methods with the baseline methods used today. Our experiments and user-based evaluation show promising results with respect to the relevance of the personalized suggestions and interactiveness. Overall, this research found that our approach is able to retrieve personalized suggestions instantly, even for products not directly containing typed terms. We expect our method to be useful for digital stores, in general.

Sammendrag

Personalisering har vist seg å være et effektivt verktøy for å hjelpe brukere å finne relevante produkter og dokementer. Flere og flere digitale butikker implementerer personaliserte løsninger for å hjelpe brukere å finne produkter. Selv om det foregår mye arbeid innen personalisering og anbefalingssystemer, er personaliseringen av søkemotorer begrenset. I dette arbeidet, utforsker vi hvordan man kan utvikle en personalisert søkemotor, som gir umiddelbare produktanbefalinger basert på brukerpreferenaser og brukerens spørring. Metoder basert på bayesiansk klassfisering og Kullback-Leiblier divergens er implementert og testet. Som del av dette, sammenligner vi metodene mot "baseline" metoder som er brukt i dag. Den kvantitative testen som ble utført viser lovende resultater med tanke på hvor relevant anbefalingene er, og hvor hyppig de blir brukt. Denne forskningen fant at søkemotoren utviklet, er i stand til å finne personaliserte anbefalinger umiddelbart, selv for produkter som ikke direkte inneholder termene fra brukeren. Vi forventer at metoden også vil være nyttig i digitale butikker generelt.

Acknowledgment

First and foremost I want to thank my advisor professor Heri Ramampiaro for his help forming the research question. I will also thank for guiding and motivating me throughout this work. Without you, this thesis would never exist.

Lastly, I want to thank my co-students at the office, Hans Kristian Henriksen, Kristine Steine and Tibor Vuković, for keeping up the morale and making excellent coffee in moments when the work felt overwhelming and morale was low.

Saving the most important for last, I want to thank my parents and sisters, for all their support and kind words throughout this period. Especially I want to thank my older sister for reading through the thesis and correcting mistakes.

I want to thank my fiancée for sticking out with me and supporting me all the way, even though there has been a lot of long nights working on this thesis.

J.A.R.R

Contents

Abstract **i**

Sammendrag **iii**

Acknowledgment **v**

Contents **vii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Problem Specification 2
1.3 Scope and limitations 3
1.4 Structure of the Report 4

2 Background Theory and Related Work **5**
2.1 Related Work 5
2.2 To Personalize or Not? 6
2.3 Gathering User Information 8
2.4 Importance of User Interface in Personalized Web Stores . . . 9
2.5 Status for Personalization in App Stores 10
2.6 Personalized Query Engines 11
2.6.1 Person-level Re-ranking 11
2.6.2 Group-level Re-ranking 12
2.7 General Overview of Content-based Recommender Systems . . 13
2.8 Information Retrieval Methods 14
2.8.1 Boolean Model 15
2.8.2 Term Frequency/Inverse Document Frequency 15

2.8.3 Normalization . . . 16
2.8.4 Vector Space Model . . . 17
2.8.5 Query Expansions . . . 17
2.9 Classification . . . 18
2.9.1 Nearest Neighbours . . . 18
2.9.2 Bayesian Classifiers . . . 19
2.9.3 Support Vector Machine . . . 20
2.9.4 Random Forest . . . 22

3 Approach **25**
3.1 Requirement Analysis . . . 25
3.1.1 Criteria . . . 25
3.1.2 Findings from Pre-study . . . 27
3.2 Design . . . 28
3.2.1 Overall Design . . . 28
3.2.2 Content Analyser . . . 30
3.2.3 Profile Learner . . . 31
3.2.4 Search and Filtering Component . . . 33
3.3 User Interface . . . 38
3.4 Initial Testing . . . 41
3.4.1 Search in Classification . . . 42
3.4.2 Classification in Search . . . 42
3.4.3 Term Reweighing . . . 42
3.5 Test for Optimization . . . 43
3.5.1 Optimizing App Information . . . 43
3.5.2 Optimizing Suggestion . . . 43
3.5.3 Optimizing Suggestion Retrieval Time . . . 45

4 Experimental Setup **47**
4.1 Achieving Anonymity . . . 47
4.2 Dataset . . . 48
4.3 Test Requirements . . . 49
4.4 Test Flow . . . 51
4.4.1 Building a User Profile . . . 53
4.4.2 Task 1 and 2 . . . 54
4.4.3 Task 3 and 4 . . . 55
4.4.4 Task 5 . . . 57
4.4.5 Task 6 and 7 . . . 58
4.4.6 Task 8 . . . 58
4.5 Gathering Information . . . 60
4.5.1 Binary Versus 5-Points Likert Scale . . . 60

4.5.2 Tracking User Behaviour 61

5 Results **63**
5.1 Population 63
5.2 Search Comparison Results 65
5.3 Suggestion Results 69
5.4 User Feedback 72

6 Discussion **77**
6.1 Data Basis 77
6.2 Comparing Search Results 78
6.3 Comparing Suggestion Results 79
6.4 Suggestion Results Grouped by Tasks 80
6.5 Analysing User Feedback 81
6.6 Evaluation of the Work 81
6.7 Limitations 83

7 Conclusion and Future work **85**
7.1 Conclusion 85
7.2 Further Work 86

Bibliography **89**

Appendices **93**

A Initial Testing **95**

B Test Results from Subgroups **99**

C Pseudocode **103**

List of Figures

2.1 Recommended apps by Google Play 10
2.2 Basic overview of a typical recommender system 14
2.3 kNN used to find the class of the diamond by comparing it to the 3 or 5 closest neighbours. We see that different choices of k can affect the result of the classification. 18
2.4 Example of how Support Vector Machine (SVM) works in a two dimensional plane. We see the separation lines in the figure and their margins. In this case A is a better separation line than B. 21
2.5 Simple illustration of a decision tree 22

3.1 A general overview of the system 30
3.2 Sequence diagram showing the sequence during popular search 34
3.3 Sequence diagram showing the sequence during search in classification 35
3.4 Sequence diagram showing the sequence during classification in search 36
3.5 Sequence diagram showing the sequence during term Reweighing and search 37
3.6 The user interface for our instant recommendations 38
3.7 The user interface Google Play's query suggestion 39
3.8 The result page for when the user performs a search 40
3.9 Screenshots for app toggled 41

4.1 Register page with random username 52
4.2 Login page 53
4.3 App presented to build user profile 54
4.4 Task 1 55
4.5 Search page showing multiple apps, the participant will then go though each app and tell their preference for that app . . . 56
4.6 Task 3 57

4.7 Task 5 57
4.8 Task 6 58
4.9 Task 8 59

5.1 Age of participants 64
5.2 Gender and experience with app stores from before 64
5.3 Number of results rated as relevant, irrelevant and neither by the user 66
5.4 Fraction of results rated as relevant, irrelevant and neither by the users 67
5.5 MRR for the different search methods 69
5.6 MRR for the different suggestion methods 71
5.7 Answers to how fast the users felt the suggestion showed up . 73
5.8 Answers to how often the suggestions made the participants reformulate their query 74
5.9 Answers to how helpful the participants felt the suggestions were 75
5.10 Answers to how the participants felt the suggestions helped them the most 76

List of Tables

3.1 Criteria for algorithm 27
3.2 Algorithms compared to criteria 27

4.1 Criteria for algorithm 49

5.1 Age of participants 65
5.2 Gender of participants 65
5.3 Participants' experience with app stores from before 65
5.4 Comparisons between search methods 66
5.5 Fraction of results retrieved where relevant 68
5.6 MRR for search results 69
5.7 Comparison between suggestion methods 70
5.8 MRR of suggestions 71
5.9 Clicked and relevant suggestions for the different tasks 72
5.10 How quick the users felt the suggestions were retrieved 73
5.11 How often users felt they reformulated queries because of the suggestions retrieved 74
5.12 How helpful the suggestions were 75
5.13 Answers to how the participants felt the suggestions helped them the most 76

A.1 Differences in apps liked and disliked based on whether the user likes or dislikes the category 95
A.2 Results from initial testing of features to use for classification . 96
A.3 Results from initial testing of adding or subtracting points based on classification 97

B.1 Results by grouping participants by age 100
B.2 Results when grouping participants by gender 101
B.3 Results from initial testing of features to use for classification . 102

Abbreviations

CF Collaborative Filtering. 18, 19, 22

IDF Inverse Document Frequency. 16

KL Kullback-Leibler divergence. 17, 31, 32, 36, 37, 43, 44, 86

kNN k-Nearest Neighbours. 18, 19, 27

MRR Mean reciprocal Rank. xiii, 68–71, 79

SVM Support Vector Machine. xi, 20, 21, 27, 28, 86

TF Term Frequency. 16

TF/IDF Term Frequency/Inverse Document Frequency. 15, 17, 43

VSM Vector Space Model. 17

Chapter 1

Introduction

1.1 Background and Motivation

Search is an important part of our web experience today. There are a lot of information on the web, and search has proved to be a good tool to find that information [1]. Search engines are used in navigating to different pages on the web, as well as finding specific products or pieces of information in these given pages.

There are several problems that make it difficult for search engines to retrieve relevant information to the users. One problem is that most people present short, imprecise and ambiguous queries, which leads to irrelevant results being presented to the users [2]. Often users don't really know exactly what they want beforehand, and need to try multiple queries before they find relevant information. Because of this issue, there are a lot of work in the field of helping users to find the information that they are looking for. Query suggestion and query completions are such methods, which helps users formulate better queries. These methods have proven useful in many cases [3]. Another problem is that different users can look for different things even thought they type the same query. Two users searching for apple can look for two different things. One might look for the fruit, while another might look for the company Apple[1]. Therefore, work is being done on personalizing both query suggestions and query completions. These kinds of personalization techniques are implemented by larger companies like Google[2] and Facebook[3].

[1] http://www.apple.com/
[2] www.google.com
[3] www.facebook.com

The motivation behind this thesis began after experiencing that a lot of systems today utilize personalization techniques for their users, while search engines are only personalized to a limited degree. When they offer personalization, the user often has to perform the initial search first. The process lead to multiple steps before the users are able to find the information they are looking for. To avoid this extra step, we will try to give personalized instant suggestions as users are typing the query. This is meant to help them quicker find what they are after, based on information about the users. A lot of the research on personalization is related to free text search, but we will investigate whether personalized search would give better results with more structured data.

This thesis present, a personalized search which is a combination of a search engine and a recommender system. Users will get instant recommendations based on their user profile, while they are writing their query. As a use-case, we investigate personalized search in app stores, as these have a lot of structured data, many products to search through and because this is a fast growing domain where a lot of users need help to find products suited for them. This is also a domain where search is not being utilized to more than finding specific pre-known items, and where novelty is low [4]. Most query suggestions today only suggest items or documents containing the search terms provided by the user. This search engine would be able to give instant personalized recommendations on items, beyond those containing the search terms, as opposed to search engines today which usually suggest products only directly containing the query.

1.2 Problem Specification

The following are the research questions this thesis tries to answer:

RQ: **How can we create a search engine providing instant personalized recommendations, based on user preferences and query typed?**

RQ is the main research question we try to answer. We have also formed 4 research question we will answer to evaluate the validity of instant personalized suggestions.

*RQ*1: **How will the relevance of the retrieved results from our personalized methods compare to traditional search methods used today?**

*RQ*2**: How will users use instant personalized search?**
*RQ*3**: Will the proposed personalized search help users quicker find relevant products in digital stores?**
*RQ*4**: How can we retrieve search recommendations in close to 100ms which is the requirement for instant suggestions?**

The research will not try to make a system that should be used directly in digital stores, but test the validity of different methods to see if they should be developed further.

1.3 Scope and limitations

Because of time limitations when working on this master thesis, we need to set a scope for the research. Giving suggestions instantly the way described has, to our knowledge, not been done before. We will therefore focus on investigating whether this is a method that could be effective for users of digital stores. This research will not set out to make a complete and optimal system, following the requirements of instant recommendation, but rather create a prototype showing the concept for the search suggestions. Testing will show if this kind of suggestions are beneficial for users, and should be investigated further. The methods used for this instant recommendations will not be found in this research, as this is done in a previous work conducted [5].

There are also limitations to our implementation and testing that needs to be taken into consideration. One limitation is the data available. Our search engine will only have content-based information to work with. This is because we do not have any data containing information that can be used for collaborative filtering, as well as we do not have the time to gather data that can be used for this information. We will also only be able to get explicit information from the user about app preferences, and not implicit information like play time and other metrics used in some stores today [4]. This is because the users will not actually download or use these apps.

Another limitation is the tests we can perform to evaluate the system. We will not be able to gather user logs or histories, neither will we be able to test the system with multiple users over a longer time period. This will make it challenging to apply some of the methods used in providing recommendations today. It can also be hard to evaluate this system compared to those used in existing systems.

1.4 Structure of the Report

Chapter 2 Background Theory and Related Work: An introduction to related work in personalization and search. Also explains methods evaluated and used in creating the instant personalized search engine.

Chapter 3 Approach: Explains the creation of the instant personalized search engine, and how the design works.

Chapter 4 Experimental Setup: Presents how the quantitative test is created and designed.

Chapter 5 Results: Presents the data gathered from the quantitative test conducted.

Chapter 6 Discussion: A discussion of the results gathered from the quantitative test.

Chapter 7 Conclusion and Future work: Presents the conclusion from the research, and suggests further work based on the research done.

Chapter 2

Background Theory and Related Work

In this chapter, we look into related work to instant personalized search, as well as investigate relevant theory for building this kind of search. Section 2.1 looks into related work, investigating similarities and differences. Section 2.2 discuss the benefits and disadvantages of personalizing web solutions in general. Section 2.3 investigate how to gather and use information about users. Section 2.4 looks into the importance of user interface when creating a personalized solution for the user. In section 2.5 we investigate the status on how current app stores are personalized for their users. Section 2.6 investigate methods for personalizing queries. Section 2.7 gives an overview of how typical content-based recommender systems are build today. Section 2.8 explains ways to implement search engines. The chapter ends with section 2.9 investigating different classification methods considered for the instant personalized search engine.

2.1 Related Work

When looking for related work, a combination of recommender systems, personalization and search engines are areas of research related to some of the concepts in this work. Hu [6] has created a personalized search, where they learn a user profile based on bookmarks. They gather terms from these bookmarks and use them to re-weight the search query. This work researches a lot of the same concept as the proposed solution. Some differences are that they do work in a different domain with text documents on the web, and

with less structured data. They also don't give instant feedback to the user, but they show that personalizing their search engine provides better precision and recall than what they got at Google's engine.

Shi and Ali [4] tries to help users find apps in app stores by creating top lists of personalized apps. They point to the problems of today's solution by presenting only the most popular apps. Only promoting these apps causes almost all their users to play a very limited number of apps, while the majority of apps on these stores have almost no users. They also show that content-based recommendation works better for many cases in the app store domains than collaborative filtering. Since the data set in this domains is very sparse.

Amazon [7] and Netflix[1] [8] are some large companies that has a lot of research on personalization. They work on recommender system and use a large variety of techniques in order to personalize the experience for their users. They present items that are related, liked by friends or recommended by other. Despite this, their search engines are only personalized to a limited degree. Their suggestions are provided after the search is completed and not instantly.

Matthijs and Radlinki [9] creates a personalized search engine for the web, by using long term user history, which proved to be a significant improvement from Google's standard search. They use a re-ranking technique based on the user profile to give personalized results for each user. Dou et al. [10] perform a large evaluation of different re-ranking techniques for personalized search. Where they describe different methods of user-level and group-level re-ranking methods. Which are based both on user preferences and user history.

All of these systems have some form of suggestions or personalization, like our proposed personalized search engine. The difference is that our design will present instant personalized suggestions, not necessarily containing the query typed. Most of these methods either completes the query, only shows product directly containing the query or presents recommendations before or after the user has searched for products. Where we try to give users recommendations instantly while they write the query, more based on their preferences than previous methods.

2.2 To Personalize or Not?

One question that is important to answer before implementing a personalized method is whether it's worth it. This is a question which has been researched,

[1] www.netflix.com

but also has a lot of subjective opinions. An article that describes this question thoroughly is the research by Jaime Teevan [11]. The article states that personalized queries perform well for ambiguous queries. The reason is that users often want different answers to the ambiguous queries, while well-formed precise queries often benefit from other methods, as the users usually want the same document. It is also stated that in some situation, personalized queries can perform worse than non-personalized when *"unreliable personal information swamps the effect of aggregate group information"*. Note that this article also researched on what queries can be considered ambiguous and states that it is a hard problem to find which queries are ambiguous and not.

Another thing to note when talking about personalization is anonymity. In order to give personalized results to users, it is required to obsess some information about them. On the other hand, this can compromise anonymity. Companies like Google and Facebook gathers a lot of information about their users, and some of it may be sensitive [12]. Therefore, It is important that the companies handle their information ethically, and that the users agree to provide this information. If a user searches a lot for a specific disease on Google, one could believe that this user suffers from the given disease. If insurance companies were to get a hold of this information, they could use it against that person. Because some users are skeptic when providing personal information, and since there have been incidences where information meant to be anonymous has been compromised, like the Sony hack in 2011 [13], there are some users who are not willing to provide information that can be used for personalization [14]. People often think of personalization as a way of offering more advertisements, like Facebook and Google's targeted ads.

Another method used instead of personalization, is using most popular results [8]. This method is the opposite of personalized results as it will always give the same result to the same query. As Jaime Teevan [11] states, this technique can seem to outperform personalized queries when the query provided is precise. Our personalized search engine should outperform a search engine only showing most popular results in order to be effective. Also, it has to be taken into consideration that using personal information in search engines can also increase the time needed to compute the results. This can also be too much of a drawback to implementing a personalized solution.

Even though personalization leads to more work and computational power, many companies still focus heavily on personalization. Some of these are Netflix and Amazon. Their experience is that it helps user to find what they like, and can also be helpful in cases where users are not sure of what they want beforehand. Another positive effect is that it can increase novelty, as

the users are not only presented with the most popular items, but the best suited for them.

2.3 Gathering User Information

In general, there are differences in how much information about their users the different digital stores gather and use to give personalized experiences. Some of this is due to design decision, others are due to limitations in their stores. There are generally two ways to gather information about users, with or without user interaction. A company that implements personalization techniques for their users is Netflix, they have described the different methods they use in Amatriain [8]. They use a combination of implicit and explicit user feedback. One could think that explicit feedback would give a very accurate picture of the user, as they themselves provide the information, but the article states that their experiments show that information gathered explicitly is noisy. Thus, the results get better with a hybrid solution of implicit and explicit user information.

Even though a lot of information is available, this does not mean that it is smart to use all of it. Twitter talks about this in the article about their architecture for query suggestion [15]. They have a strict requirement with low latency, and therefore don't have the time to use all the data they have available for their computations. They have limited themselves to use only two sources of data; tweets and search session. What information our system should use, would also have to be carefully chosen in order to be able to give quick results

Getjar [4] created their own model which they used for app suggestions. They extracted features from apps, and used these to create similarity matrices between apps. They showed that using these features extracted from apps they could recommend apps with better results than the Memory-based and PureSVD method they tested. This suggests that using app features to as information to use for suggestions, might work well in some cases.

2.4 Importance of User Interface in Personalized Web Stores

There are a lot that can be said about design of user interfaces, and how to design websites. A lot of this information is not considered relevant for this thesis, but what is interesting to see is how design of the personalized solution affects the user experience and the overall result. Kumar et al. [16] talk about the effect user interfaces have on the users, and specifically in personalizes solutions. The article suggests that there are differences in how different groups of people will perceive the user interface. Experienced users are only interested in the elements that help them with their tasks, while more inexperienced users will look at the whole site for information that can help them. They also claim that it is important for the user to have related information close to each other, as users don't like to scroll down or look around for relevant information. They claim there are some parts of digital stores which benefits from personalization and some parts which do not. All in all their research shows that for most people, personalizing the search engine can be beneficial. This is as long as the suggestions are accurate and understandable.

Since experienced users only focus on the part of the website they are working on, query suggestions should be in close proximity to where they insert the query. Multiple search engines, as well as Google's, show the suggestions directly below the query. Users also perceive elements in close proximity to relate to each other according to the Gestalt principles [17]. So a solution for the proposed personalized search, should make sure to show the user that the suggestions are related to the query presented.

In the research done by Kelly et al. [3], they tried to provide query suggestions in different ways by giving keywords or full sentences. The research suggests that it is beneficial for users to get suggestions in complete sentences instead of terms. The article states that the users did not want to generate a new query, and therefore users liked better when the query was generated for them. This is an interesting observation as term suggestion could seem to add or remove exactly the relevant information for the query, while query suggestions will sometimes add words which do not directly add useful information to the query. This research is positive for this thesis as the proposed solution will also help user quicker get the items they are after, instead of going extra steps to generate new queries.

2.5 Status for Personalization in App Stores

Since there are vast amounts of apps available, and many users are not sure of what they want when searching for apps, we see app stores personalizing more and more. The article by Shi and Ali [4] claims that back in 2012, Google Play only did personalization based on user locations and what devices they had. Today we can see them trying to give recommendations based on apps you have downloaded, as in figure 2.1, and apps your friends like. App Store by Apple does not seem to focus on personalization. At least not in a way that is visible to the users. Instead, they focus on selecting apps themselves to show as featured apps, and create a different list of apps they present to users. This would make sure that apps they consider good would be more exposed to the user, and that user themselves have to learn about less popular apps.

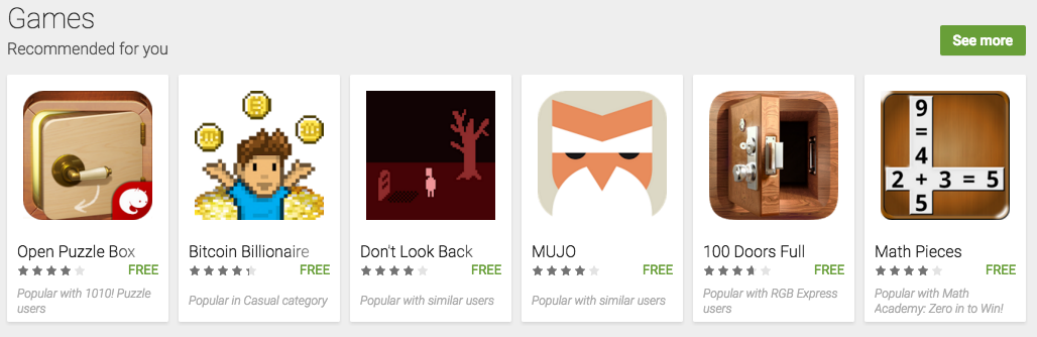

Figure 2.1: Recommended apps by Google Play

At Google I/O 2013 [18] it was released how search in Google Play worked. Google uses location to rank apps when searching. After testing different ways to replicate this, it seemed like changing the language used had no effect on the ordering of the apps. Logging in on different accounts did not seem to change the behavior of the search either. The only thing that had some minor effects, was changing the IP address to another country. This changed the ordering of the result, as well as giving other query suggestions. This seems to correspond to the information given at Google I/O.

The lack of personalization can be a disadvantage for specifically two reasons, as pointed out by Shi and Ali [4]. The first being that only users already knowing what apps they want benefits from showing most popular apps, while people exploring for new apps will probably not get apps that suit them. The other problem is that even though there is a lot of apps available in

the different stores, there are only a few of them which are actively used. In 2012 the top 1% of apps accounted for 58% of all usage [4]. This might make it harder for individual developers, who don't have the same budget as bigger companies, to reach out with their apps. They also claim "Search is also ineffective because we find that most users don't know what to search for. About 90% of search queries at GetJar are titles (or close variants) of popular apps, which means search currently is not being used as an effective tool to discover new apps.". This leads to believe that there should be more potential to be exploited for search engines in app stores [4].

2.6 Personalized Query Engines

In this section, we will see some methods used by current solutions in implementing personalized search. This will both provide useful information to how the method in this thesis could be implemented, and investigate if there exist similar solutions already. We will investigate the two main categories of methods used in personalized search, person-level re-ranking and group level re-ranking [10].

2.6.1 Person-level Re-ranking

Person-level re-ranking is based on individual users preferences. The system will learn user profiles to the different users, and use this information to re-rank the search performed. We will look deeper into two main methods used when re-ranking on person level, based on user history and based on user preferences.

Based on User History

The general thought behind this method is that web pages frequently clicked by the user, is probably more relevant to them than those that are seldom clicked. The personalized score for the user u and query q can be calculated by the formula:

$$S^{P-Clicks}(q,p,u) = \frac{|Clicks(q,p,u)|}{|Clicks(q,\dot{x},u)|+\beta}$$

In this equation $|Clicks(q, p, u)|$ is number of clicks for user u, on web page p with query q, and $|Clicks(q, \dot{x}, u)|$ is number of clicks in total by the user with the query q. β is the smoothing factor, and is used for normalization. A typical value for this could be 0.5.

Even though this gives a simple equation to calculate the personalized score for the user, it has the disadvantage that it can't give scores to new queries for the user. The fact that it can't give score to new queries makes it less interesting for the proposed algorithm in this thesis, as we want to help the user find new apps and not the ones he has already seen. The fact that user history is used, makes it hard to use for our test of the algorithm, as we will not have user histories to work with.

Based on User Interests

There are multiple ways to use and store user interests. One of the methods proposed by Dou et al. [10] is pre-defining certain features, store the users' interest in those features, and user them to re-weight the query. Each web page also has to store weighting in the pre-defined features that can be used together with the user profile. There are many ways to compare these feature vectors, but we will here show the general principle. One simple formula that can be used is:

$$S^{L-Profile}(q, p, u) = \frac{c_l(u) * c(p)}{||c_l(u)||||c(p)||}$$

Here $c_l(u)$ is the user profile and $c(p)$ is the category vector for the web page. How these features are found have a huge impact on the results, and as Dou et al. [10] states, it is also important to normalize the data to get accurate results. One positive part about this method over the one based on user history is that this method will work on queries never asked by the user before. Though there are some problems when categorizing and calculating the different feature vectors.

2.6.2 Group-level Re-ranking

When personalizing on a group level, the system tries to group up similar users. Then the system can use the combined user histories of the group in total to do query re-ranking. The method suggested by Dou et al. [10] is

to use K-Nearest Neighbour on the user profiles and find similarities by the formula:

$$Sim(u_1, u_2) = \frac{c_l(u_1) * c_l(u_2)}{||c_l(u_1)||||c_l(u_2)||}$$

One benefit of this method is in the way it gets more data to use in the re-ranking, as the system can use information on multiple users. One negative part of this algorithm is that it will not necessary be as specific for each individual user. This method does not gain a lot if it's not used with user histories, like for example when using simple re-ranking based on user interest as described in 2.6.1.

2.7 General Overview of Content-based Recommender Systems

As this thesis are looking for content-based methods for recommending apps according to a user query, we will first have a look at a basic overview of how content-based recommender systems work today. As stated by Dietmar Jannach [19], who has done a thorough study of state of the art content-based recommender systems, these systems generally consists of three main parts shown in figure 2.2:

- Content analyser - Structures the data for the other steps. The main concept is extracting features from the items and in order to represent them in a way that is efficient and meaningful for profile learning and filtering components. In many areas, this is a hard task to automatize, and therefore still requires interaction by humans with domain knowledge. This is usually required in the film domain, to add year, the cast, director etc.
- Profile learner - Is used to build user profiles by finding content the user likes or dislikes. Often times, a form of machine learning is used to build up these models based on user interaction or feedback. As mentioned earlier, this part of the system is not the focus of this thesis. How this part works, is still crucial for the system to be able to give accurate suggestions for users.
- Filtering component - Use the information from the content analyser and the profile learner, to find items relevant to the user. The result

can either be a ranked list, or a list of relevant items. Finding relevant items are done by doing similarity calculations between the items and the user profile. The slight difference from the personalized search in this thesis, and typical recommender systems, is that our search engine would have to combine a query as well as the user profile to filter items.

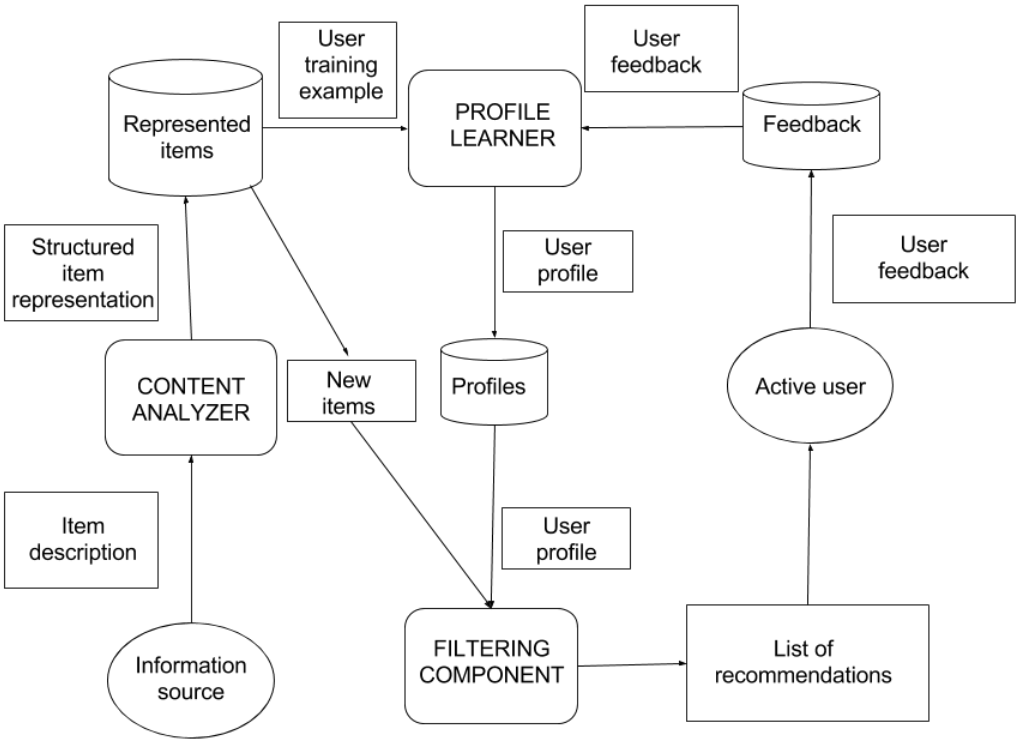

Figure 2.2: Basic overview of a typical recommender system

2.8 Information Retrieval Methods

There are multiple ways to implement search algorithms. We will look at some of the methods used by Elasticsearch[2] built on Lyceene[3], to get a picture on how one state of the art search systems is designed. The description of the system is gathered from Elasticsearch's website [4]. We will also look at some other techniques commonly used in search engines.

[2] https://www.elastic.co

[3] https://lucene.apache.org/

[4] https://www.elastic.co/guide/en/elasticsearch/guide/current/scoring-theory.html

2.8.1 Boolean Model

The Boolean model [20], is one of the simplest approaches when it comes to search. It uses logical operations such as AND, OR and NOT as conditions in the query to find all documents that match. A query like "A and B or C and not D" would match all documents that either contained A and B or contained C and not D. This is a simple and fast approach to finding documents. Even though it is not always the best technique for free text search, it can be used to filter documents that would not match the query anyway. By first filtering away all documents that do not match the logical operation, the system can use more computational heavy methods on the remaining documents.

In general, this method is regarded as simple and cannot perform advanced queries or rank the documents. This method has the benefit that it allows sorting of the documents by a given criteria, highlight occurrences of the keywords, and allow feedback to reformulate the query. Since this method is compositional, a query tree is defined, where the leaves correspond to the basic logical operations in the query.

These kinds of logical queries can be hard to use for users not trained in mathematics. There are some solutions who uses fussy logical operands, so that the search only need to fulfil some of the conditions. This way, the AND operator is stricter than an OR, but may not require all the conditions to be satisfied. It is also possible to use fussy keywords, where you allow the keywords to match partially, and may allow words to be similar. When using a partial match, using distance measures such as Levenshtein edit distance, is helpful. This is the distance preferred by Elasticsearch.

2.8.2 Term Frequency/Inverse Document Frequency

Term Frequency/Inverse Document Frequency (TF/IDF) [20] is the most popular term weighting scheme in information retrieval. And as the name implies, it is built up by the two concepts term frequency and inverse document frequency.

Term Frequency

Term Frequency (TF) [20] is simply a weight given to a term based on how often the term occurs in a given document. The thought behind weighting the

terms this way, is that a document containing a term often is more relevant than a document only mentioning the term once. There are multiple formulas used to compute the term weights, the one used in Elasticsearch is:

$$tf(t\ in\ d) = \sqrt{frequency}$$

Inverse Document Frequency

Inverse Document Frequency (IDF) [20] is used to find how rare a term is. The more times the term is present in all the documents in our collection, the lower the weight is. The thought behind it is that if a document includes one of the words from the query which is not common in the document set, it should count more towards the relevance of the document than the words usually present in documents. Words like "and" or "the" should not have the same weight as words like "koala" or "rose". These rare words often also carry more semantics than common words. There are multiple ways to calculate IDF, the formula used in Elasticsearch is:

$$idf(t) = 1 + log(numDocs/(docFreq + 1))$$

2.8.3 Normalization

The longer a document or field in a document is, the more likely it is to contain the words in the query [20]. Therefore, we often say that if a word is present in a document or field with few words, it carries more semantic than if it occurs in a document or field with several words. This method is used to balance documents, in the way that longer and shorter documents should have an equally chance of being ranked first. Even though longer documents have a higher probability of containing the words being searched for, and also having multiple instances of the word. The same normalization can be used for shorter fields so that for instance the title field gives a higher weight to terms than the body. The formula used in Elasticsearch is:

$$norm(d) = 1/\sqrt{numTerms}$$

2.8.4 Vector Space Model

Vector Space Model (VSM) [20] is a spatial representation of text documents. Each document is represented by a vector in an n-dimensional space. The number of dimension n, is determined by the number of terms in the given document collection. The weight of each term has to be calculated, and this can be done in a variety of ways. A common solution is using TF/IDF as described in the section 2.8.2. When comparing these documents to a query, one can simply create a similar vector in the n-dimensional space from the query and compare this vector to the vector of the documents. The search is then reduced to finding the vectors which are closest to the query.

There are some problems with VSM. One is that it can have problems with high dimensionality, since there will be a lot of dimensions with weights that are non-existent. Another problem is that it does not capture the semantics of a document. So if two different words have the same meaning, VSM will not be able to capture this.

2.8.5 Query Expansions

In order to help users find documents they are after, a technique called query expansion can be used. The idea is to find relevant terms from the initial search performed by the user, and use these terms to expand the query.

One method to find such terms is by using Kullback-Leibler divergence (KL) to score terms. This is a method proven efficient [21]. The thought behind the method is to analyse the term distribution in the top-k documents retrieved and the distribution of terms in the entire collection. This is used to maximize the divergence between the two. The terms with the highest score, are those contributing to the highest divergence, and are used for the query expansion. The following equation is used to calculate the KL-score:

$$KL = P_{Rel}(t) \times log[\frac{P_{Rel}(t)}{P_{Col}(t)}]$$

$P_{Rel}(t)$ is the probability that term t appears in the top-k documents, and can be calculated by finding the number of times the term occurs divided by the total number of terms in the top-k documents. $P_{Col}(t)$ is the probability that t appears in the whole collection, and can be calculated by finding the total number of times the term occurs in the collection divided by the total number of terms in the collection.

2.9 Classification

Classification is the task of assigning documents to one or more classes or labels. Classification techniques are often used in recommender systems to group the products or items. A common method is to classify the items in "liked" or "disliked" classes, where liked are the ones desirable for the user. Generally, we divide these methods into supervised or unsupervised methods. Where supervised means we have some labels that are known in advance, and use a training set to train up the system to classify correctly. In unsupervised classifications, the labels are not known in advance, and the task becomes to organize the elements based on the structure of the data.

2.9.1 Nearest Neighbours

One of the most common methods to use with Collaborative Filtering (CF) is the k-Nearest Neighbours (kNN) classifier [22]. kNN falls in the category of lazy learners. This means that it does not build a classification model in advance, but perform the classification process as new documents are added. The classification is based on the k-nearest neighbours to the document, where the distance has to be in a metric predefined for the space.

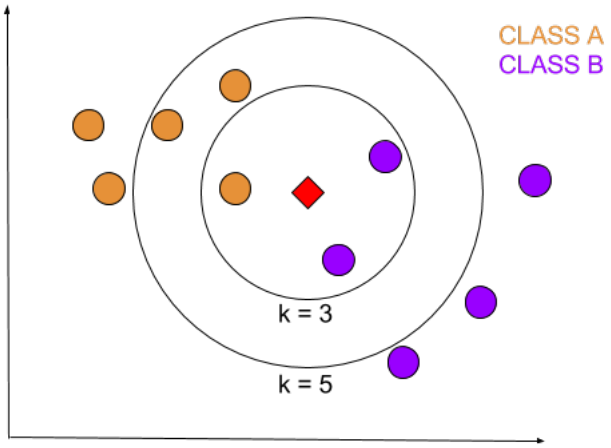

Figure 2.3: kNN used to find the class of the diamond by comparing it to the 3 or 5 closest neighbours. We see that different choices of k can affect the result of the classification.

This method works by storing training records with given labels as desirable or not desirable. When finding the classification of a new document, the method finds the k-closest documents and take the class of what the majority of these documents have.

kNN has some benefits being that it's conceptually simple and similar to the basic idea of CF, which is finding like-minded users or similar items. It is also one of the most common approaches to CF. Another positive thing is that it does not need to maintain a given model, as it is a lazy algorithm. There are also some negative aspects of this algorithm. One of them being that it's computational costly. Every time you add a new document, you have to calculate the similarity with all training documents. This can be partially avoided with the help of special purpose indexes, but it would still have a problem with performance. Another problem is finding a value for k which optimizes the classification process.

2.9.2 Bayesian Classifiers

Bayesian classifiers [22] use probabilistic methods to classify the different documents. The general principle is that you have a record with N attributes (A_1,A_2,...,A_N). Then you find the class C_k that maximizes the posterior probability of the class with data $P(C_k|A_1,A_2,...,A_N)$, and classify it with the class that maximizes the probability. The method is named after the Bayes theorem which is used when calculating probabilities and looks like this:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

A common Bayesian classifier is the Naive Bayesian Classifier, which assumes all attributes to be independent. This means that the presence or absence of any attribute is unrelated to the presence or absence of any other attribute.

An article that describes recommendation with Naive Bayesian Classifier is Miyahara and Pazzani [23]. They define two classes: "like" and "don't like", and uses two different models to classify. The first method is a Transformed Data Model, which assumes all features to be completely independent. Before they do this they use feature extraction as a preprocessing step. The other method is using the Sparse Data Model where they only use known features for classification, and classify based on features multiple users rated in common. The article found that both methods perform better than correlation-based CF.

The Naive Bayesian Classifiers are robust to noise points and irrelevant attributes, while handling missing values well. Bayesian classifiers are also common to use in content-based recommender systems, which suits well for the problem of this thesis. Ghani and Fano [24] use this method in a content-based recommender system, which allowed them to recommend unrelated categories. This is something that could prove useful for the personalized search described in this thesis. One thing that could prove a problem is the fact that attributes have to be independent to give accurate results. In these situations methods like Bayesian Belief Networks can be used to overcome the issue. Another problem using Bayesian classifiers is that they are susceptible to overfitting, and need time to be tuned correctly.

2.9.3 Support Vector Machine

Another classifier is SVM [22]. SVM is a newer and more advanced method than the others described. The basic thought behind the method is to separate the different classes with a linear hyperplane. This is done in such a way that the margin between them and the line is maximized. This maximizes the likelihood of classifying a new item correctly. The task becomes an optimization problem on how to maximize the margins from each item to the line. There are a lot of mathematics and theories in making this algorithm work, we will here only explain the basics and some benefits and problems with the method.

If the items are not linearly separable, there are two main ways to overcome this. One way is to use a soft margin by introducing a slack variable. This allows the classifier make a few "mistakes" while creating the separation line. It measures the misclassified points, and how far away they are from the surface. We can then compute a cost for the different misclassified points. This technique is also useful for handling outliers/noise data.

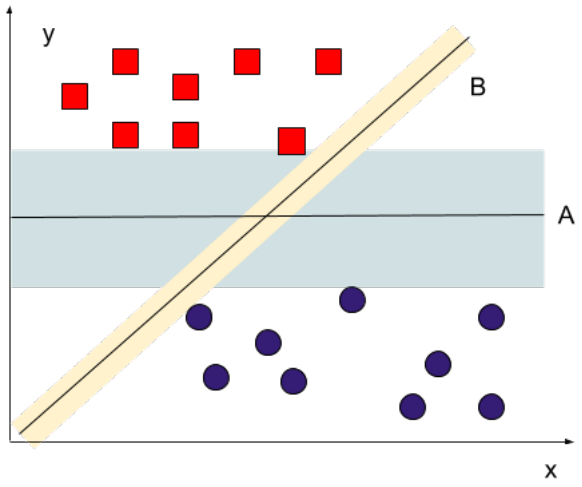

Figure 2.4: Example of how SVM works in a two dimensional plane. We see the separation lines in the figure and their margins. In this case A is a better separation line than B.

The other solution is mapping the items to a higher dimensional plane where there exists a linear separable solution. The way this is done is by defining different kernel functions to the items, instead of mapping them directly. The optimization problem again becomes an optimization task with creating the now linear hyperplane as in the original non-linear problem.

SVM is a method that gets a lot of attention lately, because it in many cases shows good performance and efficiency. There are also cases where SVM starts to show promising results in recommender systems. As this is an efficient method, it could be interesting to experiment with, when creating personalized search. Even though this method, as with the Bayesian classifier, is prone to overfitting and needs tuning to work well.

2.9.4 Random Forest

Random forest classification is built on decision trees [22]. A decision tree is one of the closest things we come to an off the shelf method for data mining tasks, in the way that it produces respectable models, handles irrelevant features well and is invariant under scaling [25]. Decision trees can be used in both collaborative filtering and in content-based systems. In CF the method

works by attempting to find different properties from the users that like certain products. After finding some common criteria, these properties are used to evaluate whether a new user would like the product, based on their preferences or properties.

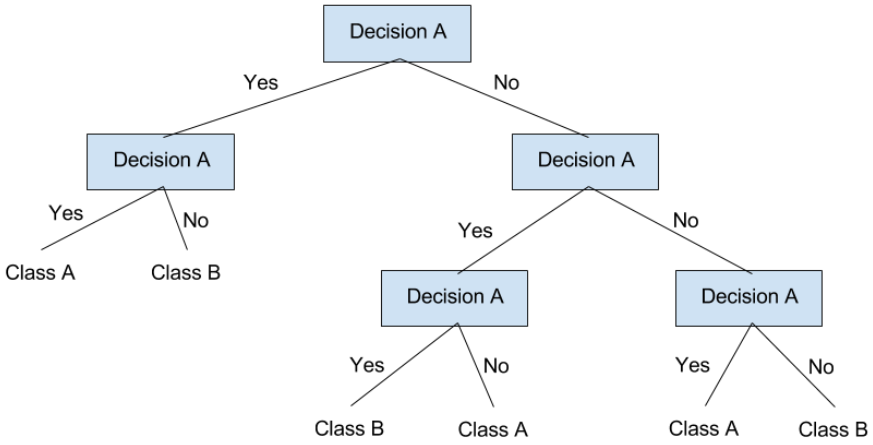

Figure 2.5: Simple illustration of a decision tree

Let us see how this could be done with a content-based approach. There are multiple ways to solve this. One method is using features about the items to decide whether a user likes it or not. E.g *"if the user is interested in fantasy, the item should be recommended"*. After checking the user against all the relevant features a decision is made on whether the user would like the product. The other method is creating one giant decision tree for all your products, and use user preferences to find relevant items. A problem using decision trees to classify data is that they have a tendency to overfit to the training set provided, and are also seldom accurate.

To overcome these issues random forest can be used [22]. This method creates multiple decision trees, and some average between these trees are used to create the final decision tree. The different trees will use different training sets to create their model, so the result should not be overfitted to a specific training set. Using random forests usually reduces variance and produces better results for the final model. Negative aspects of this method are that it needs more computations. This is usually not a problem since this is done offline. Other cons of random forest are that it can increase bias as well as losing some interpretability. When used to creating personalized search, another negative thing about this method is that it does not rank the product

classified as likeable by the user. This method will only tell whether the user will like it or not.

Chapter 3

Approach

In this chapter, we describe how the instant personalized search engine is designed and created. We also look on some testing and optimization done. Section 3.1 presents the criteria for our instant personalized search engine and findings from the pre-study. Section 3.2 explains the design for the prototype created. Section 3.3 explains how the user interface is created and what it looks like. Section 3.4 presents initial tests conducted to check the validity of the different methods implemented. In section 3.5 we test for parameters and techniques that optimize our prototype.

3.1 Requirement Analysis

In this section, we define criteria for the personalized search engine, and describe the designs found in the pre-study

3.1.1 Criteria

In order to create a search engine suitable to test the concept of instant personalized search, we need to set some criteria for the system.

The goal is for the search to be fast and give accurate personalized results. One of the key aspects of this personalized search is to not only complete the user's query or return apps with titles starting with the same words as typed in. In order for this suggestion to provide anything new to the world of personalized search, it has to be able to return results beyond this. The

thought is that showing suggestions for apps based on preference and query, might reduce the number of steps required for the user to find the product they like. This will in turn, make it quicker for the users to find products they like.

When building a search engine, different techniques can work on different datasets. The data we managed to get, which we can use for our search engine, is information about the apps on Google Play. This means our solution has to work well with only content-based information, as we do not have the necessary information to use collaborative filtering. To work well with only content-based information is not a general criterion for this kind of personalized search, but a restriction based on the data we have access to in this thesis.

As we have seen in section 2.5, the dataset for app usage in Google Play is very sparse. This leads us to believe that using content-based information in general, might be a good idea, as the dataset for collaborative filtering is very sparse. Personalizing results based on content-based information might also increase novelty, meaning we might help users find apps they like which they were unaware of beforehand. When using collaborative filtering for suggestions in the app store domain, suggesting new apps can be hard as usually the same apps are downloaded by most people. New apps therefore do not have the same data basis to be suggested to users. The search engine we are making will be used to show whether it is beneficial for the user to be exposed to apps they are not aware of from before and explore deeper than just completing the search, as search today is often used to find specific or popular apps [4].

The number of apps in Google Play are massive, and growing quickly [26]. Therefore, it is crucial that the system will work with large amounts of data. Even though we will only create a concept which can be developed further, scaling is one aspect we need to keep in mind.

Based on this information and the research questions in section 1.2, this list of requirements for the system was made 3.1:

#	**Criteria**
QC 1	Be able to produce results in around 100ms
QC 2	Works by only using content-based information
QC 3	Works without user's search log or purchase history
QC 4	Works without necessarily containing keywords typed
QC 5	Rank recommended items based on user profile and query
QC 6	Scales with large number of products and users
QC 7	Works well in very sparse data sets

Table 3.1: Criteria for algorithm

These criteria were first created in the pre-study [5] conducted when designing this proposed personalized search.

3.1.2 Findings from Pre-study

The pre-study conducted for this personalized search, analysed state of the art methods used for personalization and suggestion, while analysing which algorithms could be used for the prototype we try to make.

This pre-study analysed the classifications described in section 2.9, together with query personalization investigated in section 2.6 and search described in section. 2.8. These methods were put together, making different solutions for the problem. These designs were compared to each other and the criteria given in table 3.1, ending up with the table below 3.2

Algorithm	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
BookmarkRecommendation	-	X	X	X	X	X	-
Getjar	-	X	X	X	X	X	X
Long-term user history	-	-	-	-	X	X	X
SR-kNN	-	-	X	X	X	-	-
SR-Bayesian	X	X	X	X	X	X	X
SR-SVM	X	X	X	X	X	X	X
SR-RandomForest	X	-	X	X	X	X	-
RUP-kNN	-	-	X	X	X	-	-
RUP-Bayesian	X	X	X	X	X	X	X
RUP-SVM	X	X	X	X	X	X	X
RUP-RandomForest	X	-	X	X	X	X	-

Table 3.2: Algorithms compared to criteria

Here we can see some solutions mentioned in related work 2.1, and different methods designed during the pre-study. The designs made in the paper is named either SR-[classifier name] or RUP-[classifier name]. SR is short for search in recommendations, and RUP is short for Re-rank based on User preferences. These methods will be described in detail in section 3.2.4.

As we can see from table 3.2, we ended up with four relevant designs. These are SR-Bayesian, SR-SVM, RUP-Bayesian and RUP-SVM. The designs seem to fulfil all the criteria, but the research ended by concluding they needed further investigation to test their viability. These four designs are the general methods used when creating the prototype for personalized search.

The Naive Bayesian classifier and Support Vector Machine ends up with somewhat the same result, a ranked list of apps recommended to the user. Because of the time limitations of this thesis, we will only test with the Naive Bayesian classifier. This is a simpler algorithm, and more commonly used in recommendations than SVM [22]. There are also packages that implement Bayesian classifiers, thus it will not require us to implement the algorithm ourselves. Even though it might be the case that SVM is better suited than the Naive Bayesian classifier, the purpose of this thesis is only to show whether the concept works. If the test shows that this form of personalized search is beneficial for users, more research can be conducted in finding the most efficient way to provide results to the user.

3.2 Design

This section will describe the design of the system, and the process of how this design was decided. First, we will give an overview of the design with the technologies used, then we will follow the categories used for a contend-based recommender system in section 2.7 and look on the content analyser, profile learner and filtering component used in the prototype created.

3.2.1 Overall Design

An important factor for how the system was designed is the choice made to run the system as a website. This was because the system is meant to be tested for online stores, and because it would make it easier for testing when people from all over the world could access the site. In order to achieve instant search, the technology chosen had to be able to quickly give response

to the user. The choice fell upon Nodejs [1]. There were several reasons for this choice. It is a language that is well known by the researchers of this thesis, which meant less time would be spent learning the language. Nodejs has also shown to have a high performance thanks to the technique called the event loop, and the fact that it is running on top of V8 [27]. One last thing to note is that Node also has a well-developed library system called NPM [2], with packages to help perform a number of tasks. This makes it possible to simply use tools others have made for suggestion and add it to our own system to spend less time making well-known algorithms.

MongoDB was the choice for database. MongoDB [3] is a document based database, as opposed to relational databases such as MySQL [4]. MongoDB has a library called Mongoose [5] which makes it work easily together with Nodejs. It is also efficient on larger datasets [28] like major online stores have today. NoSQL databases seem to be quicker than SQL databases in most cases [28], especially when retrieving objects from the database which does not involve joining multiple types of objects.

Even though it is possible to do queries directly to MongoDB, other systems can be used for more advanced search. Our prototype use Elasticshearch described in section 2.8. Elasticsearch is quick to set up and use, it also has a lot of methods to perform more advanced search. It is also possible to distribute the data across different shards in order to speed up queries on larger datasets.

Figure 3.1 shows an overview of how the different parts of the system are connected.

[1] https://nodejs.org
[2] https://www.npmjs.com/
[3] https://www.mongodb.org/
[4] https://www.mysql.com/
[5] http://mongoosejs.com/

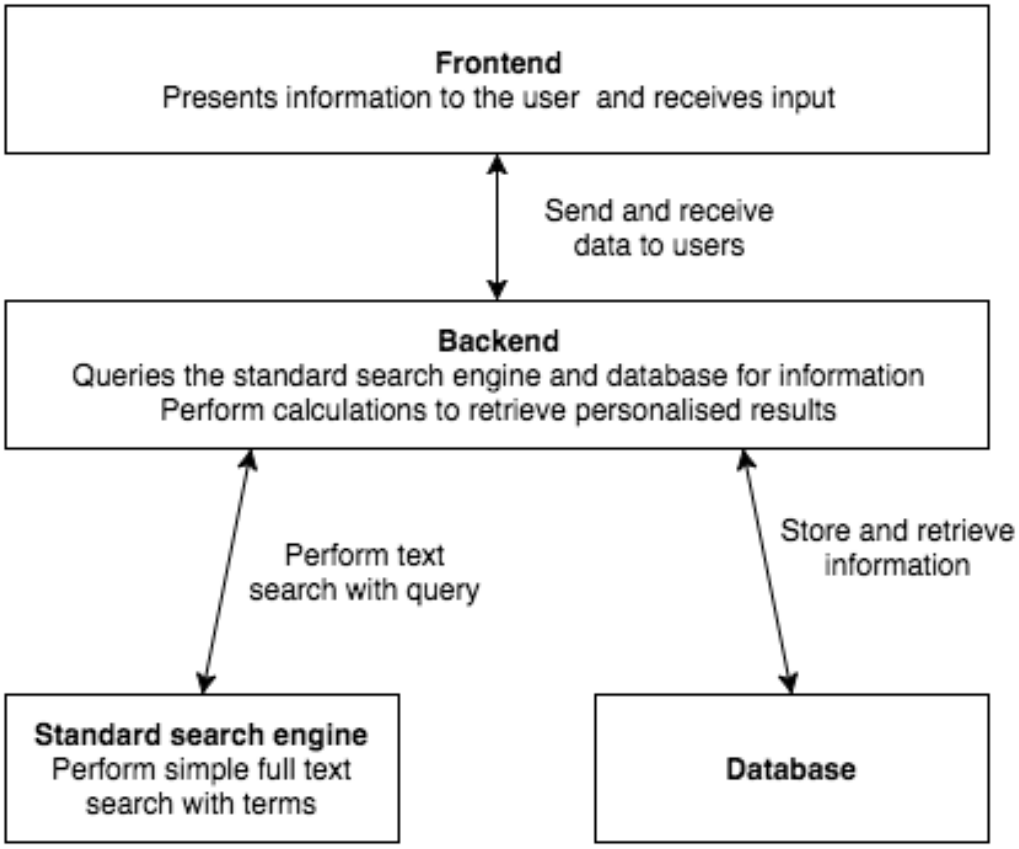

Figure 3.1: A general overview of the system

3.2.2 Content Analyser

In order to make sure we get the best possible results, and that they are retrieved quickly, it is important to do the necessary preprocessing. The apps gathered contains title, images, description, number of downloads, rating, reviews and some other properties.

Since the only information we had about the content of these apps were the description, it was chosen to extract features from this property. As these descriptions contain multiple stopwords, and comparing all the words in the text could be time-consuming while providing a lot of noisy data, it was chosen to only use the most relevant terms as features.

A list of the most common words was made using Zipf's Law [29]. This was done by finding the frequency of the terms, order these terms by frequency and find where big gaps in term occurrences happened. The places where

these big gaps occur, are the ones that should be used to decide what words are defined as stopwords according to Lo [29]. There are multiple places such gaps appeared, and testing was done to see what cuts would give the best results for the Bayesian classifier.

In addition to creating a list with the most common words, which would be removed from the description when extracting features, the text was also tokenized and stemmed. The natural library [6] for Nodejs, were used for the task. This gave us a list of all the words in the text not considered to be stop words, that could be stored together with the apps.

These terms could be used directly as features, but it was decided to do some other calculations to find the most important terms and minimize the noise from our data. After some testing, using KL-value with one app at a time to find the top 5 relevant terms, and using these as features showed to produce good results. The testing performed and detailed description of this is given in section 3.5.1.

3.2.3 Profile Learner

The profile learner builds the users' profiles by trying to find what they like and dislike. There are mainly two ways to collect user information as we have mentioned earlier, implicit or explicit. Even though we see that it can often be beneficial to collect information using both methods, implicit information is harder to get in a test setting where the user will only use the system for a short time. This is because we do not have the ability let user download and use the apps. In order to make sure the system get information about the user's preferences quickly, the user will have to explicitly tell if they like a number of apps. Whether a user likes an app or not will determine what features a user is believed to like and what they are believed to dislike. The different designs for personalized search created all use the Bayesian classifier as profile learner. In addition to this profile, we also build a profile for users' preference to features used in apps. This information is only used by the re-rank based on user preferences method.

Naive Bayesian Classifier Profile

How the Naive Bayesian classifier works is described in section 2.9.2. Since the system could use the Bayesian classifier straight out of the box, the library

[6] https://github.com/NaturalNode/natural

Natural, which was also used for tokenizing and stemming, was used. Every time a user express their preference for an app, the relevant features are added to the classifier together with their preference. This information is then used as a training set. The classifier can then be trained using all the information stored, and then again used to try predicting whether a new app is relevant based on its features. We give the classifier two classes, "like" and "dislike". The dislike class here does not necessarily mean the user dislikes the item, just that the user do not like it.

The classifier will give a score from 0 to 1 in both classes presented to the classifier. An example would be ['Dislike': 0.8, 'Liked': 0.02]. In this example, there is a high chance that the user dislikes the app, and a low likelihood that the user likes the app.

The system can use this profile both to store apps the user likes, ordering them based on how likely the user is to like the app, and to classify whether a user likes an app returned from one of the search methods.

Term Scores

In section 2.6 we discussed how user preference can be stored in liked features or categories. Our solution implements this feature by calculating the KL-score for the terms in the description of the app. The KL-score is then added to the score already stored by the user which is 0 by default. If a user likes a feature it will be a positive number, and if he or she dislikes a feature it will be a negative number. It is important to normalize these preferences to get accurate results [10]. This is done so that 1 and -1 are the maximum and minimum preference a user will have to each term.

This means we have to keep two lists stored with user preferences. One list with the total scores given for each feature, and one list with the normalized score. The way we normalize these scores is by finding the largest absolute number in the list with total scores. Then we divide all the scores by this absolute number, leaving a list with scores ranging between -1 and 1.

The reason we use KL-score instead of TF/IDF, which is also used to calculate the relevance of terms, is because KL-score is used in the term reweighting process described in 3.4.3. This means that combining these scores will then be easier as they are calculated the same way.

3.2.4 Search and Filtering Component

The last part of a recommender system is the filtering component. Since we are building a search engine together with a recommender system, we also need to search for apps when filtering which of them are considered relevant. Below we will discuss the different designs used for this step.

Normal and Popular Search

Two baseline search techniques are implemented. We have chosen to call these normal and popular search.

When performing normal search, the user query is presented directly to Elasticsearch. Elasticsearch then uses techniques to rank the results from a standard text search. How this is done is described in section 2.8. We have also tuned the search so terms represented in titles weigh more than terms found in the description of the app. This is because we see the title of the app as a good indication of what the app is about, and probably more accurate than the features extracted from the description. Elasticsearch will then retrieve results to the query, and order them according to relevance. Normal search directly shows the result in the order received.

For the popular search, we first use the normal search and retrieve the top 100 apps found. Then we rank them solely based on number of downloads. The app with most downloads will appear as result number 1, then the second most as result number 2 etc. The reason 100 apps were chosen to evaluate, was that we needed some apps to re-order to be able to produce different results with different methods used. In order to make sure the other more time-consuming methods would still show results in reasonable time, we found that 100 apps seemed to work fine.

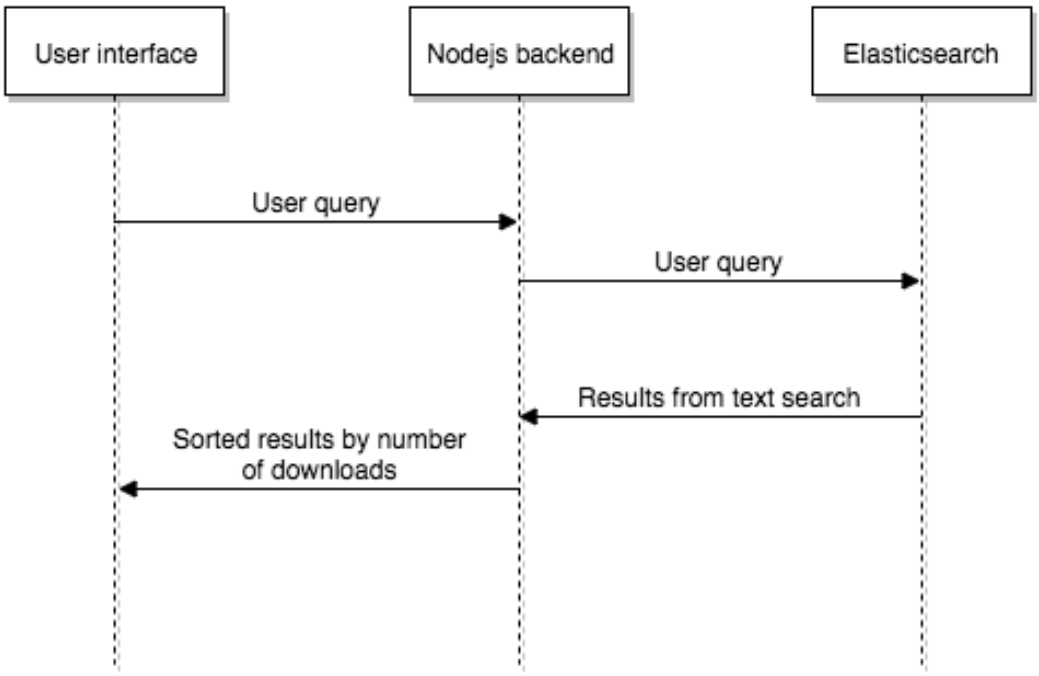

Figure 3.2: Sequence diagram showing the sequence during popular search

Search in Classification

This method was designed in the pre-study and believed to be one of the quicker methods as it had fewer operations at run time. The system will in advance classify all the products for the user, and prepare them before the search engine is used. This is done according to the description in 3.2.3.

When the user provides a query to the engine, the system can simply search in the apps already classified as interesting for the user. The result would then be ranked according to how normal search is ranked in section 3.2.4 combined with the likelihood of the user liking the item. How these scores are combined can be tuned, but as both scores are normalized, simple addition seemed to work fine.

In Elasticsearch we store all the apps relevant to a user in a specific index, so the user can search directly in relevant apps. They are also stored together with the score they got from the Bayesian classifier which can then be added to the score from the search with Elasticsearch methods. This method would probably not scale well with a number of users, as it would require a lot of indices and all users would need to store apps relevant to them as a duplication

of the existing apps. Since this test will only have a limited amount of user it can work to check the validity of the method.

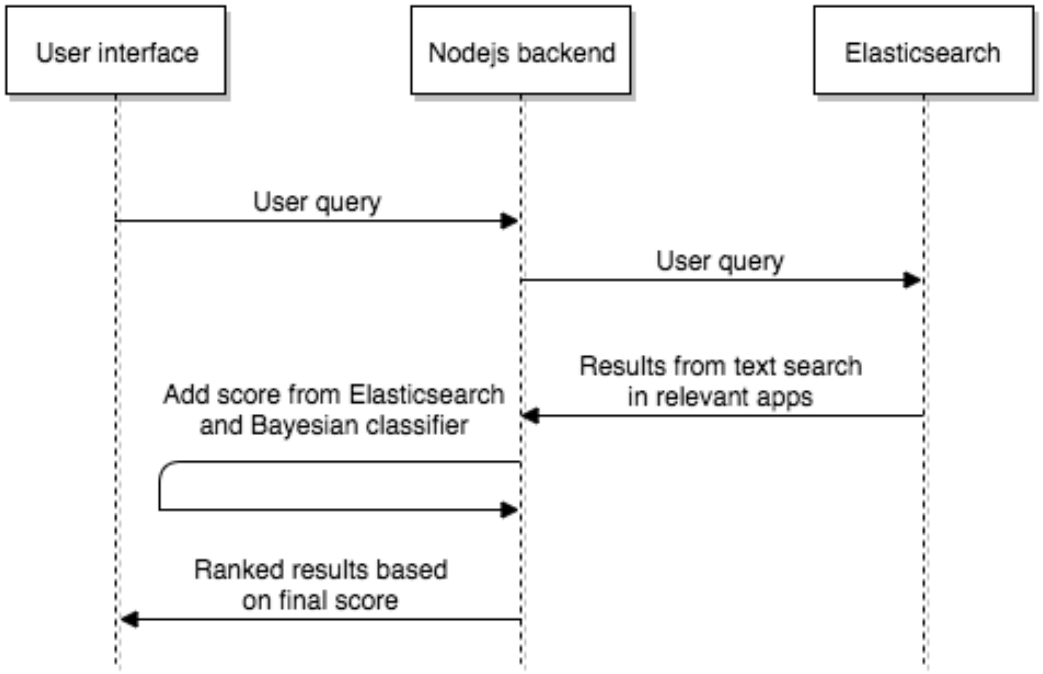

Figure 3.3: Sequence diagram showing the sequence during search in classification

Classification in Search

Classification in search is called the Bayesian method throughout this thesis. The idea behind this search is to order the results from the query based on the user's preferences. This method was not described in the pre-study, but was discovered during testing of the term reweighting method. Since the results needed to be ranked according to the user's preferences, and this method was used, it was discovered that using the classification without term reweighing would give interesting results. Users first have to express preferences for some apps, and these apps are then used as training set for the classification.

The way it works is to first do a normal search like described in section 3.2.4. We retrieve the top 100 apps from this search, and classify all these apps using a Bayesian classifier. The classifier will give a score as to how certain it is that the app is relevant to the user. The higher the score from the classifier, the higher the app will be ranked. As opposed to the previous method, we do

not combine the score for relevance of the search and user preference. Instead, we only use the likelihood of the user liking the item to give the final score. Because of this last step, the results should find apps less directly rated to the query than the search in classification method.

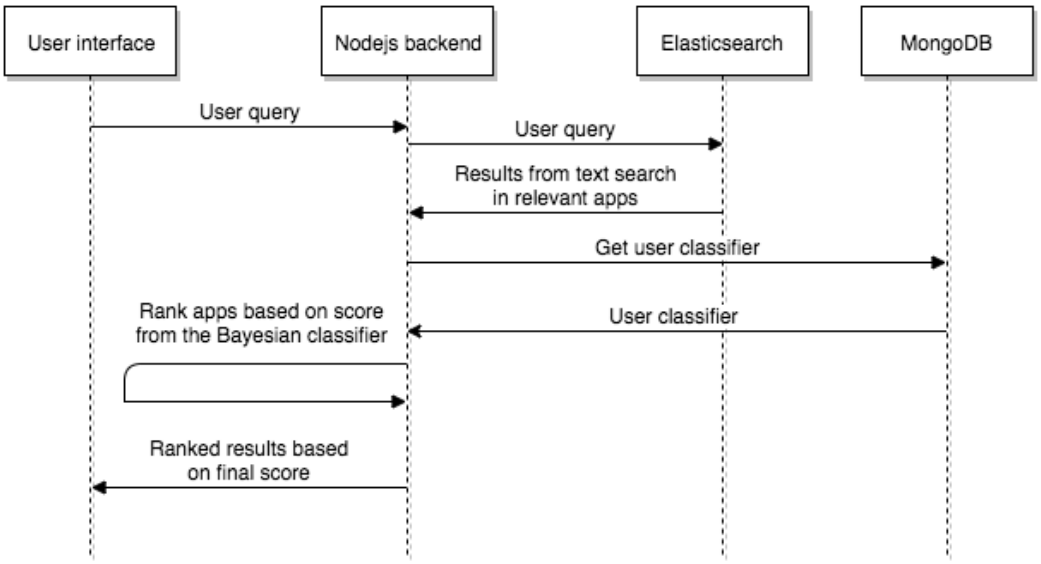

Figure 3.4: Sequence diagram showing the sequence during classification in search

Term Reweighing

This method is the most complex and time-consuming. The search is based on the more standard term reweighting methods described in section 2.8.5.

First, the user presents a query. This query is sent to Elasticsearch, which returns a set of the 100 most relevant apps to the query. When finding the most relevant terms from the query, we use the KL-method described in 2.8.5. The relevance score of the terms found is then added with the user's preference for the given terms. The final score is then used to find the top three relevant terms for the user and query combined. These top relevant terms are added to the original query presented by the user, and used to do a second search.

The new and reformulated query is presented to Elasticsearch which again returns the 100 most relevant apps to that query. Since it is important for the search recommendations that the first few results are relevant, as these are the only one seen by the user, we sort the result using Bayesian classifier.

This second step is similar to what is being done in the classification in search method described in section 3.2.4

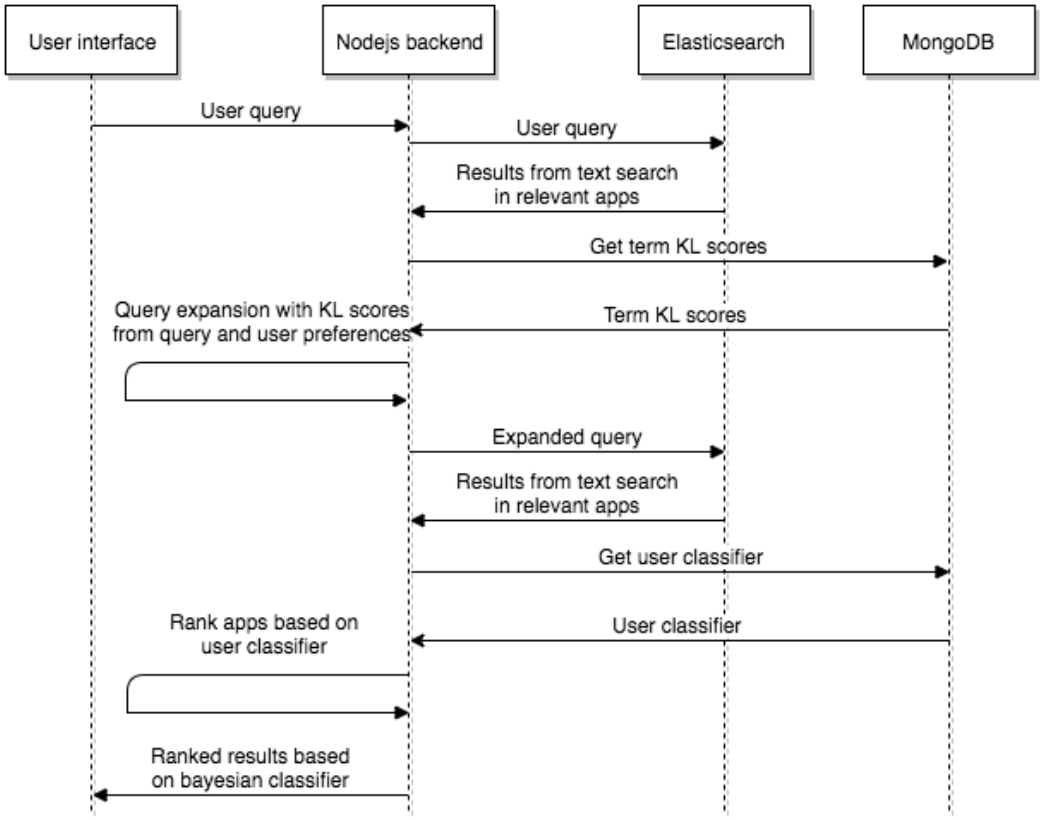

Figure 3.5: Sequence diagram showing the sequence during term Reweighing and search

Since calculating the KL-score for the apps returned from the query has to be done as quickly as possible, the data is pre-processed to reduce the calculations during online processing. In advance, all apps have been analysed, and contains a list of all terms and number of times these terms occur. The total number of terms per app has also been calculated. This way the system don't have to calculate term occurrences real-time. The number of times each term occurs in total for the whole collection, and the total number of terms for this collection has also been calculated. Because of this, we have all the values needed for calculating KL-scores in advance, except the total number of terms in the top-k documents for the query. This value is luckily less time consuming to calculate than some of the values already calculated and only has to be done once for each query. Even though the pre-processing step is

a time-consuming process, this can be done offline, speeding up the process when the users present queries to the system.

This method will be referred to as the KL method in this thesis.

3.3 User Interface

In section 2.4 we discussed the importance of user interface when creating search suggestions. Because it is not the scope of this research to test how the user interface affects the users of query suggestions, we made sure that suggestions would show similar to how they appear in Google Play today. The way this is done is by showing the suggestions directly under the search bar. Results are also automatically updated whenever the user inputs a character into the search bar.

The difference between the user interface of our search engine compared to Google Play, is the fact that our solution suggests apps while Google Play only instantly suggest queries similar to the user input.

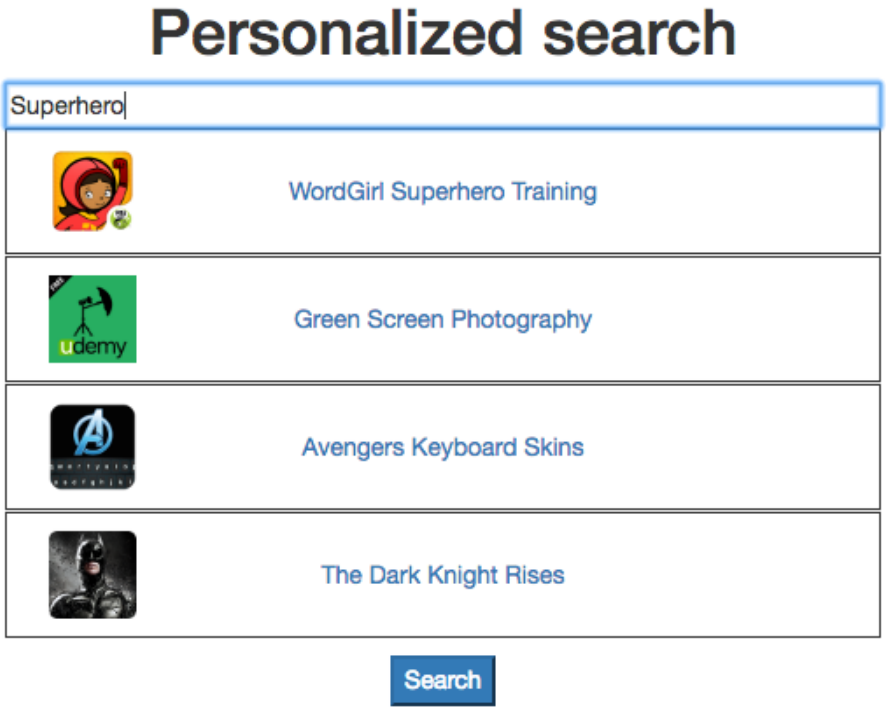

Figure 3.6: The user interface for our instant recommendations

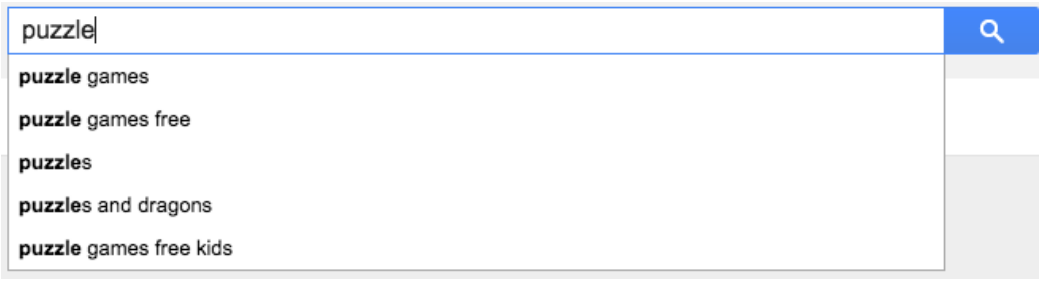

Figure 3.7: The user interface Google Play's query suggestion

As we can see in figures 3.6 and 3.7 there is a difference in the number of suggestions given. Google Play seems to dynamically adjust the number of suggestions given, where the most we found were six suggestions, and the least we got were 1 (or 0 when there were none). Our solution returns up to four results, and always return four apps when there are enough suggestions. The reason we use fewer suggestions is that these suggestions take up more space on the search site. By returning 4 results it is also made sure that the number of suggestions is somewhere between the number of results provided by Google Play.

In addition to providing suggestions, we have search pages showing more results provided by any of the four methods described in section 3.2.4. The system returns the top 10 relevant apps based on the query and search algorithm. Since we do not need the user to download these apps, the user can instead select whether they think the app is relevant. Figure 3.8 show how the top part of this result page looks like. It is possible to scroll down to find more results. As shown in figure 3.9, it is also possible to see screenshots from the apps to get a better look at how the app works. Whether to show these can be toggled by the "Click to see screenshots" button.

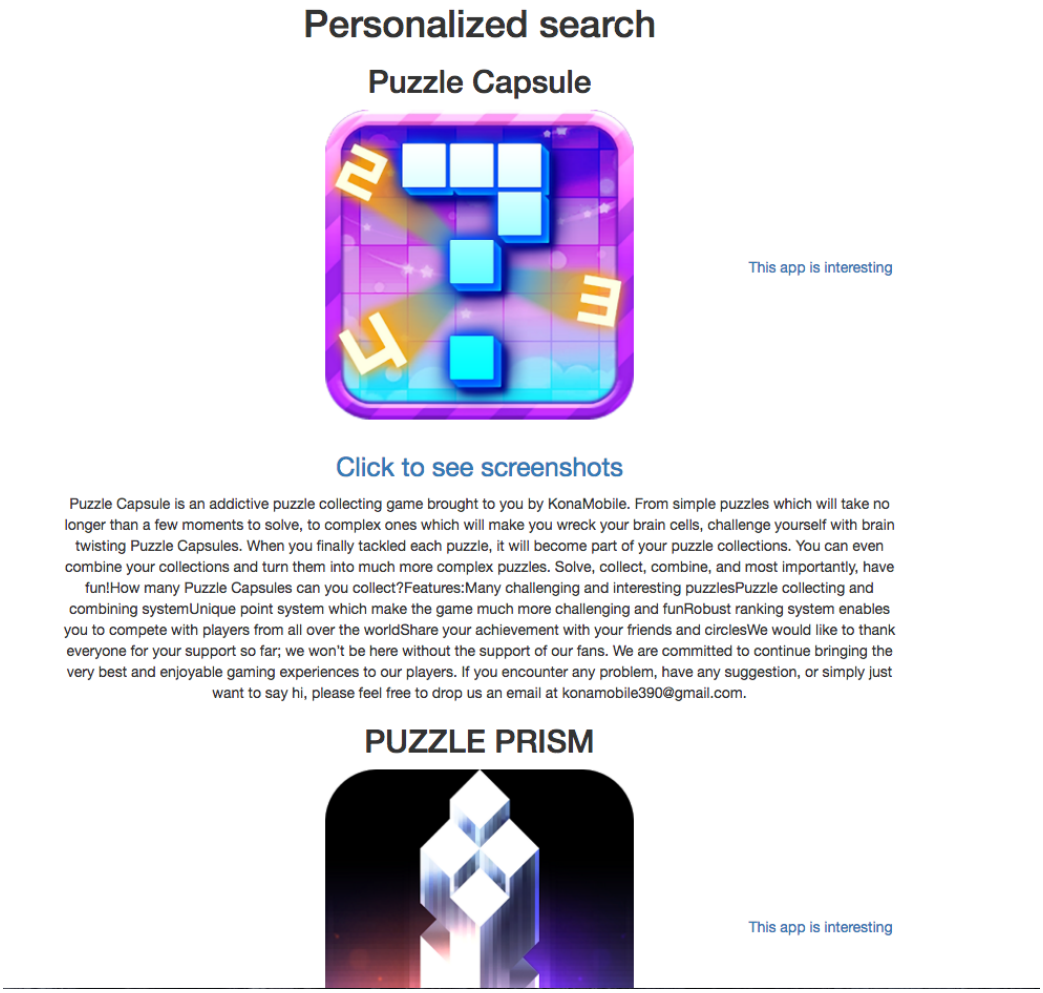

Figure 3.8: The result page for when the user performs a search

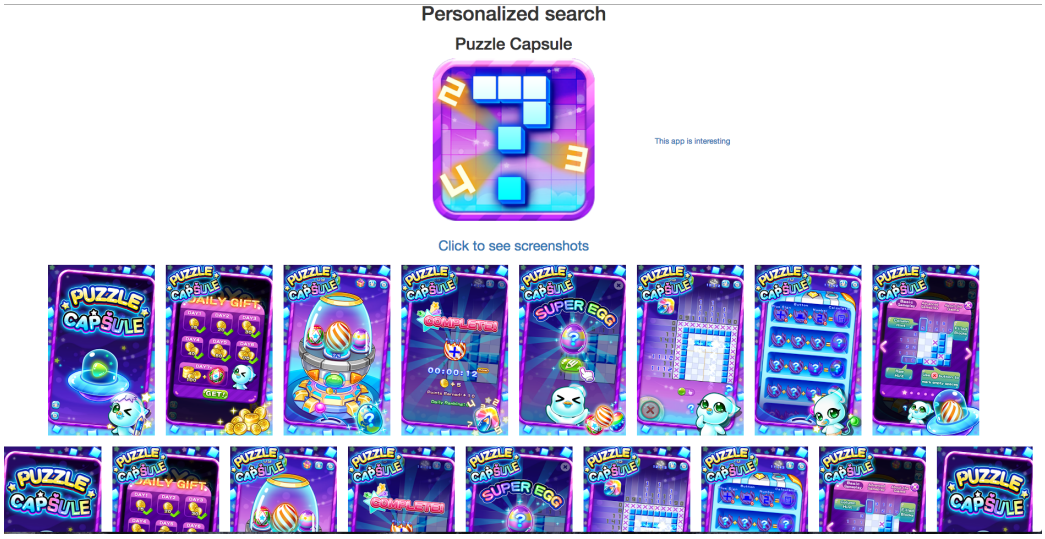

Figure 3.9: Screenshots for app toggled

3.4 Initial Testing

Before fully implementing any of the methods proposed, we chose to perform initial tests on them. This was in order to check if they looked promising before spending time implementing them fully. A small group of 5 users tested the different systems in order to see whether the search engines could provide relevant results.

The test was performed on a set of 9650 apps from Google play. The description of the apps was used to find terms or features. The words were stemmed and tokenized, and the 100 most common words were removed as stopwords. A simple user interface was created, where the results would show up under the search bar as soon as they were prepared for the user. The tests were performed on a server, so the delay between the local computer and the server would be present. The server is run on Digital Ocean [7], and the server has 1GB RAM and 1 CPU with 2.0GHz

When testing for speed locally without network delay a Macbook pro late 2011. With 2,4 Ghz Intel Core i7 and 8 GB DDR3 RAM.

[7] https://www.digitalocean.com/

3.4.1 Search in Classification

As expected these results were retrieved pretty quick, and locally the results would be retrieved in 92 ms on average. The results with latency from the server would show up in anything from 80-200 ms. This shows that the method may have potential to give instant results, which is important for this kind of search. The two main problems encountered during this period was that the same results would be given often in cases where the user had few items classified as relevant, and that the top results given were not necessarily the most relevant to the user. The last problem was due to the fact that a normal search was done in the recommended apps. Even though all the items were thought to be somewhat relevant for the user, it is important that the top four results returned is thought to be most relevant, as these are the apps presented to the user. Due to these problems, it was decided not to pursue this type of recommendations further.

3.4.2 Classification in Search

Locally the results would be retrieved in 108 ms on average. The results returned by this method showed up somewhere between 100 - 250 ms from the server. The time used might be a little too long, but could work as a prototype. The users testing this method experienced this as giving results correlated to the query and giving the most relevant results to the user. Even though this method was not designed in the pre-study, because of the promising results it was decided to conduct further tests on this method.

3.4.3 Term Reweighing

A major problem with reweighing terms based on user preferences was that this could easily be a time-consuming process. The engine needs to search two times for apps, and process both of those search results. Locally the suggestions retrieved would show up in 238ms on average. Initial testing on the server gave results in between 150 - 600 ms. Even though this is longer than the 100 ms requirement for instant search, it could give a quick enough response for testing as a proof of concept. When using this method to search for apps, user experienced that the results they got could be different from what they expected. This could prove to be both positive and negative. They also discovered new apps they found relevant, which means this method could increase novelty. Even though this initial testing does not prove this is a good

method to use, it gives an indication that it might have potential and it was decided to be tested further.

3.5 Test for Optimization

When extracting features and suggesting apps for users, some initial testing was done to optimize the results from the search engine. In order to do this, we gathered five users' preferences to 75 apps. The results from these tests can be found in appendix A.

One thing to note is that it is not necessarily a problem if the solution has trouble classifying the apps correctly. This is because the ordering of the items are more important than whether the classification method classifies correctly. In other words, it is more important that what the apps are ordered correctly, than classifying the apps correctly. This being sad, the test can still give an indication for how well the classifier performs, and how we can optimize it.

3.5.1 Optimizing App Information

As there were multiple ways to extract information from the app description, we used this test set up to see what features would give the best accuracy. The features we tested were: app description, app category, top five terms based on TF/IDF score and top five terms based on KL-score. Different combinations of these features were tested, and as we can see KL and TF/IDF features performed the best by classifying correctly about 65% of the time.

Since KL is used more in our system in general, besides the normal search Elasticsearch performs, it was chosen to use the KL-features for our system as well.

3.5.2 Optimizing Suggestion

As the previous result shows, the classifier still guesses wrong a fair amount of times. In order to overcome this, a test was conducted where we also took into account what categories the user liked. The hypothesis we wanted to test was that it is more plausible that a user likes an app in a category they claim to like, than an app in a category they do not claim to like.

The Bayesian classifier gives a number between 0 and 1 for how likely it is that a user likes the app, and how likely the user is to dislike the app. In the test, we increased the probability that the user likes an app if the category was one the user liked, and decrease the probability that a user likes an app that is not in the desired categories. We used the KL-features as the previous test in section 3.5.1 showed provides the best result.

When giving extra points to apps in liked categories, and subtracting points for apps not belonging to categories liked by the user, we ended up with a worse result where the system classified correctly about 54% of the time. When only giving more points to apps in liked categories, and not subtract points to apps belonging to categories not liked by the user, we got an even worse performance with correct classification around 50% of the time. The last test where we only gave negative points to apps not appearing in liked categories, we got the best result with correct classification around 72% of the time.

We learned two interesting facts from these test. The first being that since it is better to only give negative points to categories not liked, instead of giving more points to apps in categories liked, users seem to be more likely to dislike apps in categories they like than to like apps in categories they do not like. To check if this was correct for our dataset we performed an isolated test for this. For the participants we gathered data from, there were 52 apps they liked which were not in their liked categories, and 189 apps they disliked in their liked categories. This indicates that it is not likely for users to like apps in categories they state not to like. The second fact we learned from this is that even though modifying the classifiers score based on app categories can result in worse performance in some tests, is should be a beneficial method to use when producing the final score for the relevance of the apps. In other words, apps with categories liked by the users should generally be sorted above apps with categories the users do not like.

After performing these initial tests, we decided to keep giving negative scores to apps not in liked categories. It is important to note that it is not a problem that our system classify more apps as desirable for the user than which is the case, as long as the most desirable apps are ranked higher than products not liked by the user. Pseudocode describing how this classification is done, can be found in Appendix C

3.5.3 Optimizing Suggestion Retrieval Time

Since none of the implemented search methods are able to retrieve results under 100ms, others techniques can be used to make the suggestions show up quicker. A technique used was to start searching for suggestions early. Every time a user presses a key, the query inserted in the text field are sent to the search engine and suggestions are retrieved. The search done is fuzzy, with an edit distance of two. This means the words in the query do not have to have an exact match with the words in the documents to be retrieved. This help users who spell words wrong, and also make it so the system often finds the app for the final query in one of the three last queries presented by the user. If the user wants to search for an app called "Personalized App", all the queries "Personalized A", "Personalized Ap" and "Personalized App" would match the name. So in many cases, the apps for the final query is already found before the query is written. This can give the impression to the user that the search engine is quicker than it is, but in cases where the final result gives a different result than the previous one, the user will see that apps in the suggestion list are shifting.

The negative aspect of sending multiple queries from the user this way is that it will increase the load on the server. Every user will send multiple queries in a short amount of time, especially if the query they insert has several characters. For system like this with few users, and relatively little data being sent between server and user, it is not a problem. For larger systems with a mores users active at the same time, it could require too much resources from the servers. Another method one could use for instant search to decrease load, is to wait and check how long of a pause there is between the characters typed, and only do the search after a given amount of time. This will again increase the time before the user will get the results.

Because of the low number of users that will be active at for the test simultaneously, and to speed up the suggestion retrieval process, the final system uses this early querying to retrieve suggestions.

Chapter 4

Experimental Setup

In this chapter, we describe the quantitative test set up. Since the system is a custom design, we also had to make a custom survey. Before creating a test, we made sure to state what information we are looking for. This experiment should help to answer the research questions in 1.2. To answer these the test should gather information about the relevancy of results returned, usage patterns from the participants and the users' feelings about the system. The test will also gather information from the baseline methods implemented. The data can be used to compare our personalized solutions to the baseline methods.

Section 4.1 explains how the test is meant to make the participant anonymous. Section 4.2 presents the dataset required and the server used in the test. In section 4.3 we set up requirements to make sure the test results will answer the research questions and to make sure the test will be easy to complete for the participants. Section 4.4 explains the test in detail, and the different tasks presented to the participant. Lastly, section 4.5 explains how we store information from the test.

4.1 Achieving Anonymity

The test gathers information about the user's age group, preferences for apps and feedback about the prototype. This information is not by definition sensitive, as it can not be linked back to the respondent. To make sure users answer sincerely and feel safe answering the test, we made sure the participants would also feel anonymous. This is important, as some users

might be embarrassed by their app preferences, and not feeling anonymous might make them not answer according to their preferences. This again would lead to noisy data being collected. Here are the measures taken to make sure the users feel anonymous:

1. The user is first presented with an explanation describing the purpose of the test, and handling of data collected.
2. The users are given random usernames they can use, to make sure their names can not be associated with the person undergoing the test.
3. The test is distributed over the internet, making sure no one has to oversee the participants answering.
4. No questions are asked that can be used to figure out who the test person is.

4.2 Dataset

After the choice was made to test the instant personalized search in the app store domain, we needed to find apps to use in the test. Since creating mock apps is time-consuming, and not ideal since several users probably will look for real apps they know of, it was decided to try retrieving real apps.

After looking at some of the major app stores, like Apples App Store, Google Play and Windows Store, the easiest accessible data we found where from Google Play. There are projects where Play Store is crawled for all available information on their site, to gather a database of apps. The project we used were the Github-project Google Play Apps Crawler [1]. This is a project that crawls for apps on Google Play's website for app information, and multiple people are collaborating in gathering information to one large database. This is useful since one user can only get a number of pages before Google blocks them out for a while.

After getting a hold of the database, we were left with 1 145 535 apps. Since it was a time-consuming process trying different techniques to structure the data, and because many apps had poor description or description in other languages, we removed most of the apps. We decided to remove all apps that are not from a top developer or where the description of the app was in another language than English. By removing these apps, testing was quicker,

[1] https://github.com/MarcelloLins/GooglePlayAppsCrawler

while most of the apps that are popular and known by most users remained in our dataset. It is also believed that the most popular apps would generally have more informative descriptions than some unpopular apps. After these operations, we ended up with a dataset with 9650 apps. The app information was updated 2015-04-10.

As mentioned earlier, the dataset has multiple properties from the app. Most of these were removed as it is not considered relevant for the test. The properties we were left with were: category, cover image, description, number of installation, if it is from a top developer, name, screenshots and URL to app in Google Play. Two interesting properties we still decided to remove, were related apps and score given by users. Related apps were removed, because most of the dataset we started with was removed, meaning multiple of these related apps would not be present. The score given to the app was also removed, as it was decided to use popularity by number of downloads instead of the app's score.

The server used is the same as used in the initial testing. A server on Digital Ocean with 1 GB RAM and 1 CPU with 2GHZ.

4.3 Test Requirements

To make sure the test would fulfil its purpose we set up a list of requirements. This is helpful to make sure it will be easy to use and gather the information needed. The requirements are described in table 4.1, and explanations of the requirements follow below.

#	**Criteria**
RQ 1	The test should be easy to distribute
RQ 2	The test should be intuitive to understand and answer
RQ 3	The data from the test should be easy to analyse
RQ 4	Users should feel safe answering the test
RQ 5	The test should be visual appealing

Table 4.1: Criteria for algorithm

RQ1 - Easy to Distribute

This requirement means it should be easy to distribute to the users, in order to make sure the test gets as many participants as possible. This is solved by making the search system to a website, where users from all over the world can access. This will hopefully lower the barrier users feel for answering a test like this, while we also do not have to deliver and overwatch the test personally.

The problem of distributing a test like this, is that there is no two-way communication. The information found from the test has to be sufficient for the user to understand and answer the questions.

RQ2 - Intuitive

This requirement directly follows RQ1. Since there is no two-way communication, the design has to be carefully created to be understandable for the user. If the user is not sure what to do or gets confused it can lead to the user not completing the test, or answering incorrectly and presenting noisy data. In order to achieve an intuitive test, the questions have to be easy to understand and the design has to be carefully crafted like explained in section 3.3.

To make sure the test was intuitive, it was tested on several people without any guidance. Places where the participants had to ask questions or got stuck, we redesigned the question or interface until several participants could complete the test without guidance.

RQ3 - Easy to Analyse

There are two parts of this requirement. The first is what information to collect and the second is how to store it. To make sure that all the data can be analysed quickly, all the information gathered is stored in a database. This way scripts can be made to analyse larger amounts of data in short time. All the data presented by the user will also be stored in a structured manner, so the information will be easier comparable. Therefore, we have chosen to only store quantitative information in the database. This way no researchers has to read through answers to find information or patterns. This could be beneficial over information stored on paper or in an unorganized form, where the researcher first has to structure the data.

RQ4 - User trust

In order for users to answer the test, they should feel it is trustworthy. This is done by making a simple and clean design, and making sure the users are anonymous like described in 4.1. It is also made sure that the test will spread in a way that feels secure, and that the researchers themselves send out the request for testers. The URL for our website are also made a human readable URL, opposed to an IP address. This can also make the test seem more trustworthy

RQ5 - Visually Appealing

This requirement builds up under RQ4 and can help to make the user trust the website. This also makes it more plausible that the user will finish the test and answer correctly [30] . Even though no interaction designers were used to create the system, we made sure to keep the design simple, and follow the design of similar systems (like Google Play). The thought is that this would make it easier for the user to feel familiar with the system, as well as making it more desirable for the user to answer.

4.4 Test Flow

The finalized test consists of eight question pages. We tried to make as few questions as needed to cover our test, and since some questions are rather time-consuming, we felt that we did not want more than eight tasks.

First, the test candidates are presented with a page where they can register a user if this is their first time entering, or log in if they want to continue from where they left the test last time. Here they also find the information they need regarding anonymity and handling of data. They are also presented with some general information about the test, as well as guidance to how they should answer. Since different persons can have different thoughts about what it means for an app to be interesting, we stated some guidelines in order to minimize these differences.

1. An app is considered likeable if the user might have downloaded this app or similar apps.
2. The relevance of an app should only be based on user preferences and not relevance to the query provided.

3. The user should not consider whether the app would work on their given device, as this is not something we took into consideration when recommending apps to the user.

After the users have read this information they can register a user. Here they will be given a random username they can keep, or create a new one. When the user is satisfied with their name and password they can register the user.

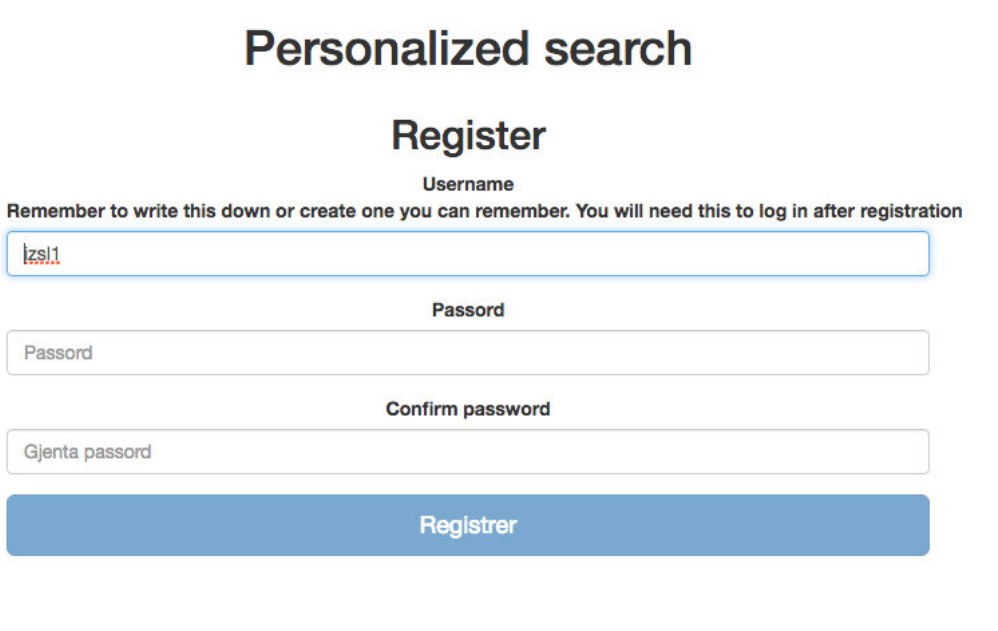

Figure 4.1: Register page with random username

After the registration, the user is directly sent to the login page where they can log in with their newly created credentials. The user is also given feedback that the user is created and that they have to fill in the fields. This is to help the user get started, and make sure the user understands the process.

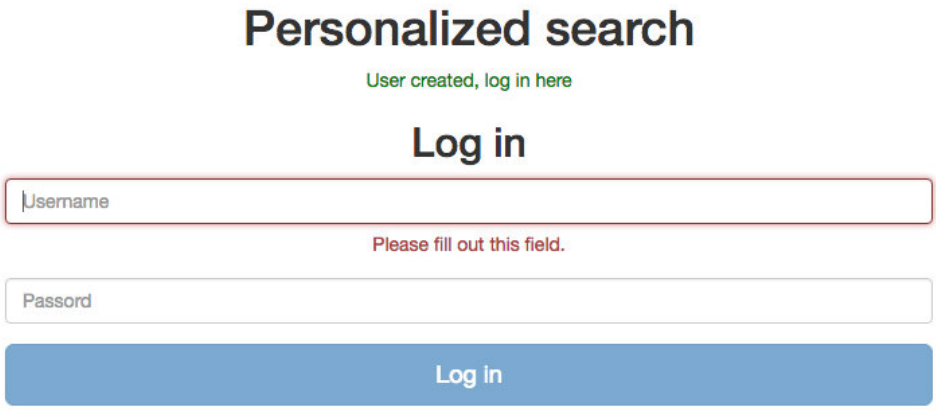

Figure 4.2: Login page

4.4.1 Building a User Profile

The user is then presented with 20 different apps where they will answer which apps they like and which they do not like. This is to gather information and build a profile for the users' preference. The information gathered in this step, is stored and used to build the Naive Bayesian classifier and preference for terms like described in section 3.2.3. If we had an app store that could run over a longer period of time we could have gathered this same information from app downloads or usage, but because of the short span and lack of user information, this needs to be done explicitly by questions. Even though answers we collect should be able to capture the user's preferences, only using explicit information can have some problem with accuracy as mentioned in section 2.3

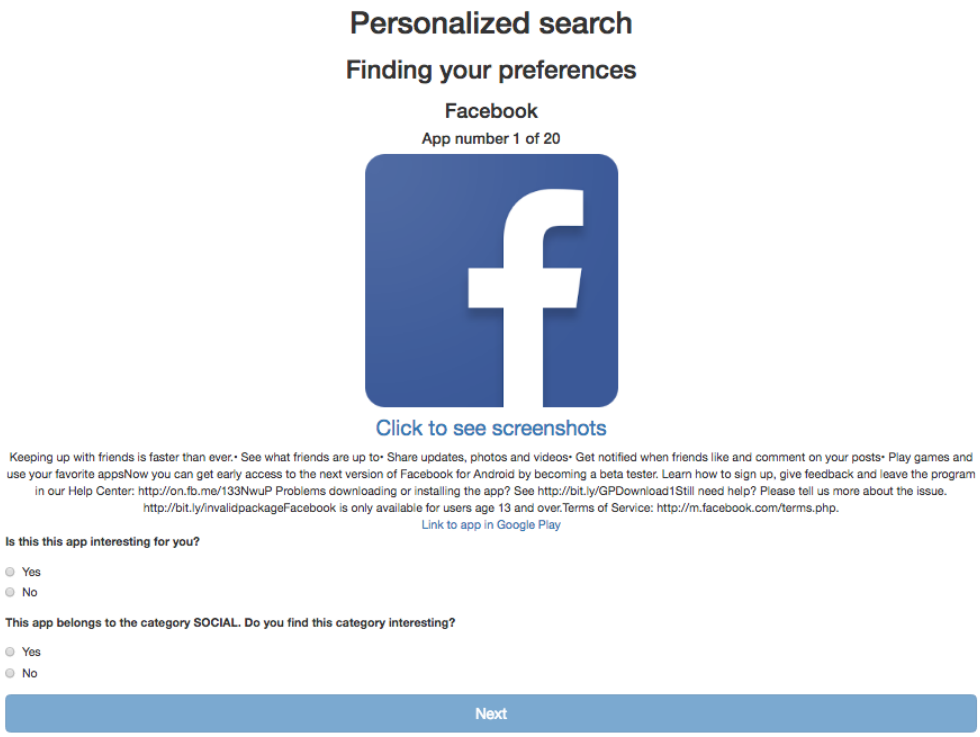

Figure 4.3: App presented to build user profile

4.4.2 Task 1 and 2

After the users' have told their preference for all the 20 apps, they will be presented with the first test. They are told to find an app they like. This is an open question, and the thought behind this is to see whether the user will use the suggestions without being told anything about how the system works. The user is not told anything about the suggestions or how to insert their query.

This task uses the Bayesian classification method for instant suggestions and the normal search for searching. When the user finds an app he likes, he can click that the app is interesting and the first task is done.

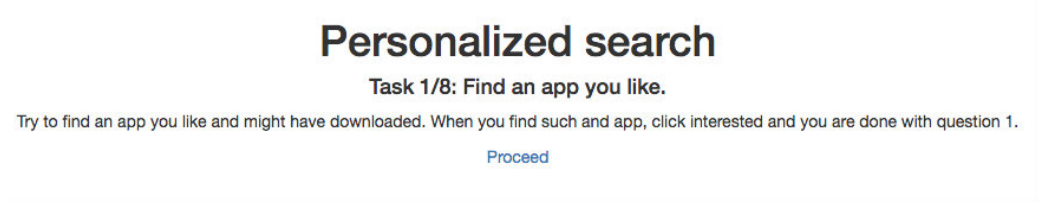

Figure 4.4: Task 1

The second task is similar to the first one. It asks the user to find another app he likes, but not the same as in task 1. This time, we use the KL method for app suggestions and retrieve the most popular apps as search results. By using both methods on similar tasks, it is easier to compare the different suggestion techniques to see what method performs best.

4.4.3 Task 3 and 4

At task 3, we make the user aware of the instant suggestions, and tell them to use their favourite features to find apps they like. It is given an example of features to make sure the user understands how to generate the query. They are also told that this time they are not only to find one app, but tell their preferences for all the apps presented. Again the user arrives at the search bar, where we use Bayesian classification to provide suggestions. When the user either clicks a suggestion and tell their preference for that app, or press the search button, they are presented with the search result to their query four times. This time, we use all the implemented search algorithms to present results to the user. Each algorithm returns 10 apps, and the user tells their preference for the given apps. This is to see how well our algorithm for app suggestion retrieves apps compared to non-personalized baseline methods.

Even though the suggestions in this task will not help the user complete the task quicker, as they have to rate results from the different methods anyway, they are told to click the suggestion if they find an app they like. This is to make sure users do not skip this step, even if it is not particularly helpful in completing the task. We need as much data as possible for how well the suggestions work. The reason the user is to insert features to their query, is to check if the methods perform better when the user search for apps based on features than searching for app names directly.

Task 4 is the same as task 3, except the user is told to use different features and the KL method is used for retrieving suggestions.

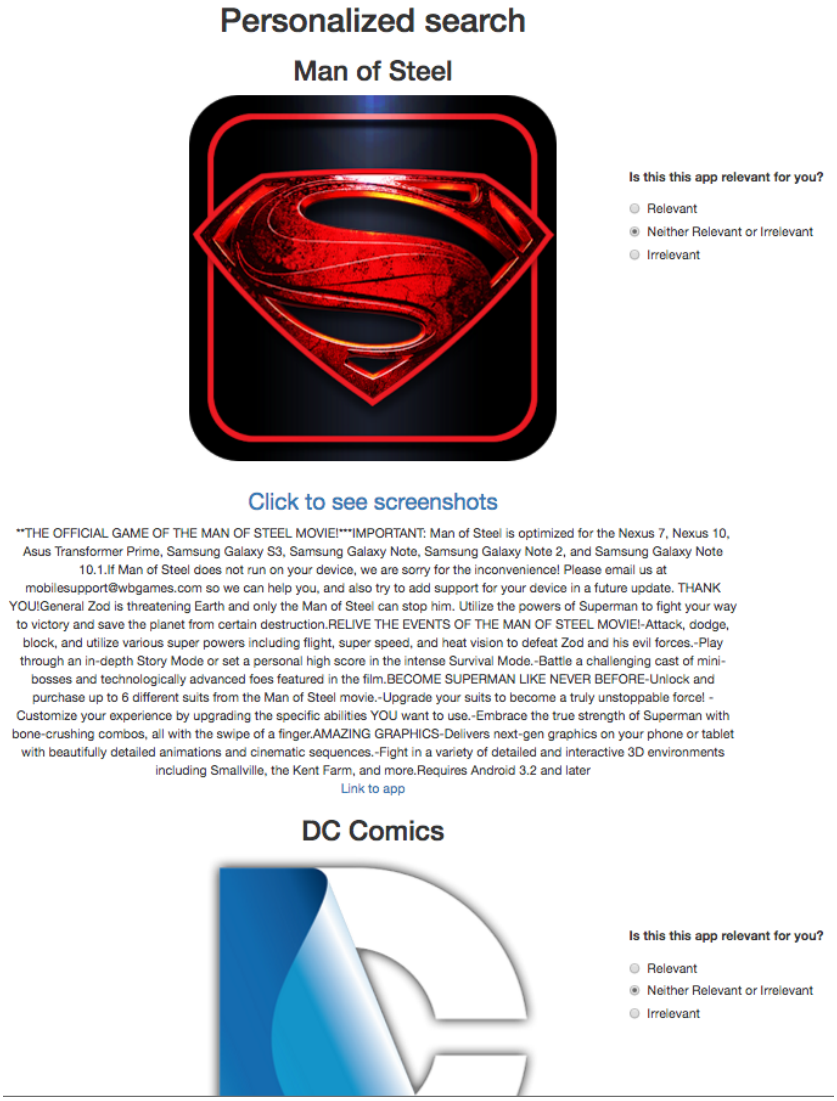

Figure 4.5: Search page showing multiple apps, the participant will then go though each app and tell their preference for that app

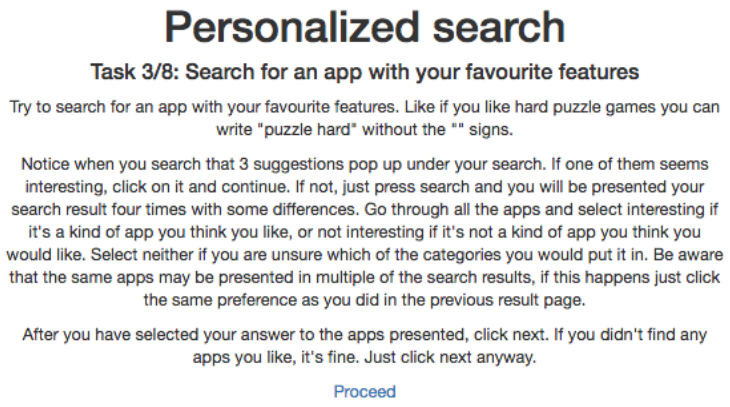

Figure 4.6: Task 3

4.4.4 Task 5

Task 5 asks the user to find a specific app they know of before, that they like. This could be an app they found previously in the test or another app they know off.

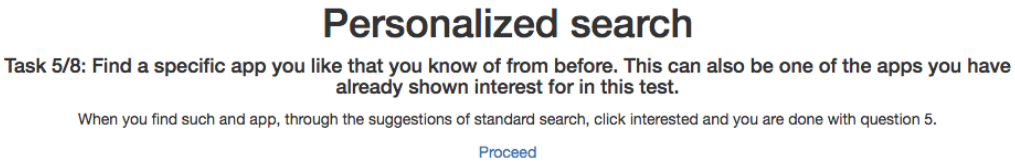

Figure 4.7: Task 5

This task is meant to track how well the suggestion works when looking for specific apps. It is believed that the suggestions will not work as well when the user is searching for a specific app, since the suggestion techniques try to search for apps beyond the query. If users don't use the suggestion in this task it will strengthen our belief, if multiple users use the suggestion for this use case, our hypothesis might be wrong and the suggestion might work for use cases like this.

4.4.5 Task 6 and 7

Now that the user is familiar with suggestions and the search engine, they are again asked to find an app they like by using app features and not searching for a specific app title.

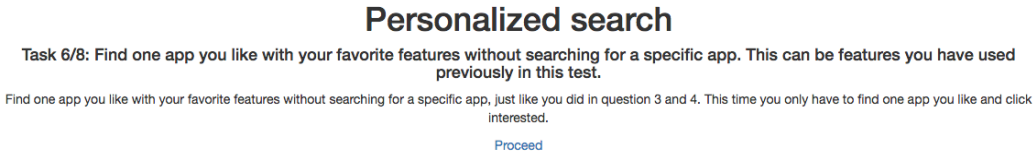

Figure 4.8: Task 6

This is similar to task 3 and 4, except this time they are done with the task when they find an app they like. The goal of these tasks, is to see how the user interacts with the system now that they are more familiar with it. Just like when they would search for an app in another app store, the task would be done when they find an app to download. Here we will see if the participants use the suggestions or the standard search to fulfil the task. Again we use both the personalized methods to be able to compare them against each other. Task 6 user the KL method for app suggestions and task 7 use the Bayesian classification method to suggest apps.

4.4.6 Task 8

In task 8 the user is asked different general questions. These questions are to categorize users into gender, age and experience with app stores from before. Then there are 4 multiple choice questions about their experience with the search engine. These questions ask how quickly they felt the suggestions would show up, if the suggestions made them reformulate the query, how helpful they felt the suggestions were and how the suggestions helped them the most (if they helped at all).

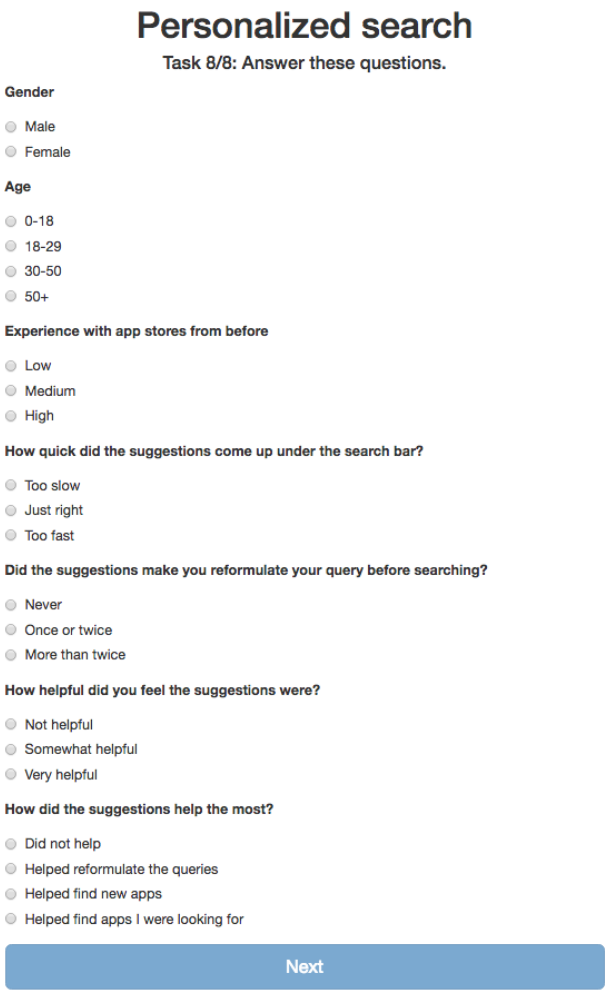

Figure 4.9: Task 8

These questions are used to gather information that is hard to track automatically, and the information is easier for us to gain when asked explicitly. This information can also be used together with the implicit information gathered during the test to check for validity. Even though users might like multiple suggestions found, it is important for the user experience that they feel the suggestions are helpful as well.

4.5 Gathering Information

Now that we have discussed the test flow, it is time to see decisions made when presenting the questions and how the information is gathered.

4.5.1 Binary Versus 5-Points Likert Scale

When deciding how to gather information from users, we considered two methods. Binary questions and the 5-points Likert-scale. These methods have different pros and cons, where binary shows higher inter-rater reliability and Likert scale shows good agreement and construct validity [31]. Both of these methods could work well for our test, but we decided upon the Binary questions. There were three reasons for this:

1. It is easier to implement
2. It is less time consuming for the user
3. It is hard for users to decide how well they like an app

The fact that users find it hard to evaluate exactly how well they like an app is found with evaluation of systems using the 5-points Likert-scale, where most users rate the app from one or five [4]. Usually, they rate the app as a five if they like it or it works, and as a one if they do not like the app or it does not work. Because of this, it was decided that a user probably would not know their preferences well enough to give accurate answers to how well they like the app. Especially in cases where the participant has not tried the app in advance. As using the 5-points Likert scale could present noisy data, it was discarded. When presented with 10 apps from the different search algorithms the user has a third choice, that they neither like nor dislike the app. This was because we experienced from the five first tests we tried, that since the participant is presented with several apps, they had a hard time evaluating all the apps. To speed up the process for the user, and make sure they would not just click a random value, we made the option for the user to tell they were unsure.

For the last task, where we ask the user questions about themselves and their thoughts about the system, we also decided to have multiple choice questions. This was to make it easier to categorize the data, and also to make sure the process of evaluating all the data would go quicker. Because of this, we ended up with a completely quantitative test.

4.5.2 Tracking User Behaviour

To make sure we get the most accurate data we can, we also make sure to track implicit data in addition to the explicit information given by the user. From the domain of personalized solutions, it is known that a combination of implicit and explicit information usually give the most accurate results [32]. The way we gather this information is tracking how many times the user clicks the suggestions, what index the different apps the user likes are, and which methods the users are using to find the apps they liked. This makes it easier to capture the actual user behaviour instead of the users' perceived behaviour of how they interacted with the system. We do not manage to track how many times the user reformulated their query before performing the search, so this information is gathered explicitly in the end.

This combination of explicit and implicit information can also be used to validate the data gathered. If the user says they did not use the suggestions, but the system have tracked they found multiple apps they liked through these suggestions, there might be reasons to question the validity of these answers and visa versa.

Chapter 5

Results

In this chapter, we present the data gathered from the test conducted. Section 5.1 presents data about number of users participating in the test, and characteristics about them. Section 5.2 shows data from comparing the search results of the different methods implemented. In section 5.3 we show results from how the suggestions were used by the participants. Section 5.4 presents the data from how the participants experienced the test.

5.1 Population

In total 67 users participated in the test. 21 of these participants aborted during the test, leaving 46 users completing the test.

Information regarding age, gender and experience can be found in figures 5.1, 5.2 and tables 5.1, 5.2 and 5.3. Most of the participants were between 19-29 years old, and since there are a limited amount of users in the other age groups, it will be hard to find information regarding how different age groups make use of the personalized suggestion.

As can be seen in table 5.2, the gender of the participants is about equally split, where 22 are female and 24 are male. Meaning the data basis from both gender is about equal.

When looking on the users' experience with app stores from before, we see that most users have medium to high experience. Only 4 participants have little experience with app stores from before, this means most of our users are familiar with the search methods already present in these stores from before.

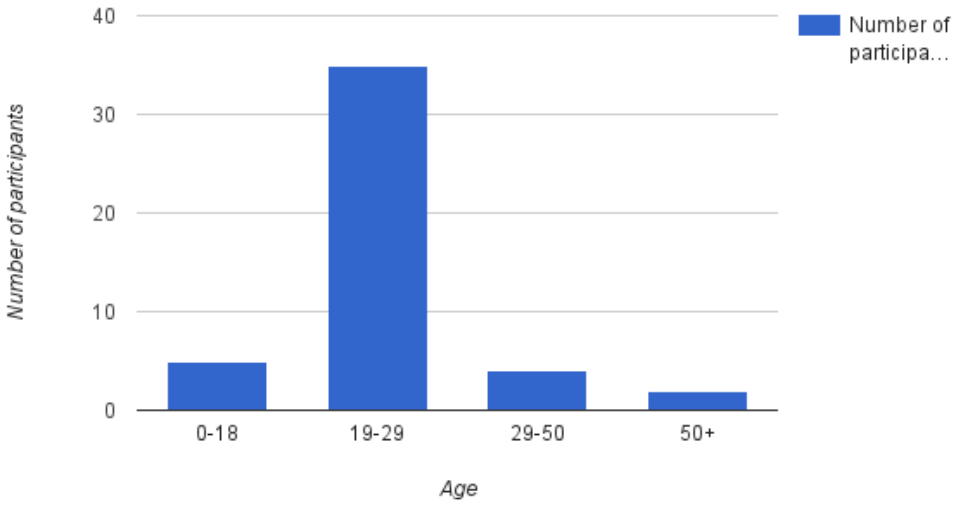

Figure 5.1: Age of participants

Figure 5.2: Gender and experience with app stores from before

Age	Number of participants
0-18	5
19-29	35
29-50	4
50+	2

Table 5.1: Age of participants

Gender	Number of participants
Female	22
Male	24

Table 5.2: Gender of participants

Experience	Number of participants
Low	4
Medium	19
High	23

Table 5.3: Participants' experience with app stores from before

5.2 Search Comparison Results

From task 3 and 4, we compared results from the different search algorithms. The participants were asked to rate all the results, in order to see which method would retrieve the highest number of relevant results. The number of results retrieved from the different methods varies some, this is because the algorithms will at most return 10 apps, but can sometimes retrieve fewer apps. The data from number of search results retrieved and which of these were considered relevant can be found in figure 5.3 and table 5.4

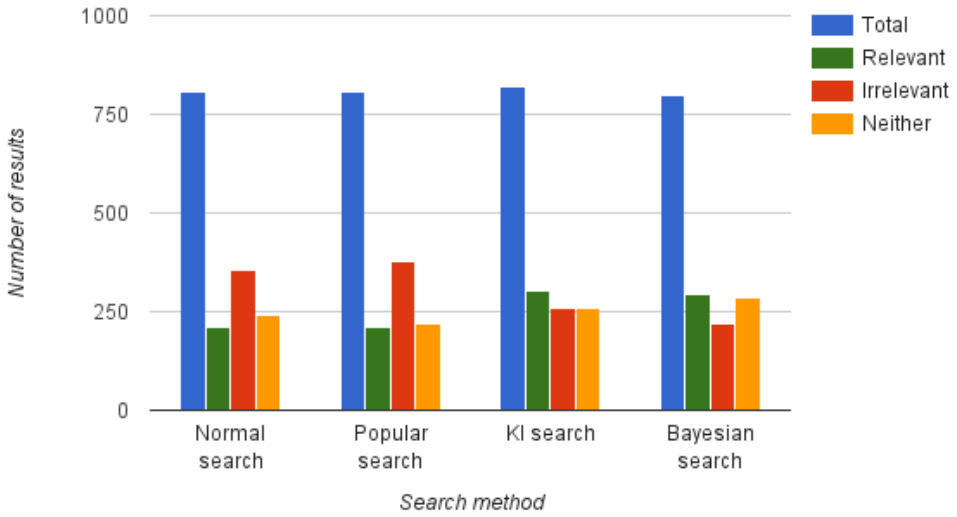

Figure 5.3: Number of results rated as relevant, irrelevant and neither by the user

Search method	Number of results
Normal search total retrieved	807
Normal search relevant	211
Normal search irrelevant	354
Normal search neither	242
Popular search total retrieved	807
Popular search relevant	210
Popular search irrelevant	377
Popular search neither	220
Kl search total retrieved	820
Kl search relevant	303
Kl search irrelevant	257
Kl search neither	260
Bayesian search total retrieved	797
Bayesian search relevant	292
Bayesian search irrelevant	221
Bayesian search neither	284

Table 5.4: Comparisons between search methods

Figure 5.4 and table 5.5 shows the fraction of retrieved apps relevant for the users. We can see that both normal search and popular search score about the same with 25% of apps being relevant. As we can see normal search had fewer irrelevant results, due to the fact that more apps were categorized as neither relevant nor irrelevant, and therefore performed a little better than popular results.

When looking at the two personalized search methods implemented, we see that both of these methods outperforms the two baseline methods. They have more relevant results and fewer irrelevant results. We also see that the two methods perform almost the same, where the KL-method has slightly more relevant results, and the Bayesian method has fewer irrelevant results.

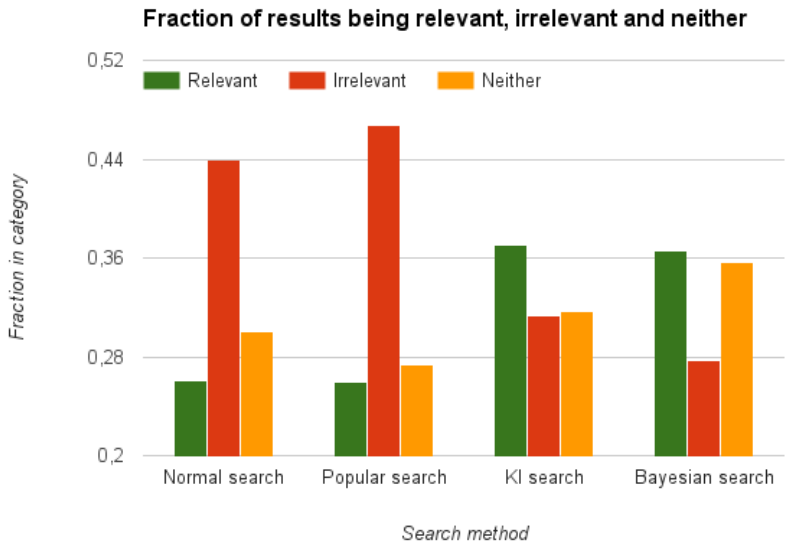

Figure 5.4: Fraction of results rated as relevant, irrelevant and neither by the users

Search method	Fraction of results
Normal search relevant	0.261
Normal search irrelevant	0.439
Normal search neither	0.300
Popular search relevant	0.260
Popular search irrelevant	0.467
Popular search neither	0.273
Kl search relevant	0.370
Kl search irrelevant	0.313
Kl search neither	0.317
Bayesian search relevant	0.366
Bayesian search irrelevant	0.277
Bayesian search neither	0.356

Table 5.5: Fraction of results retrieved where relevant

Another important factor, as well as number of relevant results returned, is how highly ranked the relevant results are. Since the personalized suggestion will typically only show somewhere between 1-6 results, if looking on how Google Play presents suggestions, it is important that the relevant results are shown in the first few retrieved apps. Figure 5.5 and table 5.6 show Mean reciprocal Rank (MRR) for the results retrieved with the different search methods. MRR is a statistic measure that finds the first relevant results for each query, and finds an average for these reciprocal ranks. This score can be used to evaluate how well a search engine ranks the relevant results, or at least how well it ranks the first relevant result from each query.

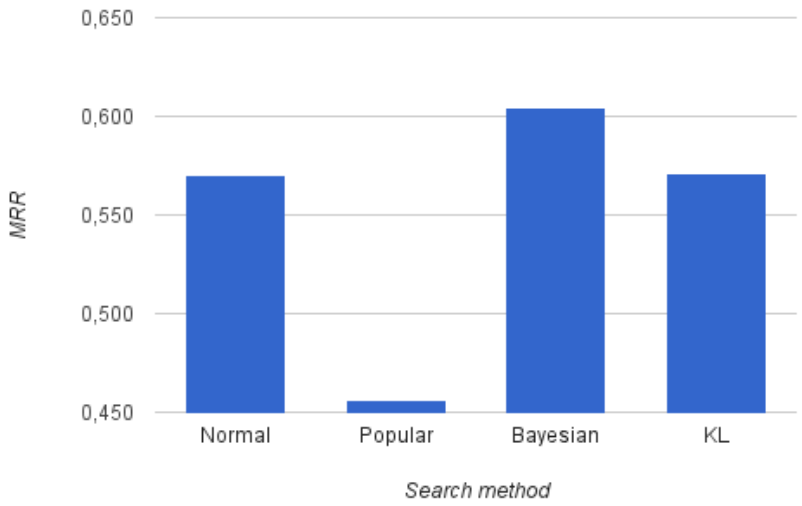

Figure 5.5: MRR for the different search methods

Search method	MRR
Normal search	0.570
Popular search	0.456
Bayesian search	0.604
Kl search	0.571

Table 5.6: MRR for search results

The results show that our Bayesian search outperforms the other methods, while KL and Normal search performs almost equally well. Popular search has the lowest MRR.

5.3 Suggestion Results

This section will present data gathered about the suggestions provided by the KL method and the Bayesian method.

Method comparison

Table 5.7 shows how often each user on average clicked the suggestions through all the tasks in the test, how often users liked the suggestions presented and how often the clicked results were relevant to the users. The test in total contains four tasks where the Bayesian method is used to suggest apps, and only three tasks where the KL method is used. It is therefore chosen not to take into consideration task 5, where the user is to look for a specific app, when calculating the scores for the suggestion in this test. By removing this task, when comparing data we get the same number of answers for both methods, and can compare tasks where the user performs similar tasks.

We can see that the suggestions from our Bayesian method seem to perform a little better, with more clicks and more liked apps being suggested. The KL method scores a little better on fraction of clicked results relevant to the user. This data also shows that the most common method to complete a task was by using the personalized suggestions, where they on average helped complete the task in 3.67 out of the 6 tasks presented to them. This means that more times than not, the user found an app they like with the help of the instant personalized suggestions.

Kl method clicked suggestions per user	2.33
Kl method liked suggestion per user	1.78
Kl method fraction of clicked suggestions relevant	0.766
Fraction of tasks using Kl method performed with suggestion	0.593
Bayesian method clicked suggestions per user	2.48
Bayesian method liked suggestion per user	1.89
Bayesian method fraction of clicked suggestions relevant	0.763
Fraction of tasks using Bayesian method performed with suggestion	0.630
Total clicked suggestions per user	4.80
Total liked suggestion per user	3.67
Total fraction of clicked suggestions relevant	0.765
Fraction of tasks in total performed with suggestion	0.612

Table 5.7: Comparison between suggestion methods

MRR of Suggestions

Figure 5.6 and table 5.8 show the MRR for the personalized search methods when suggesting apps to the user. We can see that the MRR score for the

Bayesian method is higher than the score for the KL method.

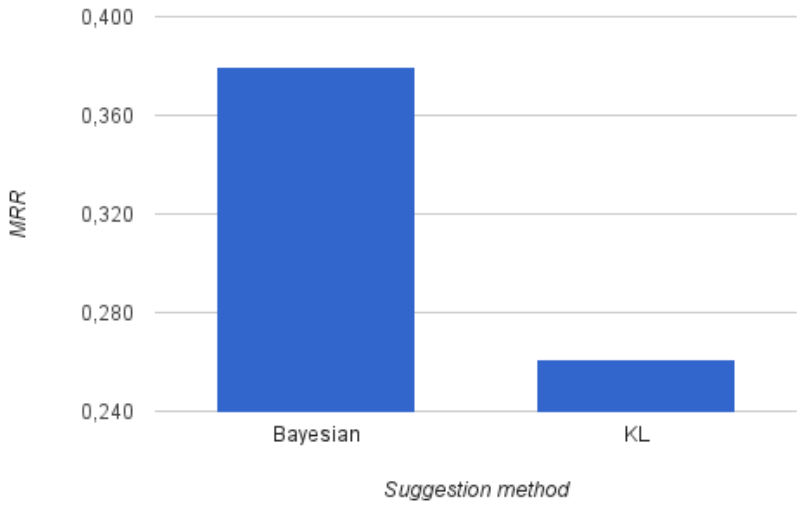

Figure 5.6: MRR for the different suggestion methods

Suggestion method	MRR
Bayesian method	0.380
Kl method	0.261

Table 5.8: MRR of suggestions

Suggestion Performance Grouped by Tasks

Lastly, we look at data gathered for the different tasks performed by the participant. Table 5.9 show the results for how the participants used the instant suggestions, grouped up in the different tasks. This information can be used to see what tasks the suggestions seems to work best for, and also what methods performs best in these tasks. We see that the performance on the different tasks is relatively similar, except from task 4 and 5. Both the number of clicks on suggestions and the number of relevant suggestions are similar to the other tasks.

Task 1	
Total clicked	37
Total relevant	31
Fraction of clicked relevant	0.838
Task 2	
Total clicked	37
Total relevant	31
Fraction of clicked relevant	0.838
Task 3	
Total clicked	39
Total relevant	28
Fraction of clicked relevant	0.718
Task 4	
Total clicked	34
Total relevant	20
Fraction of clicked relevant	0.588
Task 5	
Total clicked	29
Total relevant	26
Fraction of clicked relevant	0.588
Task 6	
Total clicked	36
Total relevant	31
Fraction of clicked relevant	0.861
Task 7	
Total clicked	38
Total relevant	28
Fraction of clicked relevant	0.737

Table 5.9: Clicked and relevant suggestions for the different tasks

5.4 User Feedback

Feedback from task 8 is given in this section. These questions are meant to track the users' experience after the test and asking questions gathering data we were not able to track.

How Fast the Suggestions Showed

The first question asks the user how fast they felt the suggestions showed up. The results are given in figure 5.7 and table 5.10. We see that almost all the participants felt the suggestions were retrieved quickly enough.

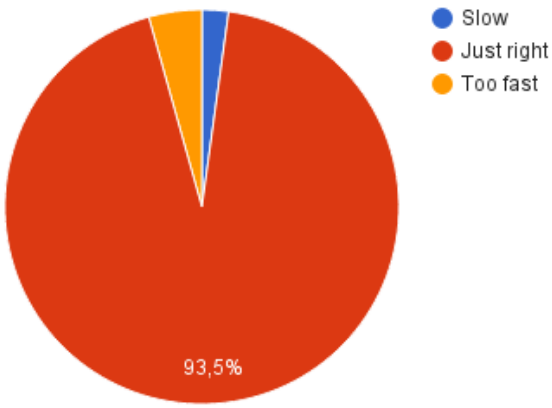

Figure 5.7: Answers to how fast the users felt the suggestion showed up

Answer	Participants
Slow	1
Just right	43
Too fast	2

Table 5.10: How quick the users felt the suggestions were retrieved

Reformulating Queries

The second question asked how often retrieved suggestions made the user reformulate their query. Data gathered from these answers can be found in figure 5.8 and table 5.11. We see that most of the participants said they reformulated queries either once, twice or more because of the suggestions.

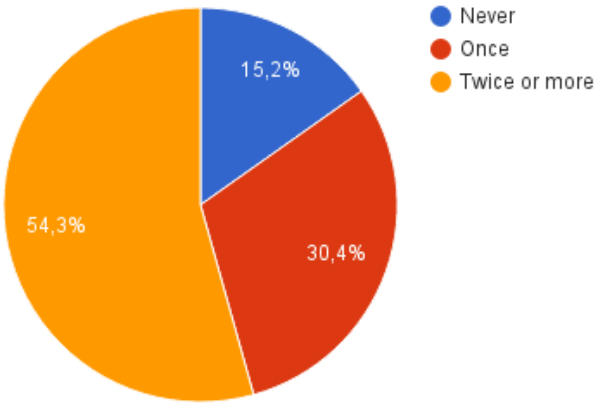

Figure 5.8: Answers to how often the suggestions made the participants reformulate their query

Answer	Participants
Never	7
Once	14
Twice or more	25

Table 5.11: How often users felt they reformulated queries because of the suggestions retrieved

Helpfulness of Suggestions

The third question asks how helpful the participants felt the suggestions were. This could show if the users felt the suggestions were more or less helpful than the data gathered about their usage patterns. The results are shown in figure 5.9 and table 5.12. We see that most of the participants felt the suggestions were helpful in performing the task. Where most of the participants answered that the suggestions were somewhat helpful.

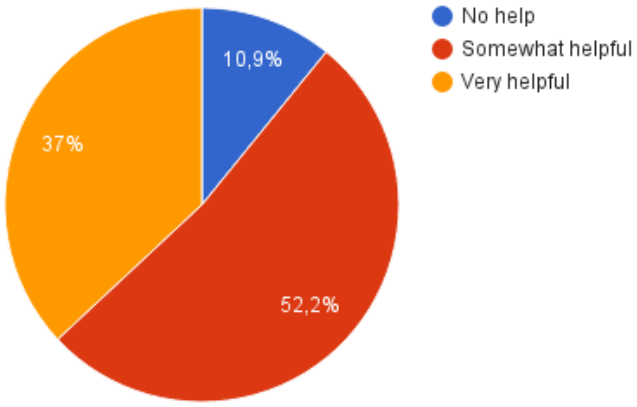

Figure 5.9: Answers to how helpful the participants felt the suggestions were

Answer	Participants
No help	5
Somewhat helpful	24
Very helpful	17

Table 5.12: How helpful the suggestions were

How the Suggestions Helped

This last question asked the participants how the suggestions helped them the most, if it helped at all. This way we can see what use cases personalized suggestions could have. The answers are given in figure 5.10 and table 5.13. We can see that the majority of participants felt the suggestions were most helpful in finding new apps or to reformulate their query.

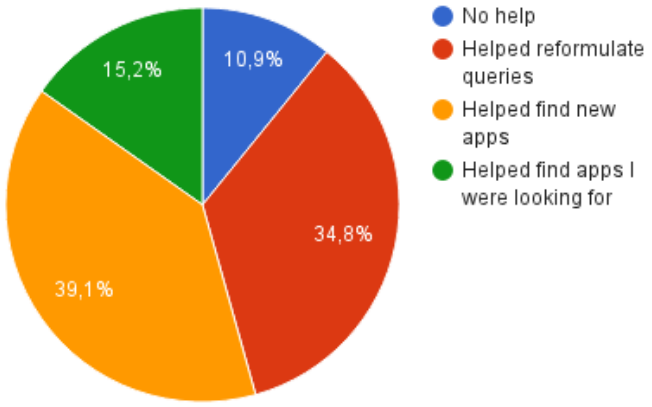

Figure 5.10: Answers to how the participants felt the suggestions helped them the most

Answer	Participants
No help	5
Helped reformulate	16
Helped find new apps	18
Helped find apps I were looking for	7

Table 5.13: Answers to how the participants felt the suggestions helped them the most

Chapter 6

Discussion

In this chapter, we discuss the results from our test. Section 6.1 discuss data basis and the validity of the data gathered. Section 6.2 discuss the results found from comparing the different search methods. Section 6.3 investigate the data found when comparing the instant suggestions. Section 6.4 discuss the findings for how users completed different tasks. In section 6.5 we analyse the data gathered from the user feedback in task 8. 6.6 evaluates our work by seeing how the work answers the research questions stated. The chapter ends with section 6.7 where we discuss the limitations of our test and findings.

6.1 Data Basis

As we saw in the results, number of participants completing the test is 46. This number is not enough to give representative information for the users of app stores or online stores in general [33]. As of now, Google Play has over 1 billion active users each month [34].

In other words, the information gathered is not enough to say whether our solution is efficient or inefficient in suggesting products to users in general. That being sad, the test conducted is not meant to answer these questions, but to see if the proof of concept would be interesting for further development and research. To this means, usually fewer respondents are being used to discover weaknesses and interest for proof of concept. Macefield, R [35] suggests that when testing concepts and prototypes, fever participants are required. The research argues that for problem discoveries, number of participants should vary between 3-20. Depending on complexity and criticality, this number

could vary. Since the test conducted is meant to find problems and test validity of the concept, as well as being a complex problem, it could be reasonable to believe that the number of participants required should be a little higher than suggested in this paper. This leads us to believe that a participant base of 46 users should be enough to test whether the concept is interesting and should be researched further.

Even though we have gathered information about gender, age groups and experience with app stores, we have chosen not to investigate differences in use patterns in these groups. This is both because the data basis on some of the subgroups is rather small, and because this is not seen as important to find the usefulness of instant personalized suggestions in general. Information about the participants might be interesting to check the diversity of our respondents. Some results gathered by grouping participants by age, gender and experience with app stores from before can be found in appendix B.

6.2 Comparing Search Results

When comparing the different search algorithms, the best improvement we see by using the personalized methods compared to the baseline methods, is the number of relevant results they retrieve. There is a gap between the performance for the baseline methods and the personalized methods, where the personalised methods retrieve relevant results about 37% of the time and the baseline methods retrieve relevant results about 26% of the time.

Another important factor is how highly ranked these relevant results are. When suggesting apps to the users it is important that the method retrieving suggestions is able to show relevant results to the user in the first few apps, as these are the suggestions the user will see. In Google Play, we see that they suggest around 0-6 queries for the user, and if none of these suggestions are relevant it does not matter how many relevant results the methods are able to retrieve. The MRR for the different designs show that the Bayesian methods seem to rank the results best, with a score of 0.604.

From the data gathered it seems that both the KL and Bayesian method outperform the baseline methods, were the Bayesian method perform better because of how highly ranked the relevant results are. It is also interesting that retrieving the most popular apps seemed to perform the worst, and just doing a simple text search performed better. As we know, showing the most popular apps are used both in Google Play and App Store. The reason this

method perform worse might also be the way it is implemented, and using other metrics for retrieving most popular apps might have improved the score.

6.3 Comparing Suggestion Results

The first thing to note is that the Bayesian method seems to slightly outperform the KL method, when looking at the performance of the suggestions retrieved. This same result we got when comparing results for the search results retrieved. The fact that these results correlate is to be expected, since suggestions retrieved are found in the same way as the search results are. This data also strengthen the reliability of the data, as data for suggestions and search show similar results.

Even though it seems like the two personalized methods perform almost equally well, the Bayesian method has slightly more clicks and more relevant suggestions. When looking at the MRR from the suggestion retrieved, we also see that the Bayesian method scores better. Even though the most important aspect is that one of the suggestions shown is relevant, MRR gives an indication of overall performance. If fewer suggestions were to be shown, it is important that the relevant suggestion is highly ranked among the four shown. Our data does not clearly state that one method is superior to the other, but show indications of the Bayesian method performing better. There might be multiple reasons for this. One reason might be that the Bayesian method return suggestions more closely related to the query. The KL method looks deeper into the user preferences when suggesting apps. Sometimes this might not fit what the user is looking for at the current time. Another thing to note, in the favour of the Bayesian method, is that the results are also retrieved quicker than with the KL method.

When looking at the total performance for both of these methods, it seems like the suggestions were commonly used by the participants. Over 50% of the time, participants found an app they liked with the help of suggestions. We see that even though around 37% of all the apps retrieved on the search pages are relevant to the user, the top four apps suggested were relevant on average 61% of the times with both methods. Because of how highly used the suggestions are, it can seem like the concept could be beneficial for users. If we look at other work investigating the use of query suggestion [36], we see that their experiments show users would use the more traditional query suggestions under 50% of the time. Even though this is not directly comparable as we count number of tasks completed, not number of queries presented, it gives an

indication of how our system compare to more traditional queries. Another thing to note is that the suggestions in their study do not necessarily lead to relevant results,as they only present a query the participant can use to look for relevant results, while all the liked suggestions in our study helped the user directly find the product they like. Data on how often instant product suggestions are relevant in systems today were not found, but having theses suggestions seems to be beneficial over no suggestion or suggestions using any of the baseline methods.

6.4 Suggestion Results Grouped by Tasks

When we look at how the participants used suggestions on the different tasks, we see that the performance was about the same, except from task 4 and 5. Why the suggestions on task 4 performed worse than the tasks in general, is hard to understand. The task is pretty much the same as task 3, and use the same method (KL) as task 2 and 6. One reason might be that the user knows from task 3 that they have to evaluate apps on the search pages, even if they find an app by clicking a suggestion, and therefore might want to skip this extra step.

Looking at task 5 strengthens our hypothesis that the suggestions would be less relevant when the user looks for a specific app. We see that this task has the lowest number of suggestions clicked. This is probably because the suggestions are looking for apps beyond the query, to find items the user likes. Therefore, it might find other apps the user likes, as opposed to a specific app the user looks for. This being said, it did perform better than expected. For this task, we used the Bayesian classifier, and we see that the suggestions helped users to find the app they were looking for 57% of the time. This is only 6% lower than what the classifier did on the other tasks. This indicates that the suggestions do not perform too poorly in situations where the user is already aware of what they want.

It is also interesting to see that the personalized suggestions worked well in the two first tasks. Here the users had no restrictions on how to present their queries, and they were not told anything about the suggestions. These are in fact the tasks with the best performance, except for task 6. It does not seem to be any positive effect after task 3 where the user is told about the suggestions and asked to click them if the suggestions are relevant. This is a positive result for the personalized suggestions, as this indicate that the search engines works well for users without any guidance and when the user

writes queries natural to them.

6.5 Analysing User Feedback

When testing out the different methods under development, we saw that the suggestions would generally show up somewhere between 150 - 600 ms with KL and 100-250 ms with Bayesian. This is higher than the requirement of 100 ms for a suggestion to be considered instant. Since we try to minimize retrieval time of suggestions, with the methods described in section 3.5.3, it is interesting to see how quickly the participants felt suggestions were retrieved. We see from the data gathered that only 1 participant felt the suggestions were retrieved too slow, while 43 users felt the time before suggestions were retrieved felt just right. These results suggest that even if systems are not able to implement personalized suggestions that are retrieved in under 100 ms consistently, by using other techniques they might be retrieved fast enough for the user.

When looking on how the participant used the suggestions, we see that most users reformulated their query at least once because of the suggestions retrieved. We also see that almost as many participants felt the system were most helpful in finding new apps, as those that felt the suggestions were most helpful in reformulating queries. This indicates that the personalized suggestions might have more use than just directly finding apps. Another thing to note, is that users in general, felt the system were most helpful in finding new apps. This means that a system utilizing instant personalized search can be especially helpful in increasing novelty. This is something which can prove useful in the app store domain where very few apps are in use, and search is mainly used the find specific apps the user is already aware of [4].

Lastly, we also see that most of the participants felt the system were somewhat or very helpful. This correlates well with the data gathered about number of times suggestions helped users find apps, and reformulate queries. It is a positive result for the search engine, that the participants also feel it is useful.

6.6 Evaluation of the Work

We will in this section go through each research question stated in section 1.2 and discuss our results.

RQ: **How can we create a search engine providing instant, personalized recommendations based on user preferences and query typed?**
In chapter 3 we explained how the system is designed and the different methods used. Testing shows that both methods are able to retrieve results not necessarily directly showing the query, and to find products based on the user's preferences. The working prototype created, is able to give personalized suggestions in under 600 ms for one method and under 250 ms for the other, including network delay. This prototype fulfilled all the criteria in table 3.1, except from the suggestion retrieval speed which is a little higher than 100ms. There is implemented some techniques which usually speeds up this process considerably, and keeps the delay in under 100ms.

*RQ*1: **How will the relevance of the retrieved results from our personalized methods compare to traditional search methods used today?**
We compared our search algorithms with traditional text search and retrieving most popular results. The results show that both the implemented solutions outperformed the standard methods. Both of the personalized methods retrieved more relevant results, and the Bayesian methods had a larger improvement in MRR as well.

*RQ*2: **How will users use instant personalized search?**
By looking at the implicit information we gathered from the test, it can be seen that the suggestions were used by the participants in most cases when completing the tasks. Which means more often than not, the users clicked the suggestions and were satisfied with the results. From the explicit answers we get, we saw that most users found the system somewhat or very helpful when performing the tasks given. This indicates that users can use the suggestion to help them find apps they like. From the test, it seems like the suggestions are most useful in finding new apps.

A number of users also felt the suggestions helped them reformulate their query. Most users reformulated their query once, twice or more because of the suggestions shown. This is an interesting result which shows that users might get help from the instant suggestions in finding apps they like, even when the suggestions are not directly relevant.

*RQ*3: **Will the proposed personalized search help users quicker find relevant products in digital stores?**
As mentioned in RQ2, users more often than not clicked the suggestions to find relevant apps for their tasks. This proves that our concept has the ability to quickly guide the user to relevant products. It is also important to note that every time these suggestions are used instead of "regular query

suggestions", the user has to perform one less step in order to get to their apps, which again would save the user some time.

When multiple people use the suggestions to reformulate their queries, it means they might save the time it takes to first perform the search, go back and then reformulate. These results show that the instant personalized search engine has the potential to help users quicker find relevant apps.

*RQ*4**: How can we retrieve search recommendations in close to 100ms which is the requirement for instant suggestions?**
None of the methods were able to consistently retrieve results in 100ms or quicker. We see that the Bayesian method often retrieved results close to this, especially when the results were found before the user inserted the last character in their query. By prepossessing the data and performing early queries to the backend, we are able to often give results in about 100ms.

The KL method would still require more optimisation to fulfil the requirement for instant suggestions. Note that we have not done much work on making these methods as efficient as possible. Other storing techniques, caching and more efficient implementations could lower the retrieval time.

Even though these suggestions used more time than the requirement to be instant, almost all the participants felt the suggestions showed quickly enough.

6.7 Limitations

There are some limitations regarding the online test performed. One being that the honesty of the participants cannot be verified. Users might say suggestions are relevant or irrelevant regardless of what they actually feel. This can be in order to be "kind" and help the prototype look good or "evil" to make the prototype look worse. The survey was designed to minimize the amount of manual work while keeping the users anonymous, it is therefore hard to verify the validity of the answers provided. Another limitation of spreading an online test is that the researchers have no control over the participants attending. This makes it hard to gather participants in the different subgroups, and make sure that only users with no interest in answering untrustworthy will participate. The only control that can be done, is checking the validity of the subjective answered by the user and the data gathered about their usage of suggestions.

It is general harder to measure how correct subjective answers are, opposed

to objective answers. Some users might feel it is hard to identify how much the system helped them, and how they used the system [37]. These questions are probably better suited to capture the users feelings and experience, than the actual usage of the prototype. Another aspect to consider is the fact that users might also have different thoughts about what "helpfulness of the system" actually implies. The same goes with whether the suggestions helped in reformulating queries or not. These facts can decrease the data's validity.

This system is believed to be usable for digital stores in general. The test will best indicate effectiveness for the app store domain, and helpfulness in other domains may vary. Multiple factors can affect how well this prototype works. Such factors can be number of items, complexity of queries, how well structured the data is etc. This being said, it is believed that this test can give indications to whether it is worth testing the principle in other online stores as well.

Chapter 7

Conclusion and Future work

In this chapter, we conclude our work, summarize what has been achieved and suggest further work. Section 7.1 concludes our work and gather the findings found in this research. Section 7.2 suggests further work that can be done based on the finding in this thesis.

7.1 Conclusion

This research has investigated how to create an instant personalized search engine, where we had some suggestions for designs from previous work. The designs were implemented and tested. An important part of building this prototype was testing whether the concept of instant personalized suggestions would help users find items, and to see if it would provide any improvements from traditional methods used today. We made an online quantitative test for the prototype to gather data about user patterns, the users' experience after using the system and comparisons between our personalized methods against standard baseline methods.

We were able to create a prototype for a search engine providing instant personalized suggestions based on Bayesian classification and Kullman-Liebler divergence. Initial testing showed that these methods seemed to give relevant results, while not necessarily directly containing the query typed. The prototype fulfills all the criteria set, except that it can sometimes use more than 100ms to retrieve results to the user.

Data gathered from the test show promising results for instant personalized suggestions. The methods outperformed the baseline methods, and helped

users complete the tasks quicker. Most users also felt the suggestions were retrieved quick enough and were helpful in finding items. Participants found the suggestions particularly helpful in finding new apps they were unaware of beforehand, meaning the instant personalized suggestions can increase novelty in search engines today. The method based on Bayesian classification seems to retrieve results quicker and be more accurate in classifying what items are relevant than the KL method. As we can see from the results for how the users completed the different tasks, we see that the suggestions work best on difficult tasks or in less concrete tasks. This is common for query suggestions in general.

The work conducted managed to create the personalized search engine, answering all the research questions. We have proven that instant personalized suggestions are an interesting concept that should be researched further. Even though the suggestions might be less accurate when finding specific products, they help in increasing novelty and have high accuracy when finding items relevant for the user. It also requires fewer steps for the user to find products they like.

7.2 Further Work

As the data from this research show positive results, there are multiple directions this work can be researched further.

- The work done in this research has only been using content-based information, because of limitations of the data gathered. Using the same concept with collaborative methods used in recommender systems today could give interesting results. Both to see if these methods can decrease time used to retrieve suggestions, and increase the accuracy of suggestions given to the users.
- Testing instant personalized suggestions in other digital stores than app stores could also prove interesting. This might show whether there are any performance differences in domains where data may be more or less structured.
- There were multiple classification techniques evaluated to use in this prototype. Trying other classification algorithms could provide positive results both for time constraints and accuracy. One method believed to be well suited is SVM, which was not tested in this research.

- Testing instant personalized suggestion where multiple sources of data can be used, together with user history, logs and implicit information, should also be investigated to see if this can improve the results.
- Optimizing time required for retrieval of items, by caching and other techniques, should be testing in order to see if suggestions could be consistently retrieved in 100ms or quicker.
- Testing with features generated by domain experts could improve the results. The features extracted from apps in this test were gathered from a general text about the product. Some digital stores have features about items given by the uploader of the product or people working at the store. By using this information, the features may be more accurate than the ones used in our test.
- As this research were only meant to test if our concept had potential or not, only a limited number of participants were used. After some optimizations, a large-scale test should be conducted to test with a significant number of participant. This can tell if such a system is beneficial in different domains.

Bibliography

[1] G. Madhu, D. A. Govardhan, and D. T. Rajinikanth, "Intelligent semantic web search engines: a brief survey," *arXiv preprint arXiv:1102.0831*, 2011.

[2] R. Baeza-Yates, C. Hurtado, and M. Mendoza, "Query recommendation using query logs in search engines," in *Current Trends in Database Technology-EDBT 2004 Workshops*. Springer, 2005, pp. 588–596.

[3] D. Kelly, K. Gyllstrom, and E. W. Bailey, "A comparison of query and term suggestion features for interactive searching," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2009, pp. 371–378.

[4] K. Shi and K. Ali, "Getjar mobile application recommendations with very sparse datasets," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 204–212.

[5] J. A. R. Rudihagen, "Instant, personalized search recommendations," Desember 2015.

[6] J. Hu, "Personalized web search by using learned user profiles in re-ranking," Ph.D. dissertation, Florida Institute of Technology, 2008.

[7] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *Internet Computing, IEEE*, vol. 7, no. 1, pp. 76–80, 2003.

[8] X. Amatriain, "Big & personal: data and models behind netflix recommendations," in *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*. ACM, 2013, pp. 1–6.

[9] N. Matthijs and F. Radlinski, "Personalizing web search using long term browsing history," in *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM, 2011, pp. 25–34.

[10] Z. Dou, R. Song, and J.-R. Wen, "A large-scale evaluation and analysis of personalized search strategies," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 581–590.

[11] D. J. L. Jaime Teevan, Susant T. Dumais, "To personalize or not to personalize: Modeling queries with variation in user intent," *Proceedings of SIGIR 2008*, 2008.

[12] A. L. Young and A. Quan-Haase, "Information revelation and internet privacy concerns on social network sites: a case study of facebook," in *Proceedings of the fourth international conference on Communities and technologies*. ACM, 2009, pp. 265–274.

[13] J. Schreier, "Sony hacked again; 25 million entertainment users' info at risk," 2011.

[14] M. S. Ackerman, L. F. Cranor, and J. Reagle, "Privacy in e-commerce: examining user scenarios and privacy preferences," in *Proceedings of the 1st ACM conference on Electronic commerce*. ACM, 1999, pp. 1–8.

[15] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin, "Fast data in the era of big data: Twitter's real-time related query suggestion architecture," *Sigmod 13*, 2013.

[16] R. L. Kumar, M. A. Smith, and S. Bannerjee, "User interface features influencing overall ease of use and personalization," *Information & Management*, vol. 41, no. 3, pp. 289–302, 2004.

[17] S. Han, G. W. Humphreys, and L. Chen, "Uniform connectedness and classical gestalt principles of perceptual grouping," *Perception & psychophysics*, vol. 61, no. 4, pp. 661–674, 1999.

[18] (2015, November). [Online]. Available: https://developers.google.com/events/io/2013/

[19] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender Systems An Introduction*. Cambridge University Press, 2011.

[20] R. Baeza-Yates and B. Ribeiro-Neto, *Moder Information Retrieval - the concepts and technology begind search*. Addison Wesley, 2011.

[21] C. Zhai and J. Lafferty, "Model-based feedback in the language modeling approach to information retrieval," in *Proceedings of the tenth interna-*

tional conference on Information and knowledge management. ACM, 2001, pp. 403–410.

[22] P. B. Kantor, L. Rokach, F. Ricci, and B. Shapira, *Recommender systems handbook*. Springer, 2011.

[23] K. Miyahara and M. J. Pazzani, "Collaborative filtering with the simple bayesian classifier," in *PRICAI 2000 Topics in Artificial Intelligence*. Springer, 2000, pp. 679–689.

[24] R. Ghani and A. Fano, "Building recommender systems using a knowledge base of product semantics," in *Proceedings of the Workshop on Recommendation and Personalization in ECommerce at the 2nd International Conference on Adaptive Hypermedia and Adaptive Web based Systems*, 2002, pp. 27–29.

[25] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[26] N. Zhong and F. Michahelles, "Google play is not a long tail market: an empirical analysis of app adoption on the google play app market," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 499–504.

[27] G. Rauch, *Smashing Node. js: JavaScript Everywhere*. John Wiley & Sons, 2012.

[28] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 2011, pp. 363–366.

[29] R. T.-W. Lo, B. He, and I. Ounis, "Automatically building a stopword list for an information retrieval system," in *Journal on Digital Information Management: Special Issue on the 5th Dutch-Belgian Information Retrieval Workshop (DIR)*, vol. 5. Citeseer, 2005, pp. 17–24.

[30] G. Iarossi, *The power of survey design: A user's guide for managing surveys, interpreting results, and influencing respondents*. World Bank Publications, 2006.

[31] Z. Awad, A. S. Taghi, P. Sethukumar, P. Ziprin, A. Darzi, and N. S. Tolley, "Binary versus 5-point likert scale in assessing otolaryngology trainees in endoscopic sinus surgery," *Otolaryngology–Head and Neck Surgery*, vol. 151, no. 1 suppl, pp. P113–P113, 2014.

[32] S. Lohr, "Netflix awards 1 million dollar prize and starts a new contest," *New York Times*, vol. 21, 2009.

[33] J. Kotrlik and C. Higgins, "Organizational research: Determining appropriate sample size in survey research appropriate sample size in survey research," *Information technology, learning, and performance journal*, vol. 19, no. 1, p. 43, 2001.

[34] [Online]. Available: https://events.google.com/io2016/

[35] R. Macefield, "How to specify the participant group size for usability studies: a practitioner's guide," *Journal of Usability Studies*, vol. 5, no. 1, pp. 34–45, 2009.

[36] X. Niu and D. Kelly, "The use of query suggestions during information search," *Information Processing & Management*, vol. 50, no. 1, pp. 218–234, 2014.

[37] W. T. Lin and B. B. Shao, "The relationship between user participation and system success: a simultaneous contingency approach," *Information & Management*, vol. 37, no. 6, pp. 283–295, 2000.

Appendices

Appendix A

Initial Testing

Below are detailed results from the tests conducted during implementations to optimize the system. We used 6 participants in the test, who told their preference for apps and categories. Each user told their preferences to 55 apps and 41 categories.

Mismatch Between Liked Category and App

The results in table A.1 shows how many apps our participants liked in categories disliked by the participant, and number of apps disliked in categories liked by the participant.

Category	Number of apps
Dislike category, but like app	52
Like category, but dislike app	189

Table A.1: Differences in apps liked and disliked based on whether the user likes or dislikes the category

Optimizing Classification Step

The table A.2 shows results from using the Bayesian classifier with different app features as input. We chose to use the KL-features as this is one of the two methods which scored best

Only description	
Apps correctly classified	221
Apps incorrectly classified	127
Fraction correctly classified	0.635
Description and name	
Apps correctly classified	221
Apps incorrectly classified	127
Fraction correctly classified	0.635
Description and category	
Apps correctly classified	203
Apps incorrectly classified	145
Fraction correctly classified	0.583
Only TF/IDF-features	
Apps correctly classified	226
Apps incorrectly classified	122
Fraction correctly classified	0.649
Only KL-features	
Apps correctly classified	226
Apps incorrectly classified	122
Fraction correctly classified	0.649
KL-features and description	
Apps correctly classified	221
Apps incorrectly classified	127
Fraction correctly classified	0.635
KL-features and Category	
Apps correctly classified	207
Apps incorrectly classified	141
Fraction correctly classified	0.595
KL-features and app name	
Apps correctly classified	226
Apps incorrectly classified	122
Fraction correctly classified	0.649

Table A.2: Results from initial testing of features to use for classification

Table A.3 shows the affect it had to add or subtract points depending on whether the app category was liked by the user.

Bonus points for liked categories, subtract points for disliked categories	
Apps correctly classified	177
Apps incorrectly classified	153
Fraction correctly classified	0.537
Bonus points for liked categories	
Apps correctly classified	159
Apps incorrectly classified	158
Fraction correctly classified	0.502
Subtract points for disliked categories	
Apps correctly classified	227
Apps incorrectly classified	90
Fraction correctly classified	0.716

Table A.3: Results from initial testing of adding or subtracting points based on classification

Appendix B

Test Results from Subgroups

This appendix contains the results from tests performed when dividing the participants into different subgroups. These results were not used in the research of the thesis, but might present some interesting results.

Age

Table B.1 show results when grouping participants by age. It is hard to find any patterns in the data, this may be due to the low data basis in some of the age groups. From these data, it would seem that participants between 30-50 years had most help from the personalized suggestions.

Participants 18 years or younger	
Participants in total	5
Suggestions clicked per user	5
Suggestions liked per user	3.8
Fraction of clicked suggestions liked	0.76
Participants between 19 and 29 years old	
Participants in total	35
Suggestions clicked per user	4.49
Suggestions liked per user	3.43
Fraction of clicked suggestions liked	0.76
Participants between 30-50 years old	
Participants in total	4
Suggestions clicked per user	7
Suggestions liked per user	5.5
Fraction of clicked suggestions liked	0.79
Participants over 50 years old	
Participants in total	2
Suggestions clicked per user	5.5
Suggestions liked per user	4
Fraction of clicked suggestions liked	0.72

Table B.1: Results by grouping participants by age

Gender

Table 5.2 shows the results we get when grouping the participants by gender. Out of the different subgroups, this is the most evenly split groups. We see that overall performance are pretty similar, but the suggestions seems to be slightly more relevant for the female participants.

Female	
Participants in total	22
Suggestions clicked per user	5.05
Suggestions liked per user	3.95
Fraction of clicked suggestions liked	0.78
Male	
Participants in total	24
Suggestions clicked per user	4.58
Suggestions liked per user	3.42
Fraction of clicked suggestions liked	0.75

Table B.2: Results when grouping participants by gender

Experience

The last subgroup investigated are grouping by experience from app stores. The results are found in table B.3. Again, we see that one subgroup has very few participants to use as a data basis. From the data gathered, it is hard to find patterns for how experience affect the use patterns of our participants. This may either be because there is no pattern, or because the data basis is too small. We see that our data shows the suggestions were most useful for participants with little experience from app store from before.

Low experience	
Participants in total	4
Suggestions clicked per user	7
Suggestions liked per user	5.5
Fraction of clicked suggestions liked	0.79
Medium experience	
Participants in total	19
Suggestions clicked per user	4.68
Suggestions liked per user	3.39
Fraction of clicked suggestions liked	0.72
High experience	
Participants in total	23
Suggestions clicked per user	4.52
Suggestions liked per user	3.61
Fraction of clicked suggestions liked	0.80

Table B.3: Results from initial testing of features to use for classification

Appendix C

Pseudocode

Pseudocode for Classifying Apps

```
Data: AppsToRank, UserClassifier, User
Result: Ranked list of apps according to the user's preference
Initialize variables;
foreach App in AppsToRank do
    ClassifierScores ← UserClassifier.getScore(app.featuresKL);
    if ClassifierScores.Liked > ClassifierScore.Disliked and
    User.likesCategory(App.Category) then
        ClassifierScores.Liked ← ClassifierScores.Liked + Score;
        if ClassifierScores.Liked > ClassifierScore.Disliked then
            Classify app as liked
        end
    end
    if ClassifierScores.Liked < ClassifierScore.Disliked and not
    User.likesCategory(App.Category) then
        ClassifierScores.Disliked ← ClassifierScores.Disliked + Score;
        if ClassifierScores.Liked < ClassifierScore.Disliked then
            Classify app as disliked
        end
    end
end
```

Algorithm 1: Pseudocode for algorithm that assigns scores to whether a user likes the apps in the list of apps retrieved from the query. In our prototype we user the Naive Bayesian classifier as user classifier.

Pseudocode for Assigning User's Preference to Terms

Data: User, AppList
Result: Set user's preferences for terms
foreach *App in AppList* **do**
    // If app is neither relevant nor irrelevant to the user, skip to next app;
    if *User.like(App) != (True OR False)* **then**
        if *User.like(App) == True* **then**
            UserPreferenceScore ← 1
        else
            UserPreferenceScore ← -1
        end
        foreach *Term in App* **do**
            TermKLValue ← KLScore(Term);
            User.TermPreference[Term] ← User.TermPreference[Term] +
            UserPreferenceScore;
        end
    end
end

Algorithm 2: Pseudocode for algorithm used to set the user's preference for terms, after the user has told the system their preference to one or multiple apps.