Felix Reisinger

Analysis and Improvement of Catrobat's Recommender System

Diploma Thesis

to achieve the university degree of

Magister rerum naturalium

Degree programme: Teacher Training

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Institute for Softwaretechnology

Graz, June 2019

Affidavit

Abstract

With the ever-growing amount of data of the information age, finding specific information has become a challenging task. In order to escape the information overload, users need an effective method of filtering data. An essential tool for dealing with this problem are recommender systems, as they help users to find relevant items in a large set of items. Based on user-specific or non-personalized data, recommender systems predict which items are suitable for specific users or the general public. One system that utilizes a recommender system is *Catrobat's sharing platform*. Catrobat is a project with the aim of teaching programming in an easy and intuitive way. By combining instructions for objects through visual elements, Catrobat's software enables users to create their own programs. Users can then upload the programs that they created to the share website, where uploaded programs can be browsed and downloaded. The objective of this diploma thesis is to improve the recommender system of Catrobat's sharing platform.

Therefore, the first step was to analyze Catrobat's recommender system and to identify potential improvements. Subsequently, it was decided to increase the aggregated diversity and thus also the levels of novelty and serendipity of Catrobat's recommender system. With this in mind, two re-ranking approaches for user-specific recommendations and one re-ranking approach for non-personalized recommendations were implemented. Two online experiments were conducted to evaluate the changes made to the recommender system. In order to avoid biases in the online experiments, a versatile and easily adaptable method for randomly assigning users to groups was implemented. In the first online experiment, which evaluated the two user-specific re-ranking approaches, the data generated by the users did not suffice to draw reasonable conclusions. On the other hand, for the second online experiment, which evaluated the non-personalized re-ranking approach, enough data was collected to make statistically significant statements. These data reveal that the newly introduced re-ranking of non-personalized recommendations considerably increased the aggregated diversity of Catrobat's recommender system. Moreover, the findings indicate that the levels of novelty and serendipity of the recommender system likely increased due to the re-ranking.

Zusammenfassung

Aufgrund der stetig wachsenden Menge an Daten des Informationszeitalters, kann es eine schwierige Aufgabe sein, bestimmte Informationen zu finden. Um dem Informationsüberfluss zu entkommen, bedarf es einer effizienten Methode zur Filterung von Daten. Ein dafür wesentliches Hilfsmittel sind Empfehlungssysteme, die Benutzern dabei helfen, für sie relevante Objekte aus einer Vielzahl an Objekten zu finden. Basierend auf benutzerspezifischen und nicht-personalisierten Daten prognostizieren Empfehlungssysteme Objekte, die für bestimmte Benutzer, oder für die breite Öffentlichkeit, geeignet sind. Ein System, das sich Empfehlungssysteme zu Nutze macht, ist *Catrobats sharing platform*. Catrobat ist ein Projekt, welches das Ziel verfolgt, Programmieren auf eine einfache und intuitive Art zu lehren. Um das zu erreichen, ermöglicht es Catrobats Software den Benutzer, Programme durch das Kombinieren von visuellen Instruktionen für Objekte zu entwickeln. Benutzer können anschließend ihre entwickelten Programme auf die sharing platform hochladen, wo diese durchgesehen und heruntergeladen werden können. Das Ziel dieser Diplomarbeit besteht darin, das Empfehlungssystem von Catrobats sharing platform zu verbessern.

Dazu wurde im ersten Schritt Catrobats Empfehlungssystem analysiert, um potenzielle Verbesserungen zu identifizieren. Anschließend wurde die Entscheidung getroffen, die aggregierte Diversität und dadurch auch die Stufen der Neuheit und Serendipität von Catrobats Empfehlungssystem zu erhöhen. Um das zu verwirklichen, wurden drei Umreihungsmethoden implementiert, davon zwei für die benutzerspezifischen und eine für die nicht-personalisierten Empfehlungen. Zur Evaluierung der Änderungen am Empfehlungssystem wurden zwei Online-Experimente durchgeführt. Damit die Online-Experimente nicht durch bestimmte Faktoren beeinflusst werden, wurde eine wiederverwendbare und einfach anpassbare Methode implementiert, mit deren Hilfe Benutzer nach dem Zufallsprinzip Gruppen zugewiesen werden können. In dem ersten Online-Experiment, welches die beiden benutzerspezifischen Umreihungsmethoden evaluierte, reichten die von den Benutzern generierten Daten nicht aus, um vernünftige Schlüsse ziehen zu können. Für das zweite Online-Experiment hingegen war es möglich, genügend Daten zu sammeln, um statistisch signifikante Aussagen treffen zu können. Diese Daten weisen darauf hin, dass die neu eingeführte Umreihungsmethode der nicht-personalisierten Empfehlungen die aggregierte Diversität von Catrobats Empfehlungssystem deutlich erhöhte. Darüber hinaus deuten die Daten an, dass sich aufgrund der Umreihungsmethode die Stufen der Neuheit und Serendipität des Empfehlungssystems erhöht haben.

Contents

List of Figures

List of Tables

List of Codes

1. Introduction

Today's amount of available data poses a great challenge to users in situations where decisions are required. Often the user has too many alternatives to choose from to be able to process each option individually. In other situations personal experience and knowledge about the domain at hand might not be sufficient to make a qualified decision between multiple alternatives. Thus, a way of filtering data effectively and presenting relevant information is required. A popular software tool to assist users in the process of decision-making, which has received much attention lately, are recommender systems. Their purpose is to recommend items to users which are interesting and relevant. Recommender systems are used in many different domains, such as e-commerce, news articles and social networks. An example of a common use of recommender systems on an online shopping platform, such as Amazon[1], is to recommend products to customers based on the customers' taste and purchase history.

The goal of this thesis is to improve the recommender system of Catrobat[2], an open-source project of the Technical University of Graz. Therefore, Catrobat's existing recommender system is analyzed and potential improvements are discussed. One of the discussed improvements is then implemented and an online experiment is conducted, in order to study the effects of the potential improvement on the system. The underlying goal of the online experiment is to answer the following research questions:

Research Question 1: *Can the aggregated diversity of Catrobat's recommender system be significantly improved by one of the implemented approaches while maintaining an acceptable level of accuracy?*

[1] *Amazon* 2019.
[2] International Catrobat Association, 2019a.

Research Question 2: *How do the implemented approaches impact the novelty and serendipity levels of Catrobat's recommender system?*

The thesis is structured as follows. Chapter 2 concentrates on recommender systems. Thereby recommender systems and their properties, as well as the basic terminology in the context of recommender systems, are defined. The recommender systems' functions are then viewed from different perspectives. Moreover, different recommendation approaches are presented, whereby exemplary recommendation processes are showcased. At the end of chapter 2 various evaluation settings for recommender systems are discussed. In chapter 3, Catrobat as an organization and as a programming language is presented. Then Catrobat's recommender system and it's environment, Catrobat's sharing platform, are portrayed. Chapter 4 describes potential improvements to Catrobat's recommender system. The improvements are then discussed and one potential improvement is chosen in order to be implemented in the practical part of this thesis. Afterwards related work is discussed. In chapter 5 the implementations of the practical part of this thesis are presented. Chapter 6 discusses the evaluation of the implemented changes to Catrobat's recommender system. Finally, in chapter 7 the findings of this thesis are summarized as well as future work.

2. Recommender Systems

This chapter covers various aspects of recommender systems. First of all, recommender systems are defined. Thereby the basic terminology used and the origins of recommender systems are outlined. Furthermore, the functions and properties of recommender systems are presented. Then different kinds of feedback are discussed, which are needed in the recommendation process. Afterwards common recommendation approaches are presented and discussed. Finally, different evaluation settings of recommender systems are presented.

2.1. Definition

In the following section, recommender systems are defined. First and foremost, the basic terminology used in the context of recommender systems is presented. Then the origin of recommender systems is portrayed, followed by a definition of their functions from the perspective of a provider of recommender systems and their users. Afterwards properties of recommender systems are presented.

2.1.1. Terminology

This section deals with the basic terminology, which is used to describe recommender systems (Ricci, Rokach, and Shapira, 2015).

Recommender system A software tool serving the purpose of presenting recommendations of items to users.

Recommendation A recommendation is an item or a list of items, which is recommended to a user by a recommender system.

Recommendation list A recommendation list is a list of items, which consists of recommendations and which is created by a recommender system.

Item Items are the objects of a system, for example books in a book shop or movies on a movie streaming platform. Thus, which objects are represented by items depends on the given recommender system. Items are usually described by item attributes and can be further described by their complexity and their value or utility. Exemplary item attributes for a movie are *genre* and *year of release*. Item complexity is usually (but not exclusively) dependent on how many attributes an item has. Examples of low-complexity items are movies, books and news articles. Examples of high-complexity items are smartphones, laptops, jobs and financial investments. Value or utility describes how relevant an item is to a user. Different recommender systems use different ratings to represent how much value or utility an item has for a user, as discussed in section 2.2.

User A user is a person who uses a system. Users vary in many factors, such as demographics, interests, experience, behavior, goals, loyalty. In an online shop, for instance, users are customers. Users who are not logged in the system are usually referred to as *guest users*. The term *active user* is used to describe a user who receives a recommendation and is primarily used in explanations of how a recommender system works.

User model A user model, or *user profile*, is the representation of a user's taste and preferences by the system. Recommender systems are designed to gather information about users in order to be able to create user models and personalized recommendations (Jannach, Zanker, et al., 2010). Which information is considered in the user model depends on how the recommender system is designed and on the domain in which the recommender system operates.

System provider The system provider is the person, group or company who provides the recommender system service.

Domain The domain of a recommender system describes the environment in which it operates (Ricci, Rokach, and Shapira, 2015). An online book shop, for example, operates in the domain of books.

Transaction Recorded interactions between the system and the active user are commonly referred to as transactions (Ricci, Rokach, and Shapira, 2015). They represent important information which can be used by the recommender system. Which transactions are recorded depends on the specific system. Examples of frequently recorded transactions are:

- a user views an item
- a user consumes an item
- a user rates an item
- a user registers a user account
- a user logs into a user account.

Conversion A conversion can generally be defined as a desired action by a user. For example, when a user consumes an item, the process can be described as conversion.

Conversion rate In general, a conversion rate describes the percentage of users who take a desired action (Nielsen, 2013). In the context of recommender systems, the conversion rate is usually defined as the ratio of how many of the viewed items have been consumed by a user. It is calculated by dividing the number of visits for a given item set by the respective number of downloads. For example, if 1,000 books have been viewed within a given time frame and 100 have been consumed, then the conversion rate equals 0.1, as shown in equation (2.1) and equation (2.2).

$$conversion\ rate = \frac{100}{1000} \tag{2.1}$$

$$conversion\ rate = 0.1 \tag{2.2}$$

2.1.2. Origin

Recommender systems as a research area came into existence in the early 1990s. The catalyst was the need to deal with the high quantity of data generated by the rapid growth of the internet. One of the first well-known recommender systems is *Tapestry* (Goldberg et al., 1992) that was developed in 1992. This mailing system introduced collaborative filtering, a method allowing people to collaborate and to rate items in order to find interesting ones and get rid of uninteresting ones (see section 2.3.2). The motivation for the creation of Tapestry was to find a solution to the problem of dealing with large amounts of electronic mail. While there were already multiple mail systems that supported filtering, the creators of Tapestry believed that including human reactions to content-based filtering makes the process more effective. In the years subsequent to the development of Tapestry, many recommender systems that use collaborative filtering were developed, such as *Fab* (Balabanović and Shoham, 1997) and *GroupLens* (Resnick, Iacovou, et al., 1994).

With the ever-growing amount of data and the booming industry of e-commerce, recommender systems have continued to receive more and more attention over the years. The fact that recommender systems play an important role in a variety of different domains led to a widespread adoption of different recommender systems by many businesses. Since individual domain characteristics, such as the average amount of items purchased by a user and unique item features necessitate different recommender systems to optimally meet the requirements and characteristics of a domain, recommender systems became an important area of research. Some of the most prominent domains in which recommender systems are commonly used are the following (Ricci, Rokach, and Shapira, 2015):

- online shopping
- streaming services
- travel business
- social networks
- advertisements
- news
- dating.

Most domains can be further divided. The domain of streaming services, for example, can be divided into the sub-domains of music streaming, movie streaming, live streaming, to name but a few.

An important characteristic of recommender systems is that higher amounts of data about users and items generally lead to higher quality recommendations, while higher quality recommendations generally result in more users and items. Therefore, if there are multiple contenders for a given domain or sub-domain, predatory competition is expected as described in Resnick and Varian, 1997. The article hints that it is likely that there remains only one provider for each domain or sub-domain who will be highly successful in the long run.

A classic example of strong competition is the *Netflix Prize contest*[1]. In 2006 the Netflix company held a contest with a prize pool of one million dollar for the winning team. The goal was to develop a recommender system that could outperform Netflix's current recommender system named *Cinematch* by at least 10% on a given data set. The competition was planned to last until at least 2011. Over 50,000 contestants in over 40,000 teams from 186 different countries took part in the challenge[2]. Every year, in which the goal of at least 10% improvement in prediction accuracy was not achieved, the current leading team received $50,000. In 2009, the challenge was successfully met by the team called *BellKor's Pragmatic Chaos*, which utilized a form of collaborative filtering (Andreas Töscher, 2009). The Netflix Prize contest has drawn considerable attention to recommender systems as an area of research.

In 2007, only one year after the start of the Netflix Prize contest, recommender systems gained again in popularity with the first annual conference on recommender systems, held by the recommender system community *RecSys Community* of the Association of Computing Machinery[3] (ACM).

2.1.3. Functions

As already described at the beginning of chapter 2, a recommender system fulfills the purpose of showing items which a user will like and of avoiding items that a user will not like. This definition can be extended by viewing the function of recommender systems from two perspectives, firstly from the perspective of the provider of the system and secondly from the perspective of the user of the system. Both the provider and the user want and

[1]Netflix, 2009a.
[2]Netflix, 2009b.
[3]RecSys Community, 2019.

expect certain functionalities as regards the recommender system, which are discussed in the following two subsections.

System Provider's View

This subsection lists several functionalities that the provider of a recommender system wants, based on the description in Ricci, Rokach, and Shapira, 2015, pp. 4–6.

Increase number of items sold The most obvious function of a recommender system is to increase the number of items sold (Ricci, Rokach, and Shapira, 2015). By showing items to users that the users most probably like, the chance that users purchase these items will most likely be higher. Therefore, usually more items are sold with the help of a recommender system. However, not only commercially oriented providers profit from an increasing number of items sold but also other domains (Ricci, Rokach, and Shapira, 2015). An increased number of items being consumed by users in non-commercial systems also brings benefits to the provider of the system, for instance by binding the user to the system as users spend more time on the website.

Sell more diverse items It often proves difficult to show less popular items to users, since the risk that users dislike less popular items is higher than the risk that users dislike popular items, which have already been well received. In many domains it is still necessary to show less popular items, such as the travel domain where users are recommended different hotels. Only showing popular hotels might have a detrimental effect due to the limited amount of available rooms per hotel and the risk of overcrowded destinations (Ricci, Rokach, and Shapira, 2015).

Increase user satisfaction Recommender systems are not only tools to generate more conversions. Their purpose is also to increase the overall user satisfaction with the system. When a user is pleased with the recommendations, while also having an enjoyable experience with the human-computer interaction, the user will be satisfied with the system. The more satisfied the user is with the system, the more likely the user will use the system again in the future.

Increase user fidelity User fidelity is an important aspect within the context of recommender systems. Firstly, a high user fidelity is desirable because a user who remains loyal to a system consumes more items in the long run. Secondly, recommender systems need data about users in order to generate accurate and relevant recommendations. A user who is loyal to the system generates more data over time, which enables the recommender system to produce more accurate and relevant recommendations for the user. This, in turn, will increase the user's satisfaction with the system.

Understanding what the user wants Another functionality of recommender systems is that data about a user's preferences are either collected, or generated. This data can be used to gain a better understanding about what users like and dislike. This information can be taken advantage of in other areas of the system. For example, such data can help a business to decide in which area to expand in the future.

User's View

Not only the provider of a recommender system has certain requirements and expectations when it comes to the functionalities of recommender systems. This also applies to the users of the system. This section lists several functionalities that users expect from recommender systems, and what motivates users to take advantage of recommender systems. The functionalities described do not refer to one specific recommender system or one specific domain. The following list is based on Herlocker, Konstan, Terveen, et al., 2004, pp. 8–12 and Ricci, Rokach, and Shapira, 2015, pp. 6–7.

Annotation in context Similar to the functionality of Tapestry (see section 2.1.2) recommender systems can annotate items in a given context (Ricci, Rokach, and Shapira, 2015). For example, on a news website a recommender system can emphasize specific news articles which a user is likely to find interesting and relevant, given the user's long time preferences. This can help users in the decision-making process.

Find good items Finding good items, where *good* generally refers to relevant and interesting, and presenting them to the user is one of the main functionalities of most recommender systems (Ricci, Rokach, and Shapira, 2015). From the user's perspective this functionality saves time, as otherwise the search for good items would have to be done manually. Furthermore, it is often the case that not enough information is available for the user to identify good items as such. Alternatively, in some instances good items cannot be identified as such because of insufficient domain experience.

Find all good items At first sight this functionality seems counter-intuitive, since one of the main motivations of recommender systems is to reduce information overload and to only present a few top items. However, in some domains, as in the medical or financial domain, it can be critical to find all relevant and interesting items, instead of only a chosen few, according to Herlocker, Konstan, Terveen, et al. (2004). The authors state that in these domains it is important to have a low false negative rate, which is the rate at which relevant and interesting items are wrongly classified as not relevant and not interesting by the recommender system.

Recommend sequence In some domains it is reasonable to recommend a sequence of items that fit together (Ricci, Rokach, and Shapira, 2015). For example, on a website that offers online courses, an intermediate math course can be a relevant and interesting recommendation after having completed a beginner's math course. Without the prerequisite of the completed beginner's math course, the intermediate math course might not be a relevant and interesting recommendation. Another example is the recommendation of a list composed of individual songs on a music streaming platform. Some songs might only be included in the list because they fit well with the other songs and would not be recommended to the user individually (Herlocker, Konstan, Terveen, et al., 2004).

Just browsing Herlocker, Konstan, Terveen, et al. (2004) found that in some cases, users use the system simply to browse through items without the intention of making a purchase, because they experience the browsing activity as pleasant. In such cases the recommender system's usability is an important factor, while a high prediction accuracy is less important than in other cases (Herlocker, Konstan, Terveen, et al., 2004).

Find credible recommender According to Herlocker, Konstan, Terveen, et al. (2004), users do not automatically trust recommender systems. The authors state that some users test the system's credibility by searching for various items they already know and by checking their ratings, if possible. Others may try to influence the system in a certain way, in order to see if the recommendations change accordingly (Herlocker, Konstan, Terveen, et al., 2004). However, it is difficult for recommender systems to appear trustworthy, since the system's algorithm might favor other items than the user expects when testing the system . Some recommender systems, for example, focus on recommending diverse and less popular items to users, in order to try to recommend items which the user does not know yet. If the system does not communicate its efforts to recommend new and less popular items, users might lose their trust in the recommender system, because the recommender system does not recommend popular items which users, who test the system, expect (Herlocker, Konstan, Terveen, et al., 2004).

Improve profile In order to create user-specific recommendations, the recommender system needs data about its users. By giving feedback about items, users can improve their user profile or user model, which is the system's representation of user preferences and interests (Ricci, Rokach, and Shapira, 2015).

Express self The motivation of some users behind rating items is not to improve the recommendations that they will receive in the future, but to express themselves, as they find satisfaction in doing so (Ricci, Rokach, and Shapira, 2015). For these users a high usability of the item rating process contributes to user satisfaction and loyalty to the system.

Help others Some users find satisfaction in helping others with their item ratings (Ricci, Rokach, and Shapira, 2015). This is prevalent in systems that are not routinely used and in domains, where usually only one item is consumed, as stated in Ricci, Rokach, and Shapira, 2015. The authors give the example of rating a bought car on a platform that sells cars. Rating the item is most probably of little value to a user, since the user usually stops using the platform after one car has been bought. However, other users might profit from the rating. Although not exclusively, the wishes to express oneself and to help others often go hand in hand (Herlocker, Konstan, Terveen, et al., 2004).

Influence others Given the fact that recommender systems rely on user feedback to generate recommendations, it is possible to explicitly influence recommender systems (Ricci, Rokach, and Shapira, 2015). This is especially frequent in commercially oriented systems, where stakeholders might try to influence users to buy their products. For example, on the online shopping platform Amazon[4], sellers often buy positive reviews for their products in order to deceive customers and make them more likely to buy the seller's products (Fornaciari and Poesio, 2014).

2.1.4. Properties

Recommender systems have many different properties that can be used to describe how a recommender system performs in different areas. The property most commonly used to portray a recommender system's performance is the *prediction accuracy* property, which describes how accurately a recommender system can predict user ratings for items. Nonetheless, accuracy is not the only property which is essential for the overall user satisfaction with the recommender system. Which properties are important in a given recommender system depends on multiple factors, such as the user's motivation when using the recommender system and the domain in which the recommender system operates.

The following list outlines properties that are commonly used to describe and evaluate recommender systems. The list is based on Ricci, Rokach, and Shapira, 2015, pp. 280–304.

Prediction Accuracy Basically recommender systems make predictions about user ratings for items that users have not seen yet, depending on prior knowledge about the users. Thereby the accuracy of a prediction can be measured and described. Is is widely assumed that accuracy plays a major role in how well a recommender system is received by its users. Therefore, maximizing the accuracy of recommender systems is a common goal in research. One example of an attempt to maximize prediction accuracy is the Netflix Prize Competition (see section 2.1.2).

[4]*Amazon* 2019.

Coverage This property can be divided into *item space coverage* and *user space coverage*, as described in Ricci, Rokach, and Shapira, 2015. The authors define item space coverage as the amount of items that are eligible for recommendation by the recommender system. In many systems it may occur that there is a small subset of items, which is very popular, and which is recommended frequently, while the rest of the item set receives only very little to no attention. This distribution is often called *long-tail* or *heavy-tail* problem (Ricci, Rokach, and Shapira, 2015). User space coverage refers to the portion of users who are eligible to receive recommendations by the recommender system (Ricci, Rokach, and Shapira, 2015). Given that recommender systems need information about users in order to predict the users' tastes, users who have not provided enough information cannot receive recommendations.

Confidence The quality of recommendations depends on multiple factors, such as the quantity of available data for users and items. Therefore, recommendations have different levels of confidence, whereby confidence can be described as trust in the correctness of the recommendation (Ricci, Rokach, and Shapira, 2015). For example, recommending an item to a user who has given the bare minimum of information required to generate a recommendation usually reaches lower confidence levels than recommending an item to a user who has already used the system for a long time and for whom the system has already established a refined user model. Similarly, recommending unpopular items, which have not been rated by many users yet, is riskier than recommending items which have already been rated by many users. Hence recommending unpopular items usually results in less confidence in the recommendation, compared to recommending popular items.

Trust Trust describes a user's trust in the system recommendation (Ricci, Rokach, and Shapira, 2015). This property is not to be confused with confidence, which describes the system's confidence in the recommendations. One way to manipulate trust is to recommend obvious items to users, so that the users can relate to the recommendation and verify that the recommender system works as intended (Ricci, Rokach, and Shapira, 2015). However, there is a trade-off between showing obvious recommendations and the goal to recommend different and novel items. Furthermore, it is assumed that a way to increase a user's acceptance of a recommendation is to disclose the recommender system's confidence in the recommendation or the ratings of similar users for the recommended items (Herlocker, Konstan, and Riedl, 2000).

Novelty In the context of recommender systems, an item can be referred to as *novel*, if a user did not know about the item before receiving the recommendation (Ricci, Rokach, and Shapira, 2015). Novelty can be an important property of recommender systems. For instance, when a user's goal is to discover new holiday resorts on a travel platform, then the user will especially look for novel items. However, high novelty levels of recommendations can have a negative impact on user satisfaction and perceived quality (Ekstrand et al., 2014, Celma and Herrera, 2008).

Serendipity Recommendations that have a high serendipity score are non-obvious recommendations, which are surprisingly successful, whereby successful means, that the user accepted the recommendation (Ricci, Rokach, and Shapira, 2015). For example, given a recommender system for an online book store and a user who only purchased books falling into the fantasy genre in the past, recommendations that feature popular fantasy books, which the user has not seen yet, will most likely obtain a high prediction accuracy. However, the recommended books will hardly surprise the user. On the other hand, books from different genres, such as thrillers, will obtain a lower prediction accuracy, but the recommendation will probably surprise the user. Thus, if a thriller is recommended and the user accepts the recommendation and buys the book, the recommendation has a high serendipity score.

Diversity A recommendation's diversity describes how different the individual items are compared to each other and is generally defined as the counterpart of similarity (Ricci, Rokach, and Shapira, 2015). What levels of diversity are desired usually depends on the domain at hand. Typically there is a trade-off between diversity and accuracy (Gediminas Adomavicius and Youngok Kwon, 2009). For example, given a movie streaming platform and a user who rated action movies starring the actor Liam Neeson very highly in the past, the platform's recommender system might only recommend action movies starring Liam Neeson to the user. The recommendation's accuracy will be very high, as the user will most probably like the movies. However, the recommendation will be one-sided and there will be a low level of diversity.

Most likely the user satisfaction would be higher if a broader range of items was offered, even though the recommendation's accuracy might decrease.
In general, diversity in the context of recommender systems can be divided into *aggregated* and *individual diversity*, as described in G. Adomavicius and Y. Kwon, 2012. The authors state that aggregated diversity describes how diverse a set of recommended items is from the perspective of the recommender system. Individual diversity, on the other hand, describes how diverse a recommendation is from a user's perspective (G. Adomavicius and Y. Kwon, 2012). A high individual diversity does not necessarily cause a high aggregated diversity. For instance, given a set of 1000 items and a subset of 10 items which are highly diverse, recommending only the subset of 10 items will produce recommendations with high individual diversity, but at the same time the aggregated diversity will be low (G. Adomavicius and Y. Kwon, 2012).

Utility Recommender systems bring utility to the system provider and the system's users. The utility of a recommender system can usually be measured in multiple ways, depending on the system providers' and the user's goals (Ricci, Rokach, and Shapira, 2015). In e-commerce systems, for example, the system provider's goal is usually to maximize profits, therefore the recommender system's utility can be measured by how much it increases profits (Ricci, Rokach, and Shapira, 2015). For users, utility generally refers to how useful the recommendations, which they receive, are, which depends on the user's goals (Ricci, Rokach, and Shapira, 2015). For example, the novelty and serendipity of the recommended items can play an important role in the question of how useful a recommendation is perceived by a user.

Risk Users generally differ in their tendency and willingness to take risks, which is an important aspect in certain domains, such as the investment market, according to Ricci, Rokach, and Shapira, 2015. Thus, some recommender systems might consider with how much risk items are associated. Usually risk-sensitive recommender systems evaluate the risk by also considering the utility variance, instead of only the expected utility (Ricci, Rokach, and Shapira, 2015).

Robustness Robustness is the term used to describe how robust the recommender system is in case of a malicious attack (Ricci, Rokach, and

Shapira, 2015). A common example of a malicious attack is to influence which items are recommended by the recommender system. This attack is widespread in the online shopping domain, where a seller might create many user accounts in order to rate competing items negatively, so that customers are more likely to buy the attacker's products. Another example of a malicious attack is when a person tries to flood a website with large amounts of requests, which results in the website being overloaded and thus unavailable for potential customers (Ricci, Rokach, and Shapira, 2015). The motivation behind such so-called *denial-of-service attacks* can be to temporarily eliminate competitors or to blackmail the victim of the attack (Ricci, Rokach, and Shapira, 2015).

Privacy Privacy is a general concern in recommender systems, but most notably in collaborative filtering recommender systems due to the fact that the main source of information on which recommendations are based on in collaborative filtering systems is personal information about the user (see section 2.3.2 and section 2.3.3) (Ricci, Rokach, and Shapira, 2015). However, most recommender systems cannot work properly without utilizing information about users, though it is important that no private information about a user is disclosed to other users (Ricci, Rokach, and Shapira, 2015).

Adaptivity Many recommender systems operate in domains in which the ability to quickly adapt to certain factors is important (Ricci, Rokach, and Shapira, 2015). For example, recommender systems that operate in the stock market domain need to be able to quickly recognize new trends and to respond appropriately. Adaptivity is also essential in the case that the system's item set changes often and the system therefore needs to work with little information (Ricci, Rokach, and Shapira, 2015). Furthermore, it is important that a recommender system quickly adapts to changes in a user's profile and in a user's preferences, because otherwise users might get the impression that changes to their profile and item ratings do not have an impact on their recommendations and might therefore be judged as wasted effort (Ricci, Rokach, and Shapira, 2015).

Scalability Scalability describes how scalable a recommender system is in terms of space and time requirements (Ricci, Rokach, and Shapira, 2015). Space requirements are usually a concern if there are many users or items, which require large amounts of data to be stored (Jannach, Zanker, et al., 2010). Time requirements are also an important aspect of recommender systems, given that the time needed to generate a recommendation generally increases with the number of users and items (Ricci, Rokach, and Shapira, 2015). Moreover, scalability might be an issue especially in memory-based recommendation approaches (see section 2.3.2), in contrast to model-based recommendation approaches, which work on a compressed set of the available data in order to increase scalability (see section 2.3.3) (Jannach, Zanker, et al., 2010).

2.2. Feedback

In order for a recommender system to generate personal recommendations, users need to provide information about themselves, for example by editing their user profile or by rating items. Without personal information about users, recommender systems can only recommend items that appeal to the average user. Therefore, providing information is an important aspect for a user of recommender systems. However, the user might not always be aware that feedback is obtained from the user's actions and behavior. In these cases, the user is providing so-called *implicit feedback* (Jannach, Lerche, and Zanker, 2018). On the other hand, if a user explicitly and deliberately provides feedback about preferences or about items, the user gives so-called *explicit feedback* (Jannach, Lerche, and Zanker, 2018). In the following sections explicit and implicit feedback are described. Afterwards a discussion follows.

2.2.1. Explicit Feedback

Explicit feedback, as described in Jannach, Lerche, and Zanker, 2018, is feedback that a user provides deliberately, unambiguously and explicitly. The authors state that an advantage of explicit feedback over implicit feedback is that it is accurate and there is no need for interpretation. A common example is to rate items on a scale, typically it is based on a one-to-five-star scale (Jannach, Lerche, and Zanker, 2018). Providing explicit feedback usually requires a certain amount of effort on the part of the user. For simple items the cost is relatively low. However, for multi-criteria items, rating multiple different criteria can be a time-consuming and complex task (Jannach, Lerche, and Zanker, 2018). Furthermore, users might not understand how providing feedback is useful to them or they might simply not care about the benefit it brings, for example that it improves the user model. Therefore, explicit feedback is generally sparse (Jannach, Lerche, and Zanker, 2018).

The following list comprises commonly used explicit feedback based on Jannach, Lerche, and Zanker, 2018:

Rating scales Rating scales often range from one to five. Visually they are commonly presented as stars or hearts.

Unary feedback Unary feedback only gives one option, for instance the option to like an item, or to share an item.

Binary feedback Binary feedback offers two options, typically a positive and a negative one. Examples of binary feedback are *thumbs-up* and *thumbs-down* buttons, or *like* and *dislike* buttons.

Negative feedback Negative feedback is used to state that a user does not like an item. Examples of negative feedback are options to avoid seeing specific content on social-media platforms and to mark items as not useful or uninteresting.

Natural language Comment sections and product reviews are often used as explicit feedback, even though natural language can be ambiguous in some cases, for example because of the use of sarcasm and irony.

Tags In some systems items can be annotated with certain tags by a user, which provide feedback about the user's opinion about the item, for instance tags such as *like* and *love* show that a user's opinion of an item is positive.

Domain-specific feedback In some domains users can provide domain-specific feedback, for example Goodreads'[5] *Want-to-Read list* and Amazon's[6] *wish list*.

Figure 2.1 shows book recommendations from Goodreads for a logged in

[5]*Goodreads Homepage* 2019.
[6]*Amazon* 2019.

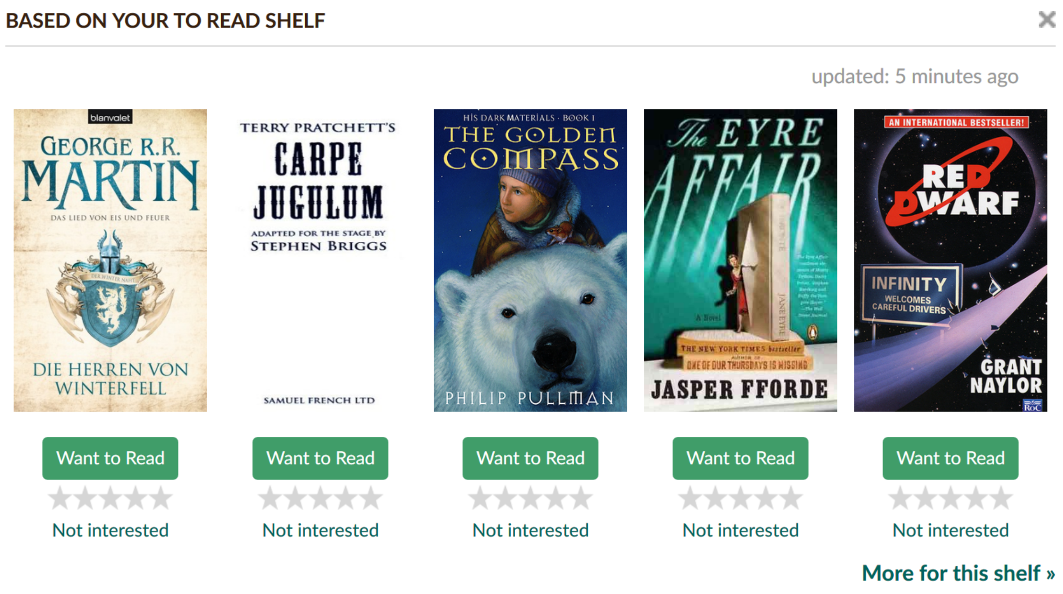


Figure 2.1.: This figure shows recommended books based on Goodreads' Want-to-Read list for a logged in user. The picture was created by a screenshot from Goodreads' recommendations page (*Goodreads Recommendations* 2019).

user based on Goodreads' Want-to-Read list[7]. The user can add books from the recommendation list to the Want-to-Read list, rate a book on a rating scale based on one to five stars if the user has already read a presented book, or provide negative feedback by stating that the user is not interested in a book.

2.2.2. Implicit Feedback

Implicit feedback is gathered from user behavior and can be interpreted in order to draw conclusions about a user's preferences. Common examples of logged user behavior are viewing an item, conversions, search behavior, time spent, clicks, scrolls and bookmarks (Ricci, Rokach, and Shapira, 2015). An advantage of implicit feedback is that there are large amounts of data available that can be logged and interpreted. Furthermore, it does not

[7]*Goodreads Recommendations* 2019.

require any effort on the part of the user since the system can log the utilized data automatically. Thus, implicit feedback is especially useful in domains where it takes considerable effort to provide explicit feedback, as for instance in domains that feature multi-criteria items.

The term positive implicit feedback is used to describe implicit feedback indicating that a user likes a given item and is gathered from logged user behavior (Jannach, Lerche, and Zanker, 2018). On the other hand, the term negative implicit feedback is used to describe implicit feedback indicating that a user dislikes an item (Jannach, Lerche, and Zanker, 2018). Negative implicit feedback is mostly found in missing data. A general problem when interpreting missing data is that it is very difficult to distinguish between different reasons why the data is missing (Jannach, Lerche, and Zanker, 2018). For example, if a user views an item on the website of an online shop but does not purchase the item afterwards, it seems that the user was initially interested in the item but did not like the item upon closer inspection. Nonetheless, this conclusion cannot be drawn for certain since there may be many different explanations why the user did not purchase the item after viewing it, other than the explanation that the user did not like the item upon closer inspection. For instance, it might be the case that the user just wanted to browse through the set of items without the intention to make a purchase, or the case that the user likes the item, but there is another similar item that the user prefers. However, also positive implicit feedback is ambiguous, as the user's motivations for certain actions can only be guessed. For example, if a user makes a purchase which is unusual, given the user's purchase history, it is unclear, whether the user made a serendipitous purchase, or whether the user purchased the item as a present or for a family member. Similarly, the fact that a user views an item for a prolonged period of time does not necessarily mean that the user finds the item interesting due to the possibility that the user could have been distracted and therefore did not pay attention to the item (Jannach, Lerche, and Zanker, 2018).

2.2.3. Discussion

Explicit and implicit feedback both have advantages and disadvantages. Explicit feedback is mostly unambiguous and more accurate and is therefore especially suitable for sectors in which high accuracy and low false positives rates are important, such as law, investment and the medical field. On the other hand, implicit feedback is comparatively much easier to gather and less sparse. It does not require any effort on the part of the user, which makes it suitable for sectors in which users are less likely to rate items, as for instance in domains, in which users usually only purchase one item. Furthermore, implicit feedback does not require the user to disclose private information, which can be an important factor in certain sectors. Utilizing both, implicit and explicit feedback simultaneously, as described in Ebadi and Krzyzak, 2016, will usually yield the best results.

2.3. Recommendation Approaches

Recommender systems are a broad research area and thus many different recommendation approaches have been developed over time. The approaches differ markedly, for instance which data are utilized in the recommendation process and how the data are used to generate recommendations for users. In the following, common recommendation approaches are presented, examples are used to illustrate some of these approaches.

2.3.1. Non-Personalized Recommendations

The simplest form of recommendations are non-personalized recommendations, which are recommendations of items, regardless of who receives them. These types of recommendations are often used for guest users and users who have not provided enough information about themselves in order to receive personalized recommendations (Ricci, Rokach, and Shapira, 2015). Therefore, the general goal of non-personalized recommender systems is to recommend items that appeal to as many users as possible. This is commonly achieved by utilizing implicit and explicit feedback in order to find

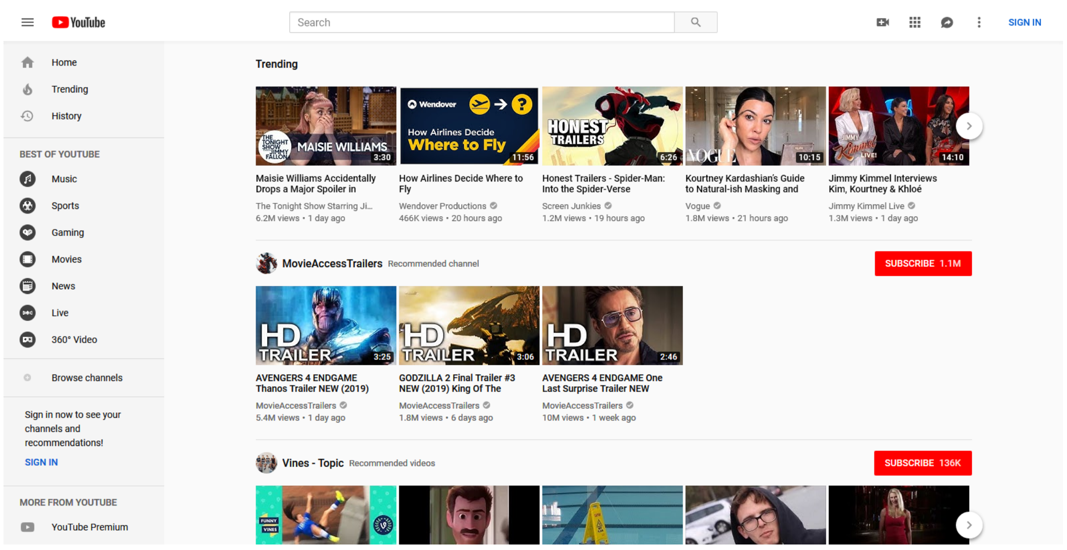

YouTube
Search
SIGN IN
Home
Trending
History
BEST OF YOUTUBE
Music
Sports
Gaming
Movies
News
Live
360° Video
Browse channels
Sign in now to see your channels and recommendations!
SIGN IN
MORE FROM YOUTUBE
YouTube Premium
Trending
Maisie Williams Accidentally Drops a Major Spoiler in
The Tonight Show Starring Ji...
6.2M views • 1 day ago
How Airlines Decide Where to Fly
Wendover Productions
466K views • 20 hours ago
Honest Trailers - Spider-Man: Into the Spider-Verse
Screen Junkies
1.2M views • 19 hours ago
Kourtney Kardashian's Guide to Natural-ish Masking and
Vogue
1.8M views • 21 hours ago
Jimmy Kimmel Interviews Kim, Kourtney & Khloé
Jimmy Kimmel Live
1.3M views • 1 day ago
MovieAccessTrailers Recommended channel
SUBSCRIBE 1.1M
AVENGERS 4 ENDGAME Thanos Trailer NEW (2019)
MovieAccessTrailers
5.4M views • 1 day ago
GODZILLA 2 Final Trailer #3 NEW (2019) King Of The
MovieAccessTrailers
1.8M views • 6 days ago
AVENGERS 4 ENDGAME One Last Surprise Trailer NEW
MovieAccessTrailers
10M views • 1 week ago
Vines - Topic Recommended videos
SUBSCRIBE 136K

Figure 2.2.: This figure shows YouTube's home page recommendations (*YouTube Homepage* 2019). It has been accessed in private mode, with the location set to the United States. It shows the non-personalized recommendations that YouTube's recommender system generates for guest users, which are trending items, recommends popular channels and videos.

popular or trending items. However, non-personalized recommendations generally do not rely on explicit feedback, as implicit feedback will suffice. Figure 2.2 shows an example of non-personalized recommendations on the video streaming platform YouTube[8] to a guest user. The recommendations consist of trending items and popular channels and video categories. The cost to implement a non-personalized recommender system is generally low compared to user-specific recommender systems, hence it is suitable for smaller systems. Classic examples of non-personalized recommendations are most popular items, trending items, popular items of certain categories, new items and random items. Recommending new and random items to the average user might have a negative impact on the overall user satisfaction (Ekstrand et al., 2014), however, it might prove helpful if the system's item coverage is low.

[8] *YouTube Homepage* 2019.

2.3.2. Memory-Based Collaborative Filtering

Collaborative filtering is a recommendation technique which is utilized to generate user-specific recommendations. Similar to word-of-mouth recommendations in daily life, collaborative filtering utilizes preferences from users who share similar tastes in order to find relevant items to recommend (Jannach, Zanker, et al., 2010). Since the focus lies on a user's taste in items, the technique is also applicable in domains where only little information about users and items is available. There are two kinds of collaborative filtering approaches, *memory-based* and *model-based*. Memory-based approaches use all available data about users and items during the recommendation process. However, in many systems there are large amounts of users and items, which can create bottlenecks as regards the performance of the recommender system (Jannach, Zanker, et al., 2010). Model-based collaborative filtering approaches, on the other hand, operate on a compressed version of the data in order to increase performance in case of large data sets (Jannach, Zanker, et al., 2010). This section focuses on memory-based approaches. Model-based collaborative filtering approaches are further discussed in section 2.3.3.

Memory-based collaborative filtering can be divided into two main approaches, *user-based* and *item-based* collaborative filtering. In their basic form both approaches try to predict a user's ratings for unseen items in order to present the items with the highest predicted ratings[9]. In the user-based approach this is achieved by finding users that are most similar to the active user, while the item-based approach focuses on finding items that were similarly rated as the items that the active user rated highly.

In the following both memory-based approaches of collaborative filtering are showcased on the basis of an example described in Jannach, Zanker, et al., 2010, in order to demonstrate the recommendation processes. Table 2.1 shows user ratings between one and five for five different movie items. The goal is to find recommendations of items for the active user u_a based on the other users' tastes in items. Formally, a user is defined as an element u of the set of all users U ($u \in U$, $U = u_1, u_2, ..., u_n$). An item is defined as

[9]Though in more refined recommender systems, other factors, such as the diversity, novelty or serendipity of the set of recommended items might be considered as well.

	Ghostbusters	Titanic	Gladiator	Inception	Casablanca
u_1	2	1	2	3	
u_2		4	3	5	5
u_3	5	1		4	
u_4	2		1		3
u_5	5		3	3	
u_a		4	2		

Table 2.1.: This table shows user ratings ranging from 1 to 5 for different movies.

an object i in the set of all items I ($i \in I$, $I = i_1, i_2, ..., i_m$). The rating matrix R is defined as $m \times n$ matrix of individual ratings r_{u_x,i_y} of items i_y with $y \in \{1, 2, ..., m\}$ by users u_x with $x \in \{1, 2, ..., n\}$ (Jannach, Zanker, et al., 2010).

User-Based Approach

User-based collaborative filtering approaches analyze other users' preferences in items in order to find users with similar tastes to the active user. These similar users are sometimes referred to as *peer users* or *nearest neighbors* (Jannach, Zanker, et al., 2010). Therefore, the first step in the exemplary recommendation process is to find the k-nearest neighbors to the active user, where k is defined as $k = 1$ in the showcased example for the sake of simplicity. In order to do so Pearson's correlation coefficient is commonly used (Jannach, Zanker, et al., 2010). The formula is described in equation (2.3) (Jannach, Zanker, et al., 2010). Thereby the similarity *sim* between two users a and b can be calculated, whereby $\overline{r_a}$ and $\overline{r_b}$ denote the average user ratings for items of a and b. Given that the average user rating is used in the calculation of Pearson's correlation coefficient shown in equation (2.3), the Pearson correlation coefficient also accounts for the fact that users interpret the rating scale differently, given that some users tend to rate items generally higher or lower than other users (Jannach, Zanker, et al., 2010).

$$sim(a,b) = \frac{\sum_{i \in I}(r_{a,i} - \overline{r_a})(r_{b,i} - \overline{r_b})}{\sqrt{\sum_{i \in I}(r_{a,i} - \overline{r_a})^2}\sqrt{\sum_{i \in I}(r_{b,i} - \overline{r_b})^2}} \tag{2.3}$$

Other algorithms which can be used to compute similarities between users are the *Mean Square Difference* (Shardanand and Maes, 1995, Ricci, Rokach, and Shapira, 2015, p. 55), the *Spearman Rank Correlation* (Ricci, Rokach, and Shapira, 2015, pp. 55–56) and the *Jaccard Distance* (Samer, 2017). More information can be found in Herlocker, Konstan, Borchers, et al., 1999.

Given the rating matrix declared in section 2.3.2 and the ratings shown in table 2.1, the user similarities between the active user u_a and other users u_x $(x = 1, 2, ..., 5)$ can be calculated. For illustration purposes, the similarities between u_a and other users are calculated. Given that the rating matrix in table 2.1 is sparse, it is specified that a user pair of two different users must have co-rated at least two of the same items in order to be able to calculate meaningful similarities. Therefore, meaningful similarities between u_a and other users can be calculated for the user pairs (u_a, u_1) and (u_a, u_2). For the similarity computation of the user pair (u_a, u_1) the average user rating of u_a is $\overline{r_a} = 3$ and the average user rating for u_1 is $\overline{r_1} = 2$. Equation (2.4) shows the calculation for the user pair (u_a, u_1) and equation (2.6) shows the result. It is noted that only ratings for items that both users of the user pair have rated are used in the formula described in equation (2.3), except for the calculations of the average ratings, where all of the users' ratings are considered. For the user pair (u_a, u_2) the average user rating of u_a is equal to $\overline{r_a} = 3$ and the average user rating of u_2 is $\overline{r_2} = 4.25$. The calculation of the similarity between the user pair (u_a, u_2) can be seen in equation (2.5). The result is shown in equation (2.7).

$$sim(u_a, u_1) = \frac{(4-3)*(1-2)+(2-3)*(2-2)}{\sqrt{(4-3)^2+(2-3)^2}*\sqrt{(1-2)^2+(2-2)^2}} \tag{2.4}$$

$$sim(u_a, u_2) = \frac{(4-3)*(4-4.25)+(2-3)*(3-4.25)}{\sqrt{(4-3)^2+(2-3)^2}*\sqrt{(4-4.25)^2+(3-4.25)^2}} \tag{2.5}$$

The Pearson correlation coefficient (shown in equation (2.3)) produces values between -1 and 1, whereby 1 indicates a strong positive correlation, 0 indicates no correlation and -1 indicates a strong negative correlation. As shown in equation (2.6), it is indicated that the user pair (u_a, u_1) correlates negatively with a value of approximately -0.71. On the other hand, it is indicated that the user pair (u_a, u_2) correlates positively with a value of approximately 0.55, which is shown in equation (2.7). Therefore, the user u_2 can be seen as the user that is most similar to the active user u_a and thus as the nearest neighbor of u_a.

$$sim(u_a, u_1) \approx -0.71 \tag{2.6}$$

$$sim(u_a, u_2) \approx 0.55 \tag{2.7}$$

After the k-nearest neighbors have been found (u_2 for $k = 1$) the next step is to predict ratings for items that u_a has not rated yet. One possible formula to calculate a prediction *pred* is shown in equation (2.8). The formula predicts a rating of a user a for an item i and considers the relative proximity of the nearest neighbors N and the average rating of the active user $\overline{r_a}$ (Jannach, Zanker, et al., 2010, pp. 15–16).

$$pred(a, i) = \overline{r_a} + \frac{\sum_{b \in N} sim(a, b) * (r_{b,i} - \overline{r_b})}{\sum_{b \in N} sim(a, b)} \tag{2.8}$$

Thus, a prediction for u_a can be made based on ratings of the nearest neighbor u_2 for items, which have been rated by u_2 and which have not been rated by u_a yet. Given table 2.1 these items are *Inception* and *Casablanca*. The calculation of the predicted rating for the item *Inception* can be seen in equation (2.9). Given that in this case there is only one nearest neighbor and the ratings for both items by u_2 are the same, both items have the same predicted rating of 3.75 for u_a, as shown in equation (2.10).

$$pred(U_a, Inception) = 3 + \frac{0.55 * (5 - 4.25)}{0.55} \tag{2.9}$$

$$pred(U_a, Inception) = 3.75 \tag{2.10}$$

Variations: There is a variety of approaches which aim to enhance the user-based collaborative filtering recommendation process. Some of these approaches work better in certain domains. Breese, Heckerman, and Kadie (1998), for instance, introduce *inverse user frequency*, which is based on the idea that items which are highly popular and have been rated by a large amount of users are worse indicators of user similarities than items that are less popular. Another example of an improvement is given in Herlocker, Konstan, Borchers, et al., 1999. This paper claims that there are commonly user pairs with a high user similarity, although both users have only rated a very small number of the same items, for example less than five. They found that predictions based on these similarities between user pairs are poor and showed that the prediction's accuracy can be significantly improved by penalizing similarities that are based on a small number of co-rated items. Therefore, the authors introduce a significance weight of $n/50$ which they applied to a Spearman's correlation based prediction algorithm, where n is the number of items rated by both users. For $n > 50$ they used a significance weight of 1.

Challenges and Drawbacks: Two major challenges of user-based collaborative filtering, which are described in Sarwar et al., 2001, are sparsity and scalability. Sparsity refers to sparse rating matrices, which usually pose a challenge in large systems with large amounts of different items and users. In such systems, most users have only rated a very small subset of items, often less than one percent. Thus, in large systems there might be no highly similar neighbors that recommendations can be based on for every user. This can result in poor recommendations for users without highly similar neighbors (Sarwar et al., 2001). The second main challenge described in Sarwar et al., 2001 is scalability. Given that user-based collaborative filtering recommender systems operate on the uncompressed set of data of all users and items, the systems' performance decreases with an increasing number of users and items. Therefore, user-based approaches are generally not suited for large systems with millions of users and items.

Furthermore, collaborative filtering recommendation approaches generally suffer from the *new-user problem*, which describes the problem that users who have not rated many items yet are difficult to be categorized accurately into a group of similar users (Burke, 2002). As outlined in section 2.3.6, the combination of a collaborative filtering approach with a knowledge-based approach can mitigate the new-user problem.

Another issue is that once the recommender system has created a mature user model for a user, it is generally difficult for the user to change the model. Burke (2007) illustrates this point with the example that, given a recommender system that recommends restaurants and a user who rated steakhouses positively in the past and who recently became vegetarian, the user will continue to receive recommendations for steakhouses for a long time until the user model has adapted to the user's new preferences. This is referred to as *stability vs. plasticity problem* by the author.

Another drawback of user-based collaborative filtering recommender systems with a small or medium amount of users is that there are users who do not consistently agree with any group of users. These users are called *gray sheep* and generally do not receive accurate recommendations from collaborative filtering systems (Claypool et al., 1999).

Item-Based Approach

Item-based collaborative filtering is a recommendation approach that recommends items to an active user based on the similarities between items that the active user has already rated and new items. It is based on the idea that a user's taste in items remains more or less stable over time. This implies that users are interested in similar items to the ones that they have already consumed. One advantage of item similarities over user similarities (as used in the user-based approach, see section 2.3.2) is that item similarities are generally more stable over time than user similarities. This is due to the fact that most users have rated only a small subset of all items, therefore new ratings can strongly influence existing user similarities. On the other hand, when there is a large amount of items, usually item similarities are not going to change drastically when a comparatively small amount of new items is introduced. Therefore, similarities between items can be calculated offline and stored from time to time, in order to save time in the recommendation process (Jannach, Zanker, et al., 2010). Although user similarities can also be calculated offline, it is generally not practical, since new ratings strongly influence existing similarities. Hence item-based collaborative filtering approaches are generally more scalable than user-based ones (Sarwar et al., 2001, Linden, Smith, and York, 2003).

For the recommendation process, first similarities between items, which a user has already rated and other items must be calculated. Otherwise, in the case that the item similarity calculation has been computed offline, the similarities only need to be looked up. The standard metric for the item similarity calculation is *cosine similarity* (Jannach, Zanker, et al., 2010). Alternatively, also a version of the Pearson correlation coefficient (similar to equation (2.3)) can also be used for the similarity calculation. The cosine similarity, as shown in equation (2.11), where *sim* is the similarity between two items i and j and "$\cdot$" denotes the dot-product of two vectors, views items as vectors in the m dimensional user space and computes the cosine of the angular between the two item vectors (Sarwar et al., 2001). The cosine similarity produces similarity values between 0 and 1. In this case 1 is the highest possible correlation and 0 the lowest.

$$sim(i,j) = cos(\vec{i},\vec{j}) = \frac{\vec{i}\cdot\vec{j}}{||\vec{i}||\cdot||\vec{j}||} \tag{2.11}$$

Given table 2.1, the similarities between the two items *Gladiator* and *Inception* can be calculated as shown in equation (2.12). The result can be seen in equation (2.13).

$$sim(Gladiator, Inception) = \frac{2*3+3*5+3*3}{\sqrt{2^2+3^2+3^2}*\sqrt{3^2+5^2+3^2}} \tag{2.12}$$

$$sim(Gladiator, Inception) \approx 0.98 \tag{2.13}$$

A drawback of the basic cosine similarity is that it does not take the users' rating behavior into account. Therefore, the *adjusted cosine similarity*, formally described in equation (2.14), subtracts the corresponding average user ratings $\overline{R}_u$ from each co-rated pair (Sarwar et al., 2001). The similarity values are represented by values between -1 and 1, similar to the values generated by Pearson's correlation coefficient (shown in equation (2.3)).

	$\overline{R}$	Gladiator	Inception
u_1	2	0	1
u_2	4.25	-1.25	0.75
u_5	3.67	-0.67	-0.67

Table 2.2.: This table shows the relevant information which is needed in order to compute similarities between the items *Gladiator* and *Inception* by using the adjusted cosine similarity. $\overline{R}$ denotes the average user rating.

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \overline{R}_u)(R_{u,j} - \overline{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,i} - \overline{R}_u)^2}\sqrt{\sum_{u \in U}(R_{u,j} - \overline{R}_u)^2}} \tag{2.14}$$

In order to calculate the similarities between the items *Gladiator* and *Inception* by using the adjusted cosine similarity and given table 2.1, the average ratings must be subtracted from the actual ratings of users who have rated both items. The relevant information to do this is presented in table 2.2. Equation (2.15) shows the similarity calculation between *Gladiator* ($Gl.$) and *Inception* ($In.$) and its result is given in equation (2.16).

$$sim(Gl., In.) = \frac{0 * 1 + (-1.25) * 0.75 + (-0.67) * (-0.67)}{\sqrt{0^2 + (-1.25)^2 + (-0.67)^2} * \sqrt{1^2 + 0.75^2 + (-0.67)^2}} \tag{2.15}$$

$$sim(Gladiator, Inception) \approx -0.24 \tag{2.16}$$

The result shows a negative correlation between the items *Gladiator* and *Inception*. Comparing the result from the basic cosine similarity (shown in equation (2.13)) with the result from the adjusted cosine similarity (shown in equation (2.16)) indicates how drastically a similarity between two items can change by considering average user ratings.

In order to compute predicted ratings for an item i for a user u, at first a set of k-nearest neighbors (NN) for i needs to be selected (Sarwar et al., 2001). Afterwards predicted ratings can be calculated according to the formula described in equation (2.17), which basically considers and weights how u rated similar items $j \in NN$ to i (Felfernig, Jeran, et al., 2014).

$$pred(u,i) = \frac{\sum_{j \in NN} sim(i,j) * r_{u,j}}{\sum_{j \in NN} sim(i,j)} \tag{2.17}$$

Challenges and Drawbacks: A drawback of content-based collaborative filtering systems is the *new-item problem*, which describes the problem that new items, which have not been rated by any user yet, cannot be recommended (Burke, 2002). As described in section 2.3.6, the combination of content-based recommender system approaches with knowledge-based recommender systems can mitigate this drawback. Furthermore, similar to user-based collaborative filtering approaches, users with a unique taste, which is inconsistent with the taste of other user groups, might not receive accurate recommendations from a content-based collaborative filtering recommender system. Moreover, the stability vs. plasticity problem, as described in section 2.3.2, also applies to content-based collaborative filtering.

2.3.3. Model-Based Collaborative Filtering

The basic idea behind model-based collaborative filtering is to bypass the performance bottleneck that exists in memory-based collaborative filtering systems with large amounts of users or items. As described in section 2.3.2 the bottleneck is caused due to the fact that memory-based recommender systems work on the uncompressed set of available user and item data. Model-based approaches, on the other hand, try to compress the large amounts of data into a model, which is then used in the recommendation process, instead of the whole rating matrix (Thi Do, Van Nguyen, and Loc Nguyen, 2010). This can be achieved by using statistical and machine learning methods (Thi Do, Van Nguyen, and Loc Nguyen, 2010). Thus, the recommender system's performance increases and the sparsity of the rating matrix, which poses a challenge in memory-based collaborative filtering systems, is ignored. Thi Do, Van Nguyen, and Loc Nguyen (2010) describe four common model-based approaches: *clustering*, *classification*, *latent class model* and *Markov decision process based collaborative filtering*. In the following, the commonly used model-based collaborative filtering method clustering is presented.

Clustering

Clustering approaches are based on the idea that users can be divided into groups (*clusters*) of similar users based on the feedback they provided in the past. Thus, if new items are to be recommended to a user who is part of a cluster, not all of the system's users have to be taken into account in the recommendation process, but only those who belong to the same cluster as the active user. In order to generate predictions for an active user, the average opinion of the cluster that the active user belongs to can be used to estimate the active user's opinion of unseen items (Xue et al., 2005). The process of assigning users to clusters can be done offline, therefore the recommender system does not operate on the uncompressed set of data during the recommendation process, but only on a subset of users and items.

At first a formula for the similarity computation between users needs to be chosen. Usually users are represented by rating vectors denoted $u_i = r_{i1}, r_{i2}, ..., r_{in}$. Thi Do, Van Nguyen, and Loc Nguyen (2010) describe three formulas that can be used: Minkowski distance (equation (2.18)), Euclidian distance (equation (2.19)) and Manhattan distance (equation (2.20)). The smaller the result of the similarity calculation, the more similar the users are. Alternatively Xue et al. (2005) use Pearson's correlation coefficient (see section 2.3.2, equation (2.3)).

$$distance_{Minkowski}(u_1, u_2) = \sqrt[q]{\sum_j (r_{1j} - r_{2j})^q} \tag{2.18}$$

$$distance_{Euclidian}(u_1, u_2) = \sqrt{\sum_j (r_{1j} - r_{2j})^2} \tag{2.19}$$

$$distance_{Manhattan}(u_1, u_2) = \sum_j |r_{1j} - r_{2j}| \tag{2.20}$$

Different clustering algorithms can be used to assign users to clusters based on their similarity to the other users. A commonly used clustering algorithm is *k-means*, which basically includes three steps, as described in Thi Do, Van Nguyen, and Loc Nguyen, 2010:

1. At first k users are selected randomly. Initially each of them represents a mean of a cluster. Therefore, there are k clusters and k means.
2. The similarity between each user and each cluster mean is computed. Formally, the minimal distance between a user u_i and the mean m_v of a cluster c_v, denoted $distance(u_i, m_v)$, is searched. Then, each user is assigned to the cluster whose cluster mean is closest to the user in terms of similarity. Figure 2.3 illustrates an exemplary k-means data set.
3. After step 2, new means of all clusters are computed. If either no mean of any cluster changes, or changes to the means are within a predefined error margin, which is defined in equation (2.21), where c_v ($v = 1, 2, ..., k$) describes a cluster with the mean m_v, the clustering is finished. Otherwise return to step 2.

$$error = \sum_{v=1}^{k} \sum_{u_i \in c_v} distance(u_i, m_v) \tag{2.21}$$

Challenges and Drawbacks: A challenge in model-based collaborative filtering systems is the high initial cost of creating the model. Moreover, for clustering algorithms, sparse rating matrices cause imprecision (Thi Do, Van Nguyen, and Loc Nguyen, 2010). This issue is addressed in Ungar and P. Foster, 1998. The stability vs. plasticity problem (described in section 2.3.2) is another possible drawback.

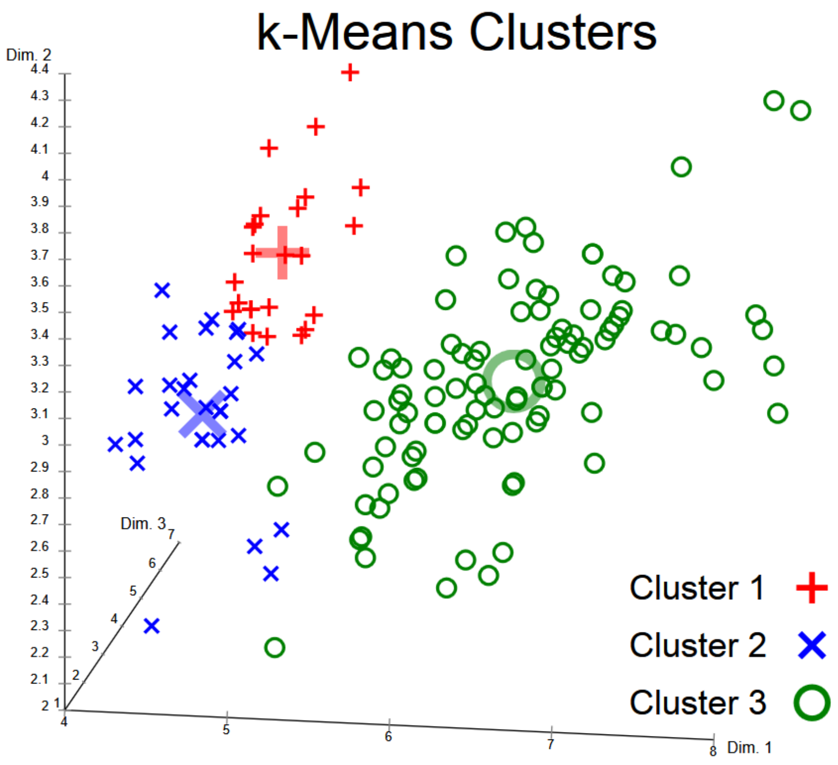


Figure 2.3.: This figure shows an exemplary k-means data set with $k = 3$ and three-dimensional vectors. The axes are labeled with integers. On the bottom right of the figure the three clusters are described. For this picture a screenshot was taken from the original source (Wikimedia Commons, 2010).

2.3.4. Content-Based Filtering

Content-based recommendation approaches recommend items based on available information about them. In relevant literature, information about items is referred to as *item attributes*, *item characteristics*, *item fields*, or *item variables* (Pazzani and Billsus, 2007). The idea behind content-based recommendation approaches is the assumption that a user's taste stays the same over time (Felfernig, Jeran, et al., 2014). Therefore, the inductive reasoning is that, given the premise that a user liked items with certain characteristics in the past, the user will also like items with the same or similar characteristics in the future. Thus, a user model is needed.

Exemplary item attributes of wristwatches can be seen in table 2.3. Usually item attributes are inserted manually, either by the provider of the item, who could be the provider of the system, or, for example, a seller on an e-commerce platform, or by the system's users. This process is associated with a certain effort and therefore involves extra costs, which can be reduced by automating the analysis of the content. An example of such an automation is a system that automatically searches documents for keywords. A content-based recommender system can then recommend documents with similar keywords which users liked in the past (Jannach, Zanker, et al., 2010).

Besides information about items, a user model for the active user is needed for the recommendation process. The user model is usually developed on the basis of a user's purchase history (or conversion history) and a user's ratings of items. In this case, the user model's accuracy increases with the number of purchases and ratings that a user made in the past. If a user has no purchase history or has not rated any items, no recommendations can be made. No information about other user's purchase history or ratings is needed, thus content-based recommender systems are able to create recommendations for a user, even if the user is the only user of the system (Jannach, Zanker, et al., 2010).

For the recommendation process the recommender system tries to find items that the active user will like based on the system's user model for the active user. In order to do so, one approach is to find similar items to the ones that the active user liked in the past among the set of items that the active user has not seen yet. Furthermore, similarities between items must be calculated,

Model	Movement	Chronograph	Case Diameter	Band Material
Watch 1	Quartz	No	38 mm	Leather
Watch 2	Quartz	Yes	43 mm	Stainless Steel
Watch 3	Automatic	No	40 mm	Stainless Steel
Watch 4	Quartz	No	35 mm	Leather
Watch 5	Automatic	Yes	45 mm	Titanium

Table 2.3.: This table shows various models of wristwatches and their item attributes. The column *Movement* describes the power source of the wristwatches. *Chronograph* is a boolean field which describes whether the watch features a timer. The column *Case Diameter* specifies the case size in mm and *Band Material* states the material from which the band is made. These attributes can be used by a content-based recommender system in order to calculate similarities between items.

which can be achieved by comparing item attributes. For example, given the items shown in table 2.3 and a user who demonstrated preferences for watches that are powered by a quartz movement in the past, watches that the active user has not seen so far which feature a quartz movement can be assigned a similarity of 1 and watches that feature an automatic movement can be assigned a similarity of 0 (Jannach, Zanker, et al., 2010). Alternatively, the similarities can be calculated by viewing items as documents and using keywords (Felfernig, Jeran, et al., 2014). Table 2.4 shows the same models of wristwatches as table 2.3, but presents the item attributes in the form of keywords. The similarities between different items can be calculated by using the *Dice coefficient* described in equation (2.22), where the similarity *sim* between two items i_a and i_b is calculated by utilizing keywords (Jannach, Zanker, et al., 2010). Another common approach is the *keyword-based Vector Space Model*, which represents text documents as vectors in an n-dimensional space, where n denotes the total number of different terms in the whole document collection and each dimension corresponds to a single term (for more information see Lops, Gemmis, and Semeraro, 2011 and Jannach, Zanker, et al., 2010). After the similarities have been calculated, k-nearest neighbor approaches (similar to section 2.3.2) can be used as a simple method of finding recommendations based on past user preferences.

$$sim(i_a, i_b) = \frac{2 * |keywords(i_a) \cap keywords(i_b)|}{|keywords(i_a)| + |keywords(i_b)|} \tag{2.22}$$

Model	Keywords
Watch 1	quartz, chronog_no, cd_38mm, band_leather
Watch 2	quartz, chronog_yes, cd_43, band_ssteel
Watch 3	automatic, chronog_no, cd_40, band_ssteel
Watch 4	quartz, chronog_no, cd_35, band_leather
Watch 5	automatic, chronog_yes, cd_45, band_titanium

Table 2.4.: This table shows various wristwatch models (items) and keywords that describe them. The keywords can be used to calculate similarities between the items.

Challenges and Drawbacks: Content-based recommender systems face the following three drawbacks, as described in Lops, Gemmis, and Semeraro, 2011:

Limited content analysis Content analysis, automated or manually, may fail to capture certain item nuances. For example, the automated analysis of a news article may fail to recognize sarcasm, jokes or a bad choice of words (Lops, Gemmis, and Semeraro, 2011). Also words often change their meaning in a given context, for example the saying "out of the frying pan into the fire" has usually nothing to do with an actual pan or fire.

Over-specialization Content-based recommender systems generally do not recommend serendipitous and novel items, since recommendations only consist of items that are similar to the ones that the active user liked in the past (Lops, Gemmis, and Semeraro, 2011). Therefore, the recommendation approach is usually not suitable for domains in which serendipity and novelty are important properties of the recommender system.

New-user problem Given that a user model is needed in order to find similar items to the ones which the active user liked in the past, a certain amount of user ratings or knowledge about user preferences is needed in order to produce high quality recommendations[10] (Lops,

[10]However, content-based recommender systems can recommend items to a user as soon as feedback about at least one item has been provided by the user. Although recommendations based on only one purchase or rating might not be the most accurate ones, content-based recommender systems offer the advantage that they are able to provide reasonable recommendations with minimal feedback.

Gemmis, and Semeraro, 2011). Thus, the recommendation quality is generally lower in content-based recommender systems for new users or for users represented by poor user models.

In addition to the three drawbacks stated in Lops, Gemmis, and Semeraro, 2011, the stability vs. plasticity problem, as described in section 2.3.2, also poses a challenge for content-based recommender systems.

2.3.5. Knowledge-Based Approaches

Besides collaborative filtering and content-based recommendation techniques, knowledge-based recommendation techniques are another major area of recommender systems research. Contrary to the other two techniques, with knowledge-based recommendation approaches user ratings for items are not an essential ingredient for the recommendation process. Instead, deep knowledge about the item set is used to customize items so that they match the user's requirements and preferences. Knowledge-based recommender systems emphasize a user's situation and how recommended items can satisfy a user's particular needs (Felfernig and Burke, 2008). Typically, a user states requirements and the recommender system tries to find adequate items. Requirements can be stated in multiple ways, for example by asking a user questions about the user's taste and preferences, by presenting tweakable parameters to the user, or by offering the user options to choose from. For instance in the car industry a car can be described as sporty, efficient or family-friendly (Jannach, Zanker, et al., 2010). If there are no items fulfilling the requirements, the requirements need to be changed (Jannach, Zanker, et al., 2010). Thus, knowledge-based approaches are ideally suited to operating with complex items, for which users usually have specific demands and expectations. Laptops, for example, are complex items with many different characteristics, as shown in table 2.5. Users might look for specific kinds of laptops, such as laptops that only need to be capable of running basic office applications, powerful workstations, gaming laptops that can handle current games at speeds of over 60 frames per second, or lightweight laptops, primarily for mobile usage. Collaborative filtering and content-based recommendation techniques are generally ill-suited to meeting such specific user requirements. Another advantage of knowledge-based

Id	Price	Size	Weight	CPU	RAM	Graphics	Storage	Battery
l_1	449	13.3	1.7	3.7	4	2.4	small	long
l_2	679	17.3	3.1	5.9	8	5.1	large	short
l_3	599	14	1.8	5.1	4	2.6	medium	long
l_4	999	15.6	2.1	7.6	8	7.9	large	medium
l_5	769	17.3	2.8	6.8	8	6.7	medium	short

Table 2.5.: This table shows various laptop models (items) and the item's attributes (some less important attributes are left out because of limited space). The item attributes are presented in the following units: price in dollar, size in inches, weight in kilograms, CPU and graphics on a scale from 0 to 10, where 10 is the highest score and 0 the lowest, RAM in gigabytes, and storage and battery in rough sizes.

recommendation approaches over collaborative filtering and content-based recommendation techniques is that there is no new-user and new-item problem in knowledge-based recommender systems, since users express their preferences during the recommendation process. Thus, there is no need for existing ratings. Since knowledge-based recommendation approaches do not rely on past ratings, they also do not face the problem of previous misleading user ratings. For instance, a user who rated a laptop highly five years ago is most probably not looking for a model with similar specifications, as the old model's hardware is outdated by today's standards. Knowledge-based recommendation techniques can be divided into different approaches, two of which are *constraint-based approaches* (Felfernig and Burke, 2008) and *case-based approaches* (Burke, 2000, Jannach, Zanker, et al., 2010), which are presented in the following sections.

Constraint-Based Approach

Constraint-based approaches find suitable items based on a set of constraints which is explicitly defined by the user, for example in the form of filters that can be tweaked. An example of such an approach can be seen in figure 2.4, which shows recommendations based on constraints from the Trivago website[11], which compares hotel prices from different websites in

[11] *Trivago* 2019.

order to find hotels for a user, which fulfill the user's needs at the best price. If there are no items that meet the defined constraints, usually a solution is provided how to remove or loosen constraints (Jannach, Zanker, et al., 2010). Figure 2.5 shows how Trivago's website reacts if there are no hotels that comply with the specified constraints. The system recommends removing filters in order to be able to find items that meet the search criteria. Alternatively the system suggests viewing the presented options.

The recommendation process can be regarded as constraint satisfaction problem, according to Jannach, Zanker, et al. (2010). Therefore, a *recommender knowledge base* needs to be described (Felfernig and Burke, 2008). This can be achieved by explicitly defining the following terms, which are explained with the help of table 2.5. The explanations of the terms are adopted from Felfernig and Burke, 2008.

Customer properties, denoted by $u_i \in U$, describe all possible customer requirements. Possible customer requirements are for instance:

- $u_1 : max\_price = integer$
- $u_2 : min\_cpu = integer$
- $u_3 : min\_storage = low, medium, high$

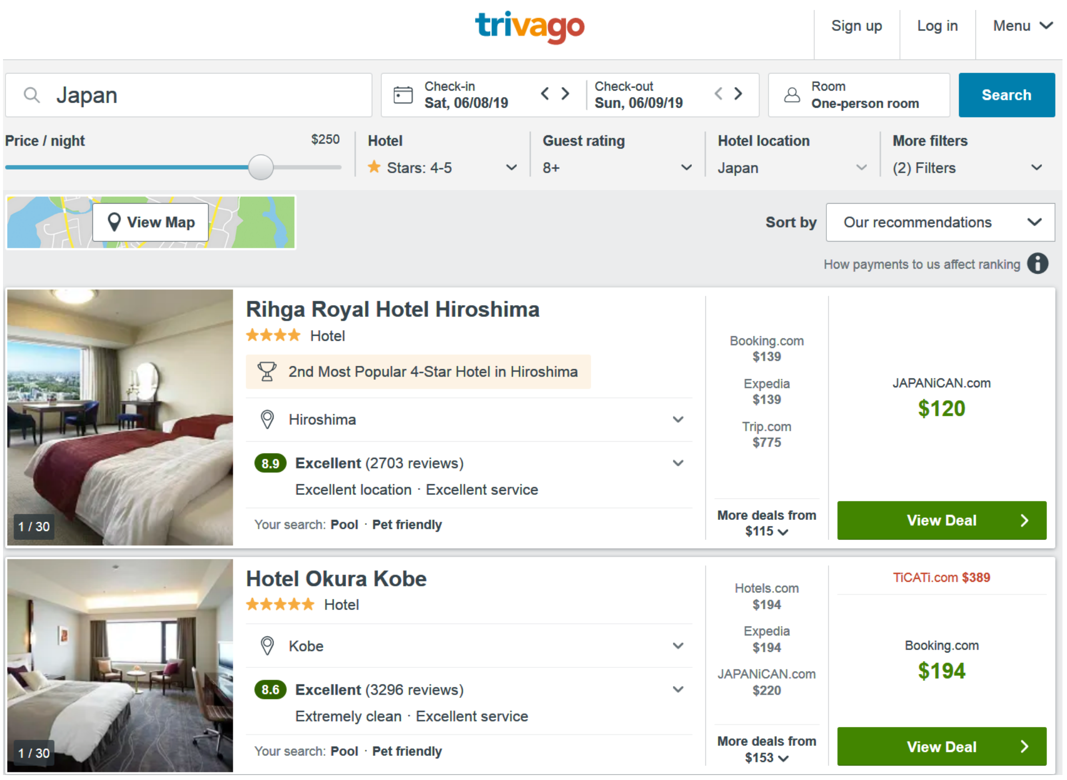
Product properties, denoted by $p_i \in P$, describe the product assortment, such as:

- $p_1 : id = l_1$
- $p_2 : price = 449$
- $p_3 : size = 13.3$

(In)Compatibility constraints, denoted by $comp_i \in COMP$, describe relations between product properties, for example:

- $comp_1$: a *storage* greater than *large* requires a *price* greater than 300
- $comp_2$: a *weight* less than 2 requires a *size* less than 14
- $comp_3$: a *graphics* performance of greater than 5 requires a *CPU* performance of greater than 3

Product constraints, denoted by $prod_i \in PROD$, restrict instantiations of variables in P. In order to enumerate the offered set of products, product constraints are used. They are similar to table 2.5, with the exception that in



Japan

Check-in Sat, 06/08/19

Check-out Sun, 06/09/19

Room One-person room

Price / night $250

Hotel Stars: 4-5

Guest rating 8+

Hotel location Japan

More filters (2) Filters

Sort by Our recommendations

How payments to us affect ranking

Rihga Royal Hotel Hiroshima
★★★★ Hotel
2nd Most Popular 4-Star Hotel in Hiroshima
Hiroshima
8.9 Excellent (2703 reviews)
Excellent location · Excellent service
Your search: Pool · Pet friendly
1 / 30

Booking.com $139
Expedia $139
Trip.com $775
More deals from $115

JAPANiCAN.com $120

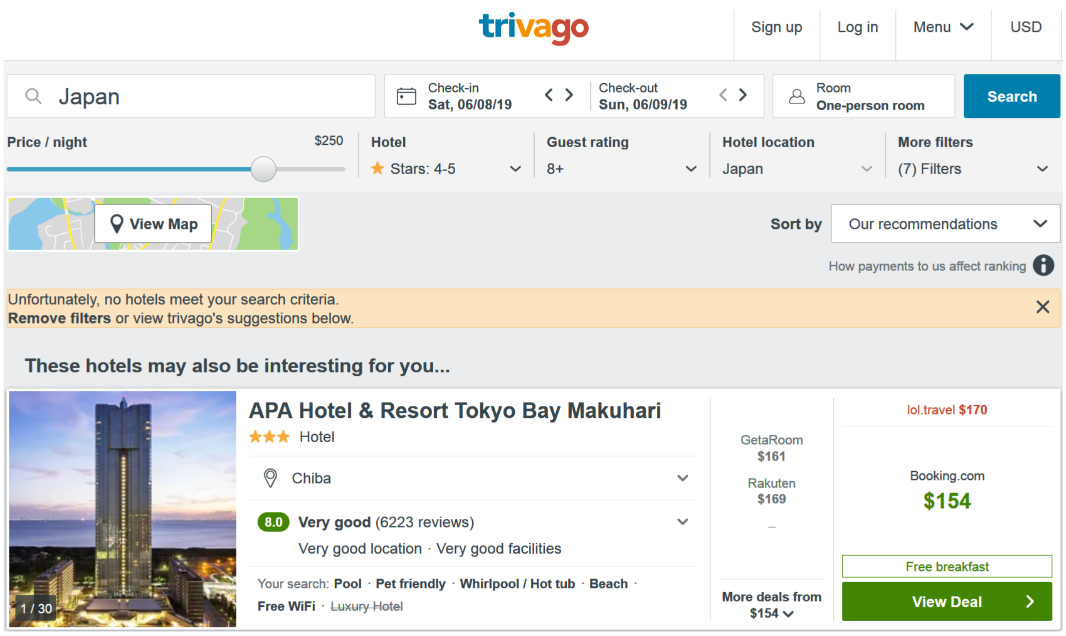
Hotel Okura Kobe
★★★★★ Hotel
Kobe
8.6 Excellent (3296 reviews)
Extremely clean · Excellent service
Your search: Pool · Pet friendly
1 / 30

Hotels.com $194
Expedia $194
JAPANiCAN.com $220
More deals from $153

TiCATi.com $389

Booking.com $194

Figure 2.4.: This figure shows constraint-based recommendations of Trivago (*Trivago* 2019). The constraints that can be tweaked can be seen on top, such as *Price/night*. In addition to the period of travel and the room category, it has been specified that potential hotels must cost less than 250$ per night, need to have a star rating of at least 4 and a guest rating of 8 or higher. The hotel must be located in Japan. Furthermore, two additional filters have been applied (the hotel must have a pool and be pet-friendly). The recommendations can be seen below the constraint specification. The website has been accessed in private mode.


trivago
Sign up | Log in | Menu | USD

Japan
Check-in Sat, 06/08/19
Check-out Sun, 06/09/19
Room One-person room
Search

Price / night $250
Hotel: Stars: 4-5
Guest rating: 8+
Hotel location: Japan
More filters: (7) Filters

View Map
Sort by: Our recommendations
How payments to us affect ranking

Unfortunately, no hotels meet your search criteria.
Remove filters or view trivago's suggestions below.

These hotels may also be interesting for you...

APA Hotel & Resort Tokyo Bay Makuhari
★★★ Hotel
Chiba
8.0 Very good (6223 reviews)
Very good location · Very good facilities
Your search: Pool · Pet friendly · Whirlpool / Hot tub · Beach · Free WiFi · ~~Luxury Hotel~~
1 / 30

GetaRoom $161
Rakuten $169
More deals from $154
lol.travel $170
Booking.com $154
Free breakfast
View Deal

Figure 2.5.: This figure shows the content that Trivago's recommender system (from *Trivago* 2019) presents if no hotels can be found in accordance with the user's constraints. The user is asked to remove filters or to view Trivago's suggestions.

the column *id*, $prod_i$ is used for the enumeration instead of $l_i \in L$ (where L denotes the set of offered laptops).

Filter constraints, denoted by $filt_i \in FILT$, define how customer requirements translate to product properties, for example:

- $filt_1$: the product *price* must be less than the *max_price* defined by the customer
- $filt_2$: the *size* of the product must be equal to the *size* defined by the user
- $filt_3$: for a product to be *light* it must have a *weight* of less than 2

The recommendation process, as defined in Felfernig and Burke, 2008, is the process of finding a complete assignment to the variables U and P, so that all constraints specified in $CR \cup COMP \cup FILT \cup PROD$, whereby CR is a set of customer requirements, are satisfied. For example, given the customer requirements $U = (u_1 : max\_price = 600, u_2 : min\_cpu = 5, u_3 : min\_storage = medium)$ and products $P = (prod_1, prod_2, prod_3, prod_4, prod_5)$ similar to the items described in table 2.5, the complete assignment to the variable P is $P = prod_3$, whereby $prod_3$ corresponds to the item with the identifier l_3 from table 2.5. A more detailed discussion about constraint satisfaction problems can be found in Tsang, 1993.

Challenges and Drawbacks: The main drawback of knowledge-based recommender systems is that it is difficult to acquire deep domain knowledge and detailed product information and keeping it up-to-date manually, or by automation. Moreover, the ability of a knowledge-based recommender system to make suggestions is static, which means that the recommender system does not learn and increase in quality over time, contrary to collaborative filtering and content-based approaches (Burke, 2002).

Case-Based Approach

Case-based recommendation approaches are based on finding the items that are most similar to the item the user is looking for. This task usually requires an understanding of the domain in which the recommender system operates and deep domain knowledge (Felfernig and Burke, 2008). For example, given an online shop which sells laptops as shown in table 2.5, a user might specifically look for a lightweight and mobile laptop. Therefore, a case-based recommender system with domain knowledge might try to find laptops with low values in *size* and *weight* and high values in *battery*. This description fits laptops with the identifiers $id = l_1$ and $id = l_3$. Similar to constrained-based approaches (see section 2.3.5), in the event that no items can be recommended on the basis of the user's requirements, case-based approaches also support the user in conducting minimal changes to the set of user requirements in order to be able to find items that meet the user's requirements (Felfernig and Burke, 2008).

The similarity of an item i to the requirements $r \in REQ$, as described in Jannach, Zanker, et al., 2010, is called *distance similarity*. Equation (2.23) shows how the distance similarity is commonly defined, where w_r describes the importance weight for a user requirement $r \in REQ$ and $sim(i,r)$ represents the distance between the r and the corresponding *item attribute value* $\phi_r(i)$ (Jannach, Zanker, et al., 2010).

$$similarity(i, REQ) = \frac{\sum_{r \in REQ} w_r * sim(i,r)}{\sum_{r \in REQ} w_r} \tag{2.23}$$

In order to calculate $sim(i,r)$, three different kinds of item attributes have to be considered (Jannach, Zanker, et al., 2010). Firstly there are attributes of the category *more-is-better*, which refers to item properties, that users generally want to maximize. Given the items in table 2.5, customers usually want to maximize CPU, RAM, graphics performance, storage and battery life. The formula to calculate the similarity between i and r for item attributes of the category more-is-better is shown in equation (2.24) (Jannach, Zanker, et al., 2010).

$$sim(i,r) = \frac{\phi_r(i) - min(r)}{max(r) - min(r)} \tag{2.24}$$

Secondly, there are item attributes, which users generally want to minimize, which are called *less-is-better*. Usually customers want to minimize the price of an item, or the weight of items that have to be carried around a lot, such as smartphones or laptops. Equation (2.25) shows the formula to calculate similarities between less-is-better item attributes (Jannach, Zanker, et al., 2010).

$$sim(i,r) = \frac{max(r) - \phi_r(i)}{max(r) - min(r)} \tag{2.25}$$

The third kind of item attributes which have to be considered are item attributes that should neither be maximized, nor minimized but met as close as possible, for example in the case that a user is looking for an item in a specific size. The corresponding similarity formula is shown in equation (2.26) (Jannach, Zanker, et al., 2010).

$$sim(i,r) = 1 - \frac{|\phi_r(i) - r|}{max(r) - min(r)} \tag{2.26}$$

Challenges and Drawbacks: Generally case-based approaches face the same challenges and drawbacks as constrained-based approaches, as described in section 2.3.5.

2.3.6. Hybrid Approaches

Given that different recommendation approaches have different strengths and shortcomings, hybrid recommendation approaches are based on the idea that the combination of two or more recommendation approaches can eliminate or reduce the shortcomings and combine the strengths of the individual approaches (Burke, 2002). Collaborative filtering, for instance, relies on the existence of multiple user ratings for an item, as otherwise the item cannot be recommended with a reasonable accuracy. On the other hand, collaborative filtering algorithms can recommend serendipitous items based on latent factors (Ricci, Rokach, and Shapira, 2015). In contrast, content-based recommendation approaches can recommend items that have not received any ratings yet, but generally fail to recommend items that are dissimilar to the items that a user rated in the past. Hence the combination of a collaborative filtering recommendation approach and a content-based recommendation approach can be beneficial. Alternatively, knowledge-based recommendation approaches do not rely on past user ratings at all, thus they do not suffer from the new-user problem and the new-item problem. Therefore, knowledge-based recommendation approaches are suitable for the combination with collaborative filtering and content-based approaches, according to Burke (2002).

One way to combine two or more recommendation approaches is by first using the individual recommendation approaches separately and then combining the outcomes by weighting the individual results in order to make predictions. One example of such an approach is *P-Tango* (Claypool et al., 1999), which combines a collaborative filtering and a content-based recommendation approach. After the computation of the individual approaches, a weighted average is taken in order to compute aggregated scores for items. The weights are user-specific, since the recommendation quality of the individual approaches varies depending on the available information about users and items. An overview of different ways of combining multiple recommendation approaches is shown in table 2.6. For further information on hybrid recommendation approaches, see Burke, 2002 and Jannach, Zanker, et al., 2010.

2.4. Evaluation

According to Shani and Gunawardana (2011) there are three types of evaluation settings for the evaluation of recommender systems: offline experiments, user studies and online experiments. In the following these three types are discussed. Further information about the evaluation of recommender systems can be found in Shani and Gunawardana, 2011 and Ricci, Rokach, and Shapira, 2015.

Hybridization Method	Description
Weighting	The scores (or votes) of several recommendation techniques are combined together to produce a single recommendation.
Switching	The system switches between recommendation techniques depending on the current situation.
Mixed	Recommendations from several different recommenders are presented at the same time
Feature combination	Features from different recommendation data sources are thrown together into a single recommendation algorithm.
Cascade	One recommender refines the recommendations given by another.
Feature augmentation	Output from one technique is used as an input feature to another.
Meta-level	The model learned by one recommender is used as input to another.

Table 2.6.: This table describes various ways of combining two or more recommendation approaches. It is adopted from Burke, 2002 with kind permission of R. Burke.

2.4.1. Offline Experiments

In offline experiments, previously collected data of users, their behavior and their ratings are used for testing purposes. These tests are based on the idea that past user behavior can be used to evaluate new recommender systems or changes in existing recommender systems. It is assumed that how a user behaved in the past is similar enough to how the user would have behaved under the circumstances tested by the offline experiment, as described by Shani and Gunawardana (2011). The authors state that in light of the low cost of offline experiments and due to the fact that no real users are involved, they are ideally suited to testing a wide range of different approaches and filtering out inappropriate ones. As outlined by Shani and Gunawardana (2011), in this case only a small set of candidate approaches remain, which can be tested more carefully by user studies and online experiments. According to the authors, this way different values for a given parameter can be tested and compared in order to see which values perform best.

The simulation of user behavior is commonly achieved by utilizing historical user data. A part of the data is hidden, which is used to simulate knowledge about a user's future behavior, while the rest of the historical user data is used in the recommendation process, as outlined in Shani and Gunawardana, 2011. The authors claim that the recommender system can then be tested by predicting ratings for items which are part of the hidden historical user data. For those items the actual user ratings are known and thus the ratings predicted by the recommender system can be compared to the actual user ratings from the hidden user data. This way the performance of the recommender system can be evaluated.

A downside of offline experiments is that the data about past user behavior obviously stems from the original system, which does not include the changes to the system that are tested in the offline experiment. Therefore, it is not possible to draw conclusions about the influence on user behavior because of the changes made to the system (Shani and Gunawardana, 2011).

2.4.2. User Studies

User studies, as described in Shani and Gunawardana, 2011, are experiments, for which a set of test subjects are recruited in order to perform certain tasks while their behavior is observed and recorded. The article states that there is also the possibility of asking the test subjects questions about desired information before, during and after the tasks, which can provide further information, such as the level of user satisfaction or insights into usability aspects.

In user studies about recommender systems, a typical use case, as described in Shani and Gunawardana, 2011, is to test different versions of a system. The article illustrates the point with the example that test subjects might be asked to navigate through a set of items. In some cases recommendations are given and in other cases no recommendations are given. Thus, the impact of recommendations on user behavior can be studied.

A definite advantage of user studies is the opportunity to put questions to the test subjects, which makes it possible to learn about the test subjects' motivations directly. This is not feasible with offline and online experiments. In this case generally only statements about correlations but not about motivations can be made (Shani and Gunawardana, 2011). On the other hand, Shani and Gunawardana (2011) state that user studies are costly, given that test subjects are either volunteers who are not readily available, or they need to be compensated for their time and effort. Thus, only a limited amount of tests can be conducted. In the light of the high cost of user studies, pilot studies may be conducted in order to test a user study for bugs and malfunctions (Shani and Gunawardana, 2011). Furthermore, according to Shani and Gunawardana (2011), there is the risk that the test subjects are biased, which might be higher in the event that test subjects are volunteers, as people might be more likely to volunteer for studies in which they are interested in. Hence, as claimed by the authors, it is important that the test subjects are as similar as possible to the actual system users.

2.4.3. Online Experiments

Online experiments within the context of recommender systems, as portrayed in Shani and Gunawardana, 2011, describe the process of comparing two or more different versions of a system in an online setting with the aim of finding out which version performs best. To do so, users are assigned to different groups and a different version of the system is presented to each group. Usually, the users are randomly assigned to groups in order to avoid potential biases. After a certain amount of time, the experiment is completed and the logged data can be evaluated.

Compared to offline experiments and user studies, online experiments reflect real-world situations most accurately, as real tasks are performed by real users, who usually do not know that they are participants in an experiment, as mentioned by Shani and Gunawardana (2011). Therefore, the results provided by online experiments can be regarded as more trustworthy than the results obtained through offline experiments and user studies (Shani and Gunawardana, 2011).

However, online experiments that present poor and unrefined versions of the system to user groups may lead to a permanent loss of users, which might not be acceptable in commercial systems (Shani and Gunawardana, 2011). Thus, before a version of the system is presented to real users, offline experiments and user studies can be conducted in advance to make sure that the presented version is reasonable and to minimize risk (Shani and Gunawardana, 2011).

Furthermore, online experiments usually do not involve questionnaires, which can be used to gain further information about the participants' motivations. Therefore, as stated in Shani and Gunawardana, 2011, it is important to keep all other variables, that are not tested in the online experiment, constant. The authors explain that this is necessary because in the event that multiple variables are changed, it is impossible to attribute differences in user behavior to certain changes, as the differences may be the result of any combination of changed variables. For instance, if a change to a recommender system algorithm is introduced that aims to improve prediction accuracy and at the same time a change is made to the user interface, it will not be possible to draw conclusions whether observed changes in user behavior are the result of the changed algorithm, the changed user interface, or a combination of both.

3. Catrobat

This chapter focuses on different aspects of Catrobat, the organization for which the practical part of this thesis is carried out. First of all, Catrobat as an organization is introduced. Then *Pocket Code*, Catrobat's mobile application is presented. Afterwards an overview of the *sharing platform*[1] is given, a place where Pocket Code users can share their creations. Finally, Catrobat's recommender system and its individual components as well as the recommendation process are described.

3.1. The Catrobat Organization

The Catrobat organization aims to motivate children and adolescents to start programming[2] and to enhance their computational and problem-solving skills. The organization was founded in 2010 at the Institute of Software Technology of the Technical University of Graz and is a non-profit open-source project with over 480 contributors as of June 2019 (Black Duck Software, Inc., 2019). Catrobat's concept is to enable their users to create, share and modify programs in an easy and intuitive way. This is achieved by using Catrobat, a visual programming language, named after the organization, which allows users to write code by using graphical elements rather than writing traditional text-based code. In June 2019, at the time of writing this thesis over 88,000 programs[3] have been uploaded to Catrobat's sharing platform[4].

[1] International Catrobat Association, 2019b.
[2] International Catrobat Association, 2019a.
[3] Painsi, 2019.
[4] International Catrobat Association, 2019b.

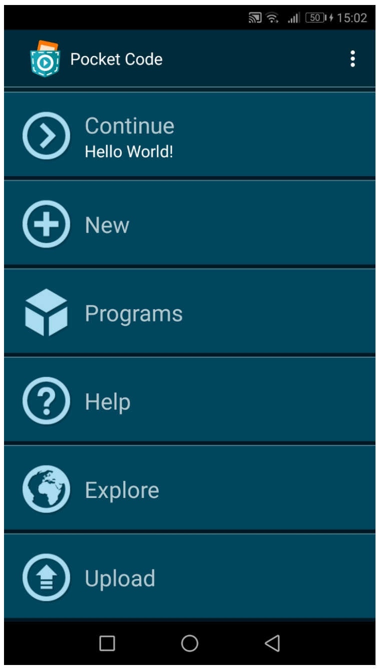


Figure 3.1.: This figure shows Pocket Code's main menu screen.

3.2. Pocket Code

Pocket Code is the name of the application (app) which is used to program in the Catrobat programming language. It is available for Android and iOS and features an integrated development environment (IDE) for the Catrobat programming language. Figure 3.1 shows the structure of the app's main menu. In this menu a user can create a new program or continue working on an existing one. The *Programs*-button allows the user to browse through created and downloaded programs. The *Help*-button guides the user to a help and tutorial page. Via the *Upload*-button users can upload programs that they have created or modified to Catrobat's sharing platform, which is further described in section 3.4. The *Explore*-button enables the user to browse through the sharing platform.

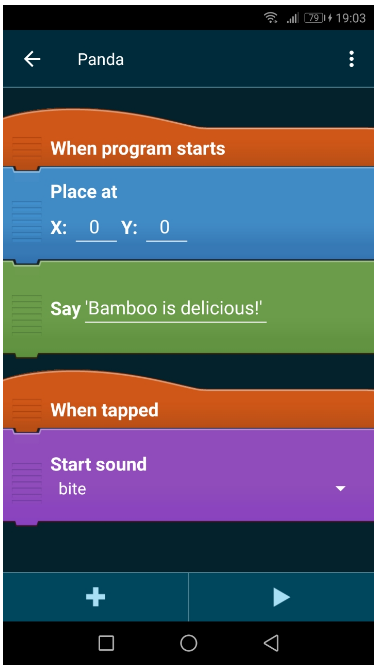


Figure 3.2.: This figure shows two exemplary scripts in the Catrobat programming language within the Pocket Code app. The scripts define the object's behavior. *Panda* is the object's title.

3.3. Catrobat Programming Language

The Catrobat programming language is a visual programming language, which is heavily influenced by Scratch[5], a well-known project of the Lifelong Kindergarten Group at the MIT Media Lab. Both programming languages can be used without writing a single line of code and instead only rely on graphical elements. A Catrobat program usually consists of objects and bricks. A brick is a single instruction for an object that can be combined with other bricks. Multiple combined bricks form scripts. Figure 3.2 shows an example of two scripts which have been combined from multiple bricks.

The brick's color indicates which brick category it belongs to. In total there are seven different brick categories.

[5]Lifelong Kindergarten Group, MIT Media Lab, 2019.

- Event bricks - Detect events and start scripts.
- Control bricks - Control the logical flow of other bricks.
- Motion bricks - Manipulate the object's position and physics.
- Sound bricks - Create sounds and let the object speak.
- Looks bricks - Manipulate the object's looks.
- Pen bricks - Enable drawing on the screen.
- Data bricks - Create and manage variables and lists.

Each script starts with an event brick, which dictates when a script is to be executed. The first script shown in figure 3.2 starts with the event brick "When program starts". Afterwards the object's position on the screen is set with the motion brick "Place at ...". The looks brick "Say ..." lets a speech bubble appear that shows the string which has been declared in the brick. The second script in figure 3.2 is executed when the object is tapped and plays the sound "bite".

When a program is started, the Pocket Code app transforms the visual code into *Extensible Markup Language* (XML), which can be interpreted by Pocket Code. On the other hand, when a program is opened in the IDE, the program's XML code is transformed into the visual Catrobat programming language. Figure 3.3 shows a running program. In addition to the code showcased in figure 3.2 a background image was set.

3.4. The Sharing Platform

Catrobat's sharing platform[6], also referred to as *community website*, is accessible via the Pocket Code app and via a browser. It is mainly used to browse through programs that other users have uploaded and to upload programs written in the Catrobat programming language. In order to present a variety of different programs, multiple program sections exists:

- Featured - Shows a selection of featured programs.
- Newest Programs - Shows the most recently uploaded programs.

[6]International Catrobat Association, 2019b.

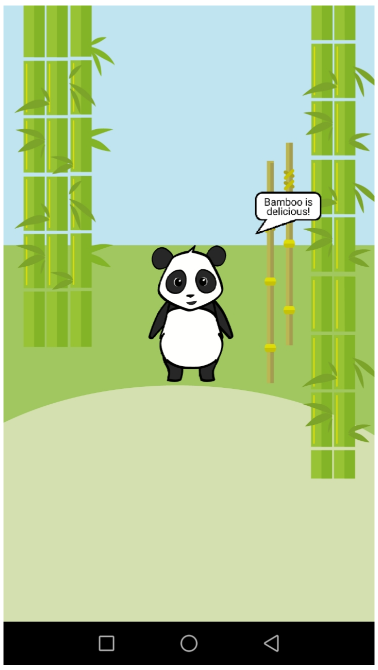


Figure 3.3.: This figure shows a running program. A background image has been set and the panda object has been positioned in the middle of the screen. A looks brick has been used to let the object express a thought.

- Recommended Programs - Shows non-personalized or user-specific recommendations.
- Most Downloaded - Shows the most downloaded programs of all times.
- Most Viewed - Shows the most viewed programs of all times.
- Random Programs - Shows random programs.

In order to upload programs, a user account needs to be created either in the Pocket Code app or directly on the sharing platform. Creating a user account enables a user to access the user profile, to upload programs and to manage uploaded programs. There is also a notification system that notifies the user if one of the user's uploaded programs receives a comment from another user. In total, Catrobat's community platform has over 80,000 registered users and over 88,000 uploaded programs. The majority of the activity is performed by mobile users.

Technical Details: The sharing platform is an object-oriented project, which is written in *PHP: Hypertext Preprocessor*[7] (PHP), a scripting language. It uses the *Symfony framework*[8]. For the database *MySQL*[9] is used. *Doctrine*[10] is used as an object-relational mapper, whereby a class that is mapped by doctrine is referred to as *entity*.

3.4.1. Remixes

A special feature of Catrobat is the possibility of modifying downloaded programs. When a user downloads a program from the sharing platform and modifies it, the result is called a *remix*. In many other domains, like in Google's *Play Store*[11], downloading an item and uploading it modified will be seen as a breach of the terms of service[12]. However, on Catrobat's

[7]PHP Group, 2019.
[8]*Symfony Homepage* 2019.
[9]Oracle Corporation, 2019.
[10]*Doctrine* 2019.
[11]Google LLC, 2019c.
[12]Google LLC, 2019a.

sharing platform it is not only allowed, but users are encouraged to do so[13]. The underlying thought is that having direct access to the code of other programs and being able to experiment with it is an opportunity to learn. At the time of writing this thesis, about 25.5% of all Catrobat programs were remixes[14].

3.5. Recommender System

On the sharing platform there are various sections that do not change based on the condition whether a user is logged in or not, namely *Featured*, *Newest Programs*, *Most Downloaded*, *Most Viewed* and *Random Programs*. The section *Recommended Programs* only shows the same programs to guest users (see section 2.3.1). For logged in users the recommendations are personalized, hence the section *Recommended Programs* shows user-specific recommendations. Catrobat's recommender system relies on user-based collaborative filtering, which is described in section 2.3.2.

3.5.1. Like Rating System

As outlined in section section 2.3.2, user-based collaborative filtering needs user feedback in order to compute how similar users are to one another. Therefore, Samer (2017) implemented a *like rating system*, which enables users to press one of four different *like* buttons to express that they like a program on the details page of a program. Figure 3.4 showcases the like rating system. The different like buttons are *Thumbs up*, *Smile*, *Love* and *Wooow!*. However, the recommendation algorithm does not differentiate between different types of likes. Likes are positive-only, unary feedback, hence from the perspective of the recommendation algorithm there are only two options, a user either liked a program or not. If a user liked a program, the obvious conclusion will be that the user actually liked the program. On the other hand, if a user has not liked a program, the conclusion that

[13]International Catrobat Association, 2019c.
[14]Painsi, 2019.

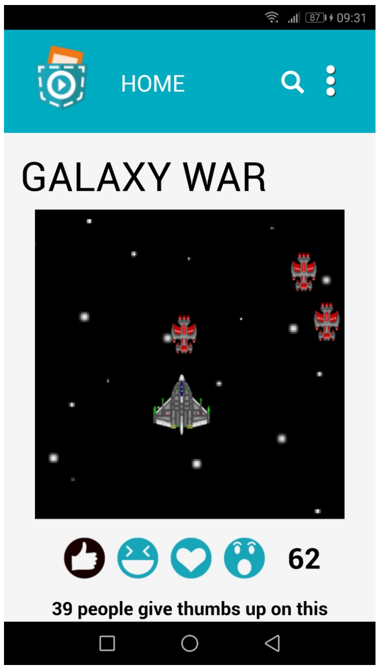


Figure 3.4.: This figure showcases the like rating system. The program *Galaxy War* received 62 likes in total. A *thumbs up* has been given, which causes Pocket Code to show the number of people who have chosen the same type of like, in this case 39.

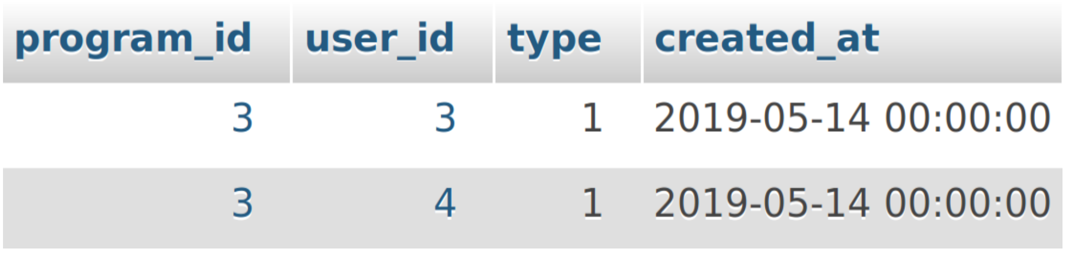
the user does not like the program may be erroneous, since the absence of positive-only feedback does not imply negative feedback. For example, a reasonable explanation why a user has not liked a program via the like rating system, which the user in fact likes, is that the user might simply have forgotten to use the like rating system. Another reason might be that the user does not want to make the effort to like the program, since it is necessary to navigate to the details page of a program in order to use the like rating system.

In total, nearly 4,500 users have liked at least one program. The average number of likes of users who liked at least one program is approximately 1.9. Table 3.1 shows how many users liked how many programs.

A *like* is implemented as an entity, which is basically a class that is mapped

Number of Users	Number of Likes
3029	1
711	2
260	3
137	4
91	5
217	6+

Table 3.1.: This table shows how many users have liked certain number of programs.



program_id	user_id	type	created_at
3	3	1	2019-05-14 00:00:00
3	4	1	2019-05-14 00:00:00

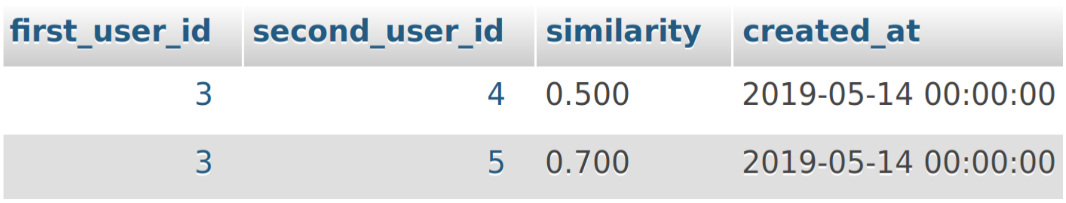
Figure 3.5.: This figure shows how the like entity is stored in the database. It can be seen that the program with the identifier 3 has received two likes from two users with the identifiers 3 and 4. It also shows the types of likes as well as the time, when the likes were received. In this case both likes were of the *thumbs up* type.

by doctrine[15], as mentioned in section 3.4. It includes information about who clicked the like button, which program was liked, which type of like was used and at what time it was received as shown in figure 3.5.

3.5.2. User Similarities

The recommender system utilizes data generated by the like rating system to compute how similar users are to one another, which is referred to as *user similarities*. All users who have liked at least one program are considered in the calculation of the user similarities. Due to the fact that nearly 4,500 users have liked at least one program at the time of writing this thesis, computing user similarities between the active user and all other users every time a new recommendation is generated is not feasible. Therefore, a

[15] *Doctrine* 2019.



first_user_id	second_user_id	similarity	created_at
3	4	0.500	2019-05-14 00:00:00
3	5	0.700	2019-05-14 00:00:00

Figure 3.6.: This figure shows how user similarities are stored in the database. In this case there are two user similarities between two user pairs. The user pair with the identifiers 3 and 4 has a similarity of 0.5, while the user pair with the identifiers 3 and 5 has a similarity of 0.7. Furthermore, the time when the user similarities have been created is stored.

method of updating all user similarities at regular intervals and of storing them in the database has been adopted (Samer, 2017, pp. 95–97). The user similarity between two users is computed by using the *Jaccard distance*, which is a similarity metric that is well suited to calculating similarities between users in case of unary feedback. Thereby the distance between two users is represented by values between 0 and 1, whereby 0 represents no similarity and 1 represents the highest possible similarity. It is calculated by dividing the number of programs which both users like by the total number of programs which any of both users like. Formally the Jaccard distance J can be expressed as shown in equation (3.1), where A and B denote the set of programs that a user u_a and a user u_b like respectively (Samer, 2017, Liu et al., 2014).

$$J(u_a, u_b) = \frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

User similarities are implemented as entities, whereby a user similarity consists of two user identifiers, a similarity value between 0 and 1 and a timestamp, which shows when the entity has been created. Their representation in the database is shown in figure 3.6. The code of the user similarity calculation is shown in code D.1.

3.5.3. Recommendation Process

This section gives a brief outline of the recommendation process. At first the non-personalized version of Catrobat's recommender system is portrayed.

Afterwards an overview of the user-specific recommendation process is presented. Then the user-specific algorithm is discussed and a heavily commented version of the original code of the user-specific recommendation algorithm by Samer (2017) is presented.

Non-Personalized Approach: The non-personalized recommendation approach is used for guest users. A list of programs is recommended, consisting of the programs with the highest amount of likes in descending order. It is based on the idea of recommending programs that appeal to a broad audience.

User-Specific Approach: In general, the process of the user-specific recommendation approach can be divided into three major steps.

1. Retrieve all relevant likes.
2. Create an array of potential recommendations.
3. Weight and rank the programs of the array of potential recommendations.

The first step will be to retrieve all relevant likes for the recommendation process. Relevant likes are all likes received from users whose user similarity to the active user is greater than zero. For a user similarity between two users to be greater than zero, at least one program must have been liked by both users (see section 3.5.2).

The second step will be to create an array which includes all programs that are potential recommendations. This is done by iterating the likes retrieved in step one and adding each program to the array of potential recommendations which is referred to by a like[16]. Excluded from the array of potential recommendations are all programs which the user receiving the recommendation already likes.

In the third and last step the programs of the array of potential recommendations are weighted. The criterion by which the programs are weighted is aggregated user similarity. Therefore, for each program in the array of

[16]Note that the like entity stores the information which program was liked, as shown in figure 3.5

Program	User	Like Type
Memory	u_1	1
Memory	u_2	3
Memory	u_3	4
Piano	u_2	2
Piano	u_3	1
Compass	u_3	4

Table 3.2.: This table shows exemplary likes of programs. The columns describe which programs were liked, the user who liked the program and the type of like.

	u_1	u_2	u_3
u_1	1	0.5	0.33
u_2		1	0.67
u_3			1

Table 3.3.: This table shows exemplary user similarities between u_1, u_2 and u_3.

potential recommendations the sum of all user similarities between other users who liked the program and the active user are aggregated in order to form weights[17]. The idea behind using user similarities to weight potential recommendations is the assumption that users who shared a similar taste in the past will also share a similar taste in the future. Thus, programs liked by users with a high user similarity to the active user are given a higher weight than programs liked by users who only have a low user similarity to the active user. Then the programs are ranked by total weight in descending order.

Given three users u_1, u_2, u_3 and likes as shown in table 3.2 and user similarities as seen in table 3.3, an exemplary recommendation can be made for u_1 ($u_a = u_1$). The example will be showcased on the basis of the aforementioned three major steps.

1. Retrieve all relevant likes:

[17]Note that in the original code (shown in code 3.1), the weights are calculated while iterating the relevant likes, which corresponds to step two. Here it is described in step three for the sake of simplicity and clarity.

Table 3.3 shows that the user similarities between u_a and other users u_2 and u_3 are greater than zero. Therefore, all likes shown in table 3.2 are relevant for the recommendation process.

2. Create an array of potential recommendations:

In this step the likes of users with a user similarity to the active user greater than zero (formally $sim(u_a, u_x) > 0$) are iterated and programs that u_a has not already liked are added to the array of potential recommendations. Hence the programs *Piano* and *Compass* are added to the array. The program *Memory* is not added to the array since u_a has already liked the program.

3. Rank the programs of the array of potential recommendations:

In order to rank the programs in the array of potential recommendations, the program's weights must be calculated. A weight of a program, denoted by $w_{program}$, is calculated by aggregating all user similarities between the active user and other users who liked the program.

Equation (3.2) shows the weight calculation of the program *Piano* and the result is given in equation (3.3). For the program *Compass* the total weight is shown in equation (3.4).

$$w_{piano} = 0.5 + 0.33 \tag{3.2}$$

$$w_{piano} = 0.83 \tag{3.3}$$

$$w_{compass} = 0.33 \tag{3.4}$$

Given that $w_{piano} > w_{compass}$ and that the array is ranked by weights in descending order, *Piano* is ranked before *Compass* in the final recommendation list.

In code 3.1 the code of the original recommender system, written by Samer (2017), is shown. Multiple comments have been added for the sake of clarity.

```
public function recommendProgramsOfLikeSimilarUsers($user,$flavor)
{
  // Verify that the user is eligible for recommendations
  $min_num_of_likes_required_to_allow_recommendations = 1;
  $all_likes_of_user = $this->program_like_repository->
    findBy(['user_id' => $user->getId()]);
  if (count($all_likes_of_user)
    < $min_num_of_likes_required_to_allow_recommendations)
  {
    return [];
  }

  // Retrieve relevant user similarities
  $user_similarity_relations =
    $this->user_like_similarity_relation_repository
      ->getRelationsOfSimilarUsers($user);
  $similar_user_similarity_mapping = [];
  foreach ($user_similarity_relations as $r)
  {
  $id_of_similar_user = ($r->getFirstUserId() != $user->getId()) ?
    $r->getFirstUserId() : $r->getSecondUserId();
  $similar_user_similarity_mapping[$id_of_similar_user] =
    $r->getSimilarity();
  }
  $ids_of_similar_users = array_keys(
    $similar_user_similarity_mapping);

  // Retrieve relevant likes of similar users
  $excluded_ids_of_liked_programs = array_unique(array_map(
    function ($like){
      return $like->getProgramId();
  }, $all_likes_of_user));
  $differing_likes = $this->program_like_repository->
    getLikesOfUsers(
      $ids_of_similar_users, $user->getId(),
        $excluded_ids_of_liked_programs, $flavor);

  // Weights are computed. Therefore, relevant likes are iterated.
  // Each like adds a weight to the liked program's total weight.
  // The weight added equals the user similarity between the user
  // who liked the program and the user receiving the
  // recommendation.
  $recommendation_weights = [];
  $programs_liked_by_others = [];
  foreach ($differing_likes as $differing_like)
  {
    $key = $differing_like->getProgramId();
    assert(!in_array($key, $excluded_ids_of_liked_programs));
    if (!array_key_exists($key, $recommendation_weights))
    {
      $recommendation_weights[$key] = 0.0;
      $programs_liked_by_others[$key] = $differing_like->
        getProgram();
    }
    $recommendation_weights[$key] +=
      $similar_user_similarity_mapping[
        $differing_like->getUserId()];
  }

  // Sort the recommendations by total weight in descending order.
  // Then convert the sorted recommendations to programs.
  arsort($recommendation_weights);
  return array_map(function ($program_id) use (
    $programs_liked_by_others) {
      return $programs_liked_by_others[$program_id];
  }, array_keys($recommendation_weights));
}
```

Code 3.1: This extract shows the original code of Catrobat's recommender system by Samer (2017).

4. Improvement to Catrobat's Recommender System

First of all, this chapter presents various potential improvements to Catrobat's recommender system. Subsequently, these approaches are discussed and one approach will be chosen, which is then implemented in the practical part of this thesis.

4.1. Potential Improvements

There are various possibilities for improving Catrobat's recommender system. Some possible approaches are described in Samer, 2017, pp. 132–134. In the following different potential improvements are presented.

4.1.1. Prediction Accuracy

One of the most prominent properties of recommender systems in literature is prediction accuracy, as demonstrated, for instance, by the Netflix Prize contest (see section 2.1.2). The prediction accuracy of Catrobat's recommender system can be described by the conversion rate of the *Recommended Programs* section on Catrobat's sharing platform for logged in users. The click-through rate, which describes the number of clicks on recommended programs in relation to the number of visits of Catrobat's sharing platform, is also of interest. In Samer's online experiment, which lasted about two months, he found that within that time frame, the user group which experienced the recommender system as it is at the time of writing this thesis, visited Catrobat's sharing platform 8213 times and clicked 1447 times on recommended programs which results in a click-through rate of approximately 17.62%, as shown in equation (4.1) (Samer, 2017, pp. 118–119).

$$17.62 \approx \frac{1447}{8213} \tag{4.1}$$

In Samer, 2017, the conversion-rate is described by the ratio of how many programs from the section of recommended programs were downloaded by users after they had visited Catrobat's sharing platform. Therefore, the conversion rate can be calculated by dividing the number of conversions by the number of visits to Catrobat's sharing platform. The result is approximately 7.05%, which is shown in equation (4.2).

$$7.05 \approx \frac{579}{8213} \tag{4.2}$$

A higher accuracy therefore means higher values of the click-through rate and especially of the conversion rate, as a high click-through rate indicates that the recommendations are interesting at first sight. On the other hand, a high conversion rate indicates that users find the recommended programs still interesting after visiting the details page of the programs.

As described in section 3.5.3, during the recommendation process weights are assigned to potential recommendations, based on the user similarities between similar users and the active user. These weights are then used to rank the recommendations. One possible approach to achieve a higher prediction accuracy might be to decrease weights of programs that have been liked by users who only have a small number of co-rated items with the active user, similar to the description in Herlocker, Konstan, Borchers, et al., 1999 (see section 2.3.2).

4.1.2. Model-Based Algorithms

Another approach to improving Catrobat's recommender system is the transition from memory-based collaborative filtering to model-based collaborative filtering. This would have the advantage that the high sparsity of the rating matrix would be reduced which would enable a better scalability of Catrobat's recommender system. Given that there are more than 88,000 programs and over 80,000 registered users on Catrobat's community platform at the time of writing this thesis, and in light of the fact that both numbers are expected to grow in the future, scalability is an important property of Catrobat's recommender system. Especially the computation of user similarities requires a large amount of resources due to the fact that Catrobat has nearly 4,500 users who are eligible for recommendations that form a total of approximately 10 million similarity calculations for user pairs[1]. Also, this number rises exponentially. If the number of users who are eligible for recommendation doubles, for instance, the number of required similarity calculation rises fourfold. K-means clustering, as presented in section 2.3.3, could be used to group similar users into clusters in order to decrease the amount of data which is needed to compute recommendations.

4.1.3. Ramp-Up Problem

A common weakness of collaborative filtering recommender systems is the *ramp-up problem* (Burke, 2002), also referred to as *cold-start problem*, which concerns the new-user and new-item problem as described in section 2.3.2 and section 2.3.2 respectively. Basically, the ramp-up problem can be approached in two ways. One way is to change the existing collaborative filtering recommendation algorithm. *PIP* (Ahn, 2008), for instance, utilizes an alternative approach to classic similarity measurements, such as the Pearson coefficient (equation (2.3)) and cosine similarity (equation (2.11)), in order to reduce the effect of the cold-start problem. An overview of different methods of approaching the new-user problem in collaborative filtering is portrayed in Son, 2014. The second way is to combine Catrobat's current collaborative filtering recommender system with a knowledge-based recommender system, as described in section 2.3.6. However, given the nature of the items in Catrobat's recommender system, a knowledge-based recommender system might be difficult to implement. Even though there is much data about programs available, such as bricks, variables, scenes, graphics

[1]In total 4,500 users form approximately 20 million user pairs, however, about half of them are duplicates.

and sounds, these data can have many different meanings depending on their usage.

4.1.4. Long-Tail Problem

The long-tail problem within the context of recommender systems describes the problem that only a small portion of the set of all items is highly popular, while most items receive only limited attention. It refers to the recommender system property item-space coverage (see section 2.1.4) and within the context of Catrobat's sharing platform to the property diversity. Due to the fact that Catrobat's current recommender system creates the weights of recommendations by aggregating the user similarities between the active user and similar users (see section 3.5.3), the more likes a program has received, the higher the program will be ranked in the list of recommended programs. Of course, niche programs, which have not received many likes, can also be ranked highly in the list of recommended items, if the user similarities to users who liked the niche programs are very high. However, it is likely that programs with many likes have a higher total weight than niche programs, even in the case that the average ratings of the highly popular programs are much lower. This leads to the effect that the recommended programs are highly similar to the most downloaded and most viewed programs of Catrobat's sharing platform. Figure 4.2 shows recommended programs of a logged in user next to the most downloaded programs. It can be seen that 10 out of 18 recommendations are identical to programs found in the section of most downloaded programs. The sharing platform is most frequently visited by mobile devices, where usually only 6 programs are shown per section, as a consequence of the smaller screen size. In this case 3 out of 6 recommendations can be found in the section of most downloaded programs. Given the limited space that is available to present programs, especially on mobile devices, showing such a high number of duplicate programs can be considered as wasteful.

Generally speaking, programs that are featured in the lists of most downloaded and most viewed programs are more likely to be downloaded and are thus more likely to receive likes from users than other programs. This results in a higher likelihood for these programs to be recommended. In

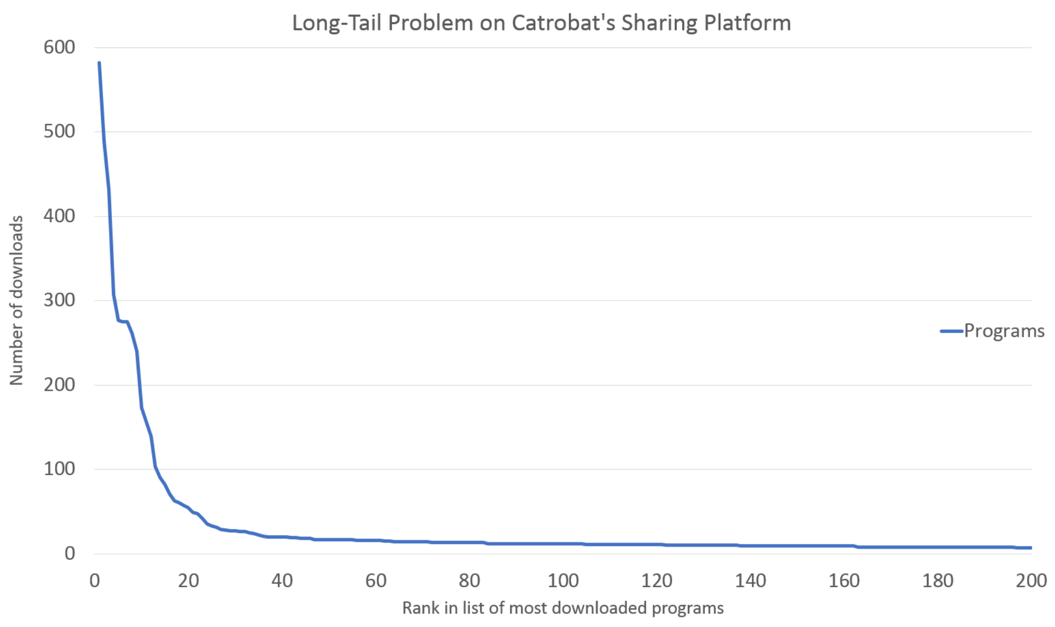


Figure 4.1.: This figure shows a line chart illustrating the long-tail problem on Catrobat's sharing platform. On the y-axis the number of downloads of the 200 most downloaded programs (that are placed on the x-axis) over a period of one month (March 15, 2019 to April 15, 2019) is displayed.

turn, programs that are recommended more often are more likely to get downloaded by users, which forms a positive feedback loop. Therefore, there are some programs which dominate all of the following three sections on Catrobat's sharing platform: *Most Downloaded*, *Most Viewed* and *Recommended Programs*. This causes Catrobat's recommender system to have a low aggregated diversity.

Figure 4.1 illustrates the long-tail problem on the home page of Catrobat's sharing platform. The figure shows the 200 most downloaded programs with their respective number of downloads over a period of one month. It demonstrates that a few programs were downloaded very often, however, the number of downloads of the majority of the programs was low.

A possible approach to reducing the effect of the long-tail problem on Catrobat's recommender system and to increasing aggregated diversity would be to also consider the average weights of programs in the recommendation process besides the total weights. Alternatively, to promote less popular programs, they could be given an advantage in the recommendation process.

4.2. Discussion and Decision

In the following the different potential improvements presented in section 4.1 are discussed and one of the improvements is chosen for the practical part of this thesis.

All of the potential improvements would most probably be valuable for Catrobat's recommender system and it is difficult to predict which improvement would have the most positive impact. However, given the fact that Catrobat has recently upgraded its hardware and has no performance or scalability issues at the time of writing this thesis and that performance and scalability issues are not expected in the next two years, adopting a model-based recommender approach might not have sufficient immediate impact in comparison with other potential improvements. Furthermore, Catrobat's current recommender system calculates user similarities offline, which causes the recommender system to be more scalable than user-based collaborative filtering approaches usually are.

Constructing a hybrid recommender system in order to reduce the effect of the ramp-up problem might be another potential improvement to Catrobat's recommender system. The most effective addition to the current collaborative filtering system is most likely a knowledge-based recommender system, which could support the collaborative filtering system by generating recommendations for users who have not provided enough feedback for reasonable predictions by the current collaborative filtering system. However, knowledge-based approaches require detailed information about item attributes. On Catrobat's community platform users are not required to provide detailed information about uploaded programs. If users were required to do so, it could lower the users' willingness to upload their programs to the sharing platform. On the other hand, a lot of data about uploaded programs is available, such as statistics indicating which bricks and files are used. This data is difficult to interpret though, as the same bricks and media

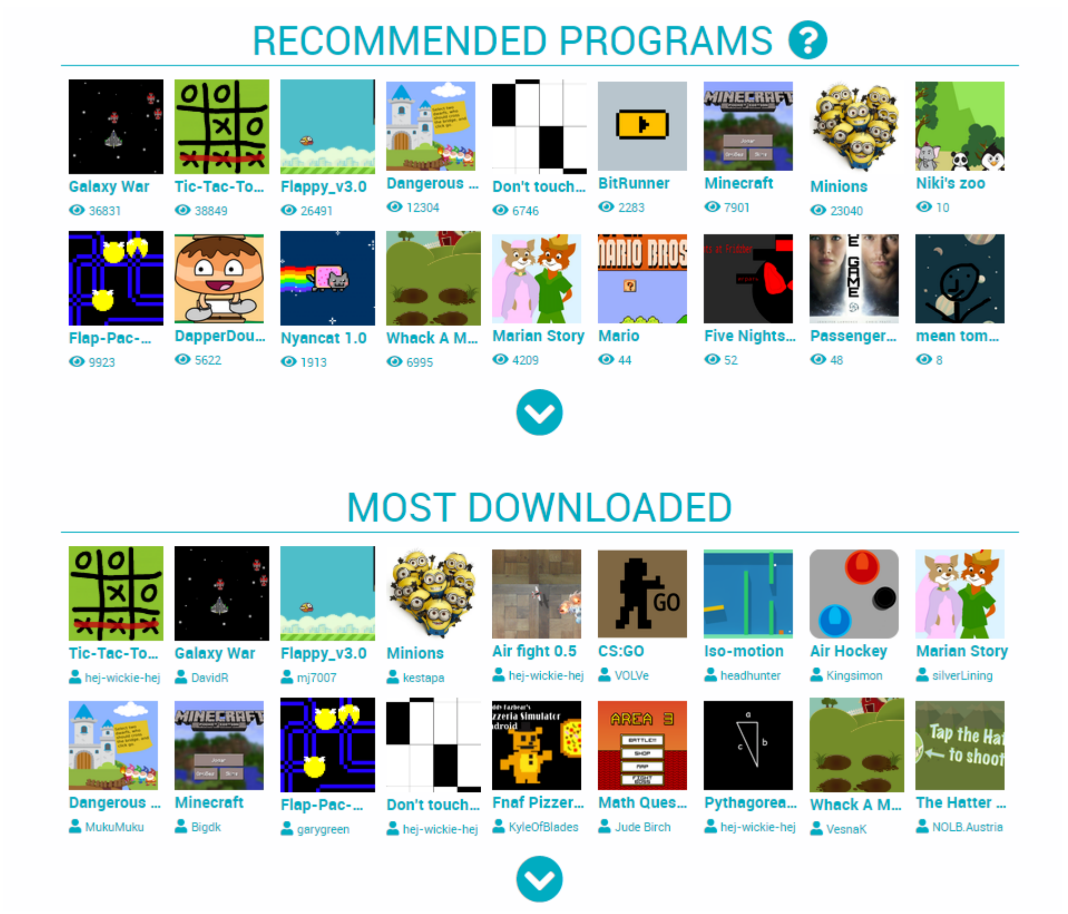


Figure 4.2.: This figure shows recommended programs and the most downloaded programs on Catrobat's sharing platform for a logged in user. The recommendations have been generated for a user who has liked five programs. The website (International Catrobat Association, 2019b) was accessed on a 21″ display.

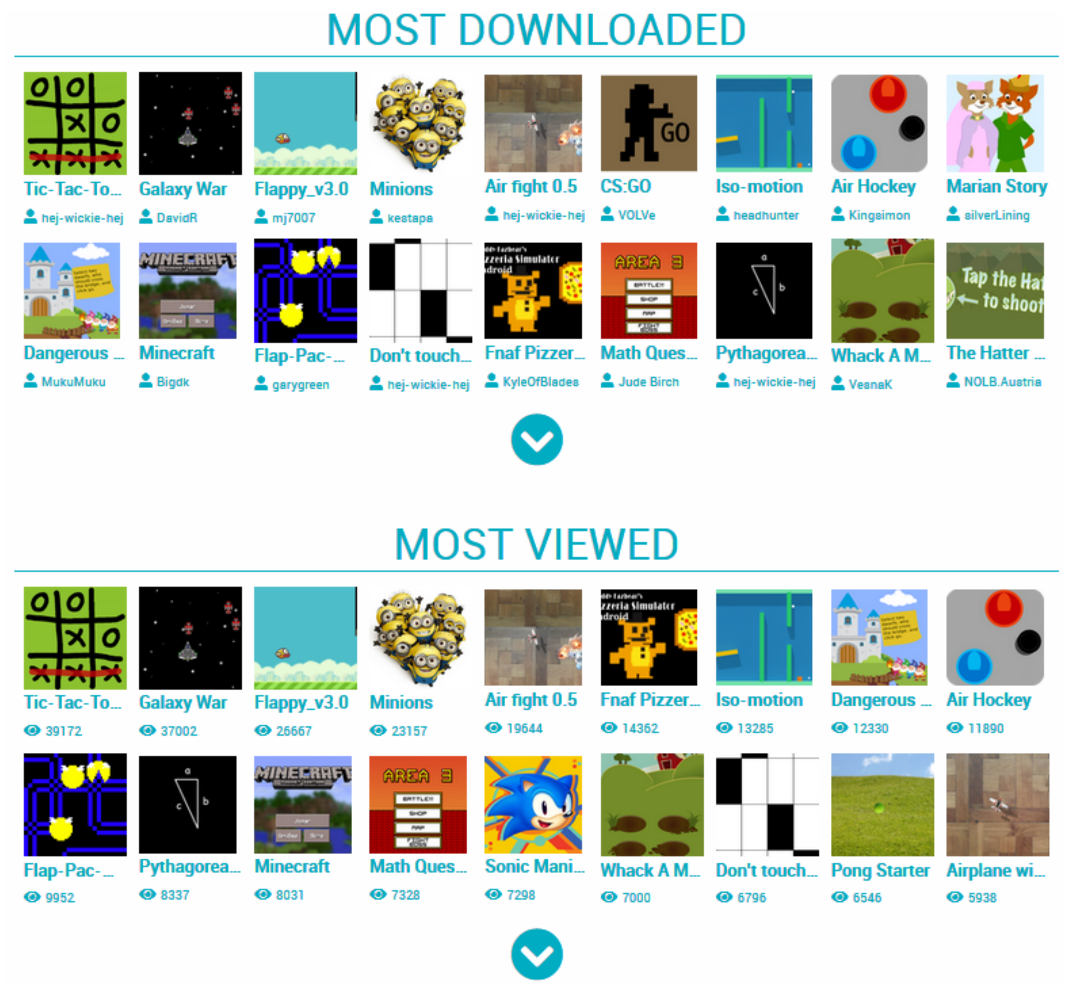
files can be used in many different ways. Especially bricks are difficult to interpret, as they represent small, commonly used code snippets, which are by nature versatile in use. An advanced machine learning approach might be able to automatically retrieve data which is suitable for a knowledge-based recommender system, however, adopting such an approach would be beyond the scope of this thesis.

Maximizing prediction accuracy is a common goal in recommender systems, for example as shown by the Netflix prize competition (see section 2.1.2). Thus, it can be assumed that Catrobat's recommender system would improve with higher accuracy levels. However, in McNee, Riedl, and Konstan, 2006 it is argued that a high accuracy alone is not sufficient for high quality recommendations. Moreover, increasing accuracy often comes with certain trade-offs, such as lowering a recommender system's diversity (G. Adomavicius and Y. Kwon, 2012). The exact trade-offs depend on the domain in which the recommender system operates. Reusing the example introduced in section 2.1.4, given a movie recommender system and an active user who rated many action movies starring the actor Liam Neeson highly, recommending action movies starring Liam Neeson, which the active user has not viewed yet, will most probably ensure a high prediction accuracy, as it is highly likely that the active user will like the recommendations. However, there are some shortcomings in such a recommendation. Firstly the chance that the user would have found the recommended items without the help of the recommendations is high. Secondly, the set of items is one-sided and features low levels of diversity, novelty and serendipity. On the other hand, the trade-offs produced by increasing the prediction accuracy of Catrobat's recommender system are difficult to estimate. One possibility is that more accurate predictions would further decrease aggregated diversity, based on the assumption that high quality programs are highly popular programs and that high quality programs are generally accurate predictions. On the contrary, another possibility is that further increasing the accuracy of Catrobat's recommender system could in fact increase aggregated diversity, as, for example, the recommender system might improve its ability to identify accurate recommendations independent of item popularity. These effects are, however, only assumptions and the exact trade-offs cannot be known for sure. Therefore, increasing the accuracy of Catrobat's recommender system, while trying to minimize unwanted trade-offs could be beneficial to the system, although it entails a considerable risk as the trade-offs are difficult to foresee.

The long-tail problem seems to have a high impact on Catrobat's sharing platform. The whole platform revolves around the idea that users can share their programs for others to see, to learn from them and to come up with new ideas themselves. However, Catrobat's current recommender system favors programs which are highly popular. Therefore, new and less popular programs that have only received a small number of likes rarely make it to the top of a recommendation list. This raises the question if duplicates in the sections of most downloaded, most viewed and in recommended programs should be avoided. Overlaps between the sections of most downloaded and most viewed programs are to be expected and are shown in figure 4.3, where it can be seen that 15 out of 18 presented programs are the same. For mobile users, who usually only see 6 programs on the smaller sized screens, 5 out of 6 programs are identical. Figure 4.4 shows the section of recommended programs and most downloaded programs for a guest user, whereby 10 out of 18 presented programs are identical in the case of a desktop guest user and 3 out of 6 programs for a mobile guest user. Given that all three sections, *Most Viewed*, *Most Downloaded* and *Recommended Programs*, are on the same page, it is extremely questionable whether showing the same program in all three sections or in two of the same sections is desirable. Especially the process of recommending the same programs that can be found in one of the other two sections should be avoided, as users can find these items simply by scrolling down anyway. Instead the recommendation field could be used to show programs which users might not find on their own. This would likely have various positive impacts on Catrobat's recommender system:

- a higher aggregated diversity of Catrobat's recommender system
- a higher overall serendipity of recommendations
- a higher overall novelty of recommendations.

An expected drawback is that the prediction accuracy would most likely decrease, given that less feedback is available for less popular items. However, G. Adomavicius and Y. Kwon (2012) showed that the trade-off between accuracy and aggregated diversity can be controlled and that small losses in accuracy can greatly improve aggregated diversity.

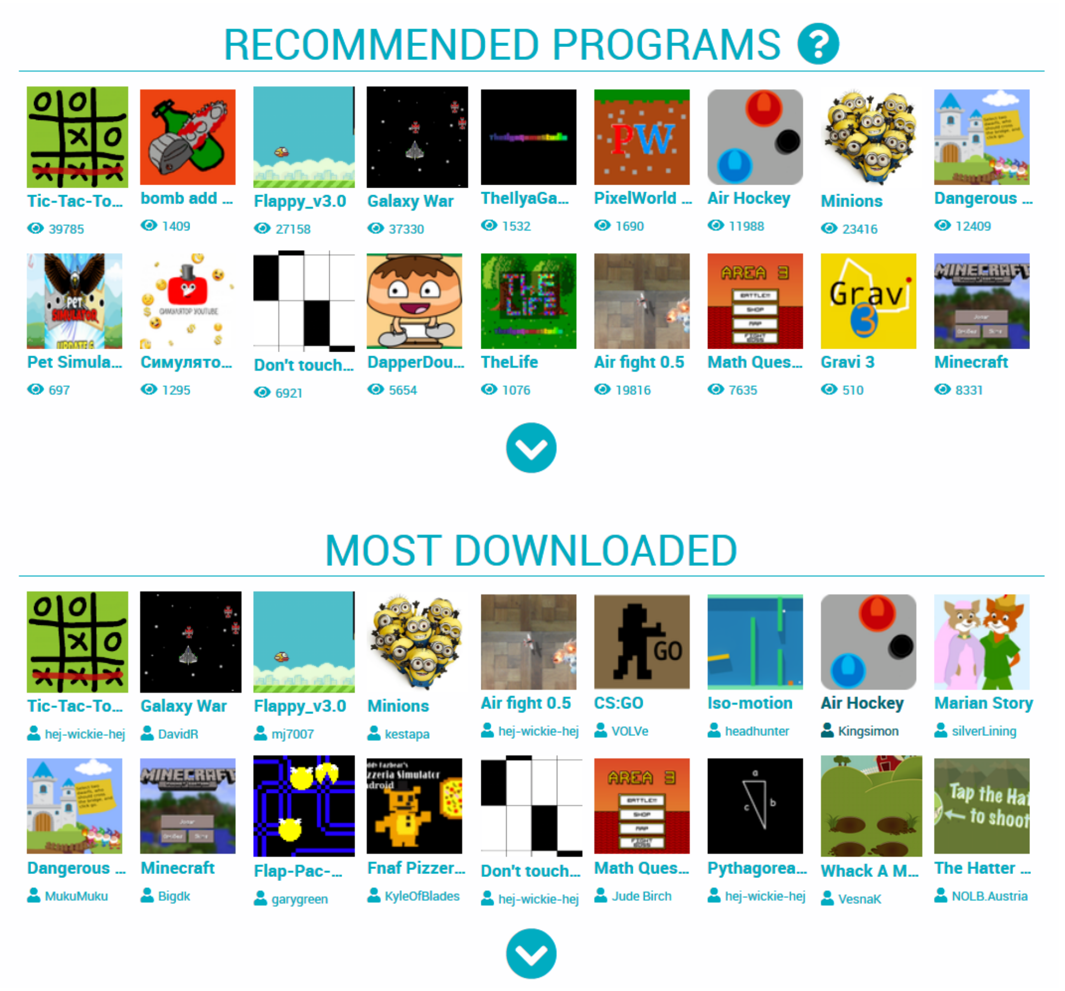

MOST DOWNLOADED

Tic-Tac-To... hej-wickie-hej
Galaxy War DavidR
Flappy_v3.0 mj7007
Minions kestapa
Air fight 0.5 hej-wickie-hej
CS:GO VOLVe
Iso-motion headhunter
Air Hockey Kingsimon
Marian Story silverLining
Dangerous ... MukuMuku
Minecraft Bigdk
Flap-Pac-... garygreen
Don't touch... hej-wickie-hej
Fnaf Pizzer... KyleOfBlades
Math Ques... Jude Birch
Pythagorea... hej-wickie-hej
Whack A M... VesnaK
The Hatter ... NOLB.Austria

MOST VIEWED

Tic-Tac-To... 39172
Galaxy War 37002
Flappy_v3.0 26667
Minions 23157
Air fight 0.5 19644
Fnaf Pizzer... 14362
Iso-motion 13285
Dangerous ... 12330
Air Hockey 11890
Flap-Pac-... 9952
Pythagorea... 8337
Minecraft 8031
Math Ques... 7328
Sonic Mani... 7298
Whack A M... 7000
Don't touch... 6796
Pong Starter 6546
Airplane wi... 5938

Figure 4.3.: This figure shows the sections of most downloaded and most viewed programs of Catrobat's sharing platform. A very high overlap can be seen. The website was accessed on a 21.5" screen on May 1, 2019 (International Catrobat Association, 2019b).



RECOMMENDED PROGRAMS

Tic-Tac-To... 39785
bomb add ... 1409
Flappy_v3.0 27158
Galaxy War 37330
TheIlyaGa... 1532
PixelWorld ... 1690
Air Hockey 11988
Minions 23416
Dangerous ... 12409
Pet Simula... 697
Симулято... 1295
Don't touch... 6921
DapperDou... 5654
TheLife 1076
Air fight 0.5 19816
Math Ques... 7635
Gravi 3 510
Minecraft 8331

MOST DOWNLOADED

Tic-Tac-To... hej-wickie-hej
Galaxy War DavidR
Flappy_v3.0 mj7007
Minions kestapa
Air fight 0.5 hej-wickie-hej
CS:GO VOLVe
Iso-motion headhunter
Air Hockey Kingsimon
Marian Story silverLining
Dangerous ... MukuMuku
Minecraft Bigdk
Flap-Pac-... garygreen
Fnaf Pizzer... KyleOfBlades
Don't touch... hej-wickie-hej
Math Ques... Jude Birch
Pythagorea... hej-wickie-hej
Whack A M... VesnaK
The Hatter ... NOLB.Austria

Figure 4.4.: This figure shows the sections of most downloaded and recommended programs of Catrobat's sharing platform for a guest user. A considerable overlap can be seen. The website was accessed on a 21.5" screen on May 20, 2019 (International Catrobat Association, 2019b).

In light of the reasons discussed in this section, reducing the effect of the long-tail problem and thus increasing the recommender system's aggregated diversity and its levels of novelty and serendipity seems to be the most efficient improvement to Catrobat's recommender system at the time of writing this thesis. With this goal in mind, the remaining practical part of this thesis is focused on changing the recommender system's algorithm.

4.3. Related Work

This section presents related work whose objective was to increase aggregated diversity in recommender systems.

One approach described in G. Adomavicius and Y. Kwon, 2012 is to re-rank an existing recommendation list taking into consideration average item ratings and item popularity. In Karakaya and Aytekin, 2017 two different approaches are proposed, firstly a graph-based re-ranking approach of an existing recommendation list and secondly the model generation phase of a model-based collaborative filtering approach is changed in order to increase aggregated diversity. *Usage context-based collaborative filtering* is introduced in Niemann and Wolpers, 2013, which combines parts of user-based and item-based collaborative filtering and considers co-occurrences of item pairs. In Patil and Wagh, 2014 collaborative filtering is used to produce a recommendation list. Furthermore, a content-based approach is adopted in order to re-rank the recommendation list based on item attributes. Javari and Jalili (2014) propose a hybrid model with adjustable levels of diversity and precision.

5. Implementation

This chapter presents the implementations for the practical part of this thesis. The changes made to the user-specific part of Catrobat's recommender system are outlined in section 5.1 and section 5.2; section 5.3 describes the changes made to the non-personalized part. Furthermore, *user groups* were implemented in order to avoid biases in the online experiments. They are presented in section 5.4.

5.1. User-Specific Re-Ranking Approach I

The first re-ranking approach offers a straightforward solution to the problem of low aggregated diversity. It is a modification of Samer's user-based collaborative filtering approach (Samer, 2017) that is used by Catrobat's recommender system at the time of writing this thesis (the original code is shown in section 3.5.3).

The basic idea is to decrease weights of highly popular programs in the recommendation list. These programs are not completely eliminated from the recommendation list, but recommended less frequently and they are ranked lower. This is achieved by identifying highly popular programs after the original algorithm created the recommendation list and by multiplying their total weights by a factor smaller than 1. After the weight has decreased, the recommendation list is sorted by total weights in descending order. Therefore, this approach can be seen as a re-ranking approach which is applied on top of Catrobat's current recommender system. The number of programs affected by the weight decrease is 75, which has been chosen arbitrarily.

Therefore, a metric is necessary to determine the popularity of a program. Potential metrics are program views, program downloads and program likes. Furthermore, there are the possibilities of measuring by raw numbers (for instance 5,000 downloads) or by a program's rank in a top list (for instance 23rd most downloaded program). In general, downloads seem to indicate a stronger interest in programs than views. It appears that a program that has been downloaded very often relative to its number of views is more popular than a program that has been viewed very often relative to its number of downloads. The notion that a program, which has received a large number of likes compared to its views and downloads, is a popular one, seems plausible as well. However, using the number of likes of a program as an indicator of popularity might result in the decrease of weights of programs that have received many likes, even though their number of views and downloads is relatively low, which is generally not desirable. Furthermore, a metric based on a program's rank in a top list is used for the sake of simplicity. Thus, program downloads measured by rank in the list of most downloaded programs seems to be the most suitable way to determine a program's popularity within this context.

A mathematical function is necessary in order to calculate the weight decrease based on a program's rank in the list of most downloaded programs. The formula, which describes the multiplier dm which decreases the weight of a program p with rank $n \in \{0, 1, ..., 74\}$ in the list of most downloaded programs can be seen in equation (5.1).

$$dm(p_n) = \cos_{deg}(75 - n) \tag{5.1}$$

The decreased weight dw of a program p is calculated by multiplying the original total weight of a program $weight_p$ by the decrease-multiplier dm, as shown in equation (5.2).

$$dw_p = weight_p * dm(p_n) \tag{5.2}$$

The formula has been chosen because it consistently reduces the weight from rank to rank, fast in the beginning, then more slowly. For instance, the difference in weight decrease between the first most popular and the

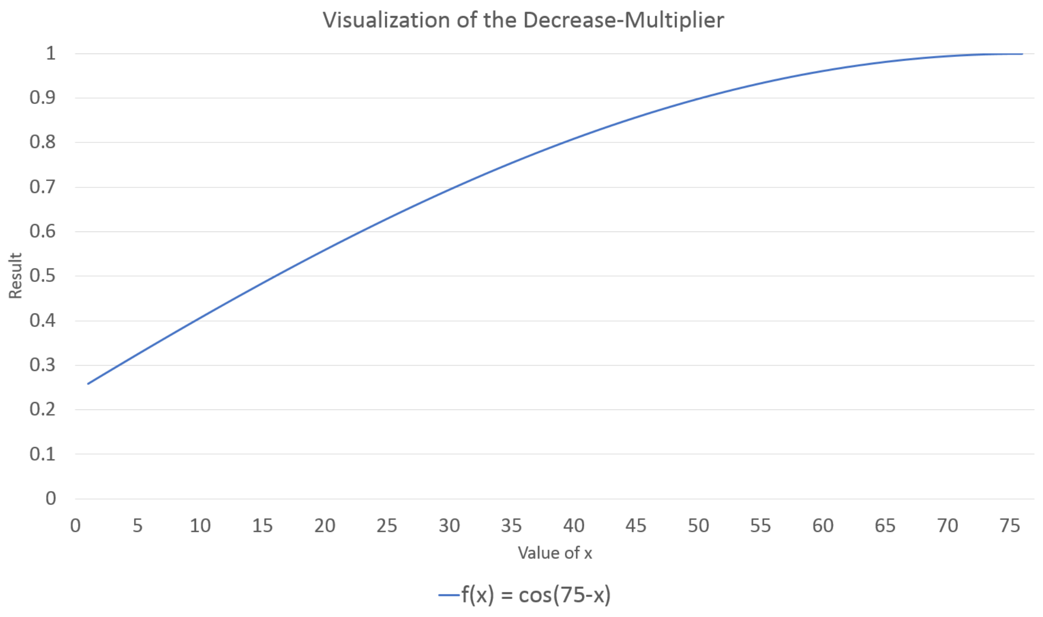


Figure 5.1.: This figure shows a line chart illustrating results (y-axis) of the function that can be seen at the bottom of the figure for values between 0 and 75 (x-axis), which is used to calculate the decrease-multiplier.

second most popular program is about 1.68%, while the difference in weight decrease between the 35th and the 36th most popular program is about 1.11%. Thus, weights of programs at the top of the list of most downloaded programs are reduced more drastically, for instance the weight of the most downloaded program is reduced by around 74%. The lower the rank of a program in the list of most downloaded programs, the lower the decrease of the program's weight in the recommendation list. In figure 5.1 the mathematical function $f(x) = \cos_{deg}(75 - x)$ is shown for values $x \in \{0, 1, ..., 74\}$. It can be seen that the gradient rises quickly for small values of x and then flattens for higher values of x.

The performance of the modified algorithm is almost identical to the original version. For one recommendation made to a user the worst case is 75 additional calculations, in the event that all of the 75 most downloaded programs are part of the recommendation list. The performance of these additional calculations has been tested by measuring the computing times

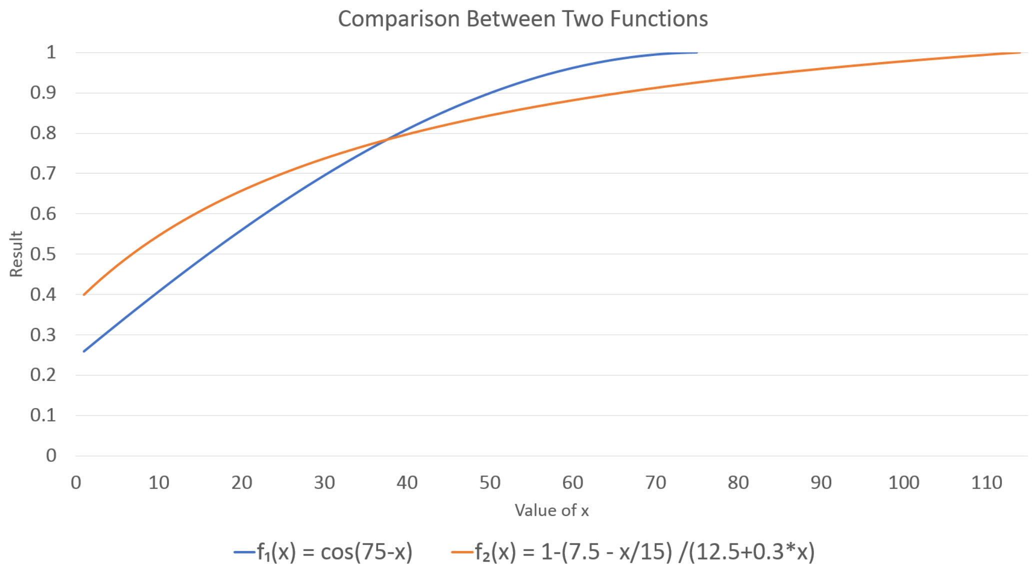


Figure 5.2.: This figure shows a line chart illustrating results (y-axis) of the functions that can be seen at the bottom of the figure for values between 0 and 112 (x-axis), which can be used to calculate the decrease-multiplier.

of the 75 additional calculations one million times and by calculating the average computing time on a low-end Ubuntu virtual machine. The result is less than 0.0002 seconds of additional computation per recommendation in the worst case and therefore negligible.

Figure 5.2 shows the chosen function in comparison with another function which was carefully considered. Both functions feature a suitable gradient, however, the cosine function was preferred since it starts with lower values and the other function's gradient flattens too early.

Other notable functions that have been considered are shown in figure 5.3. Each of these functions could be used as decrease-multiplier, however, the strong rise in the beginning and the flattening towards the end of the shown cosine function suit the requirements best.

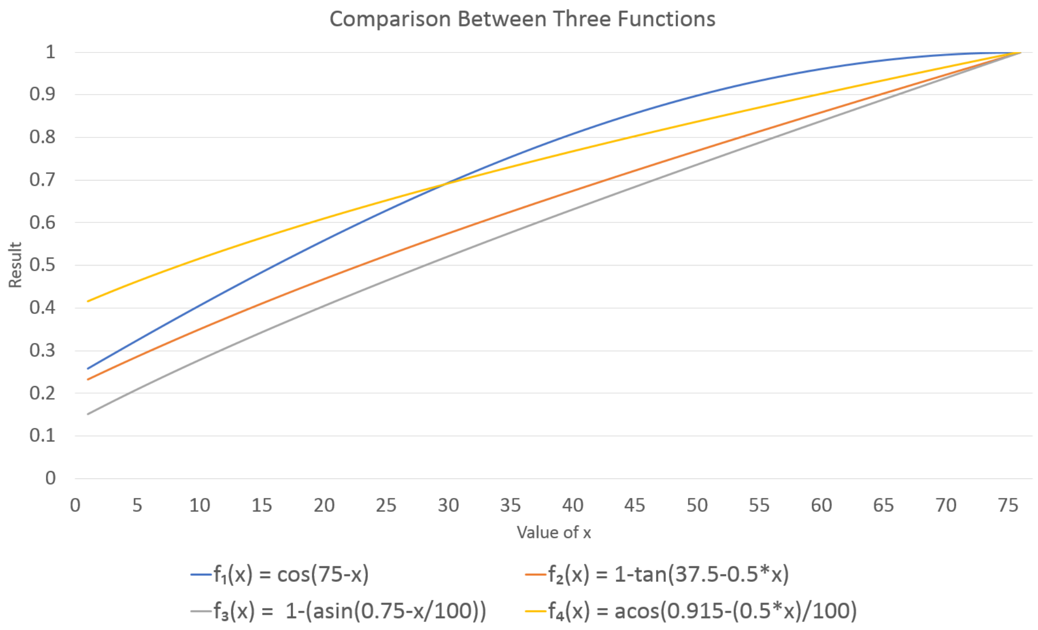


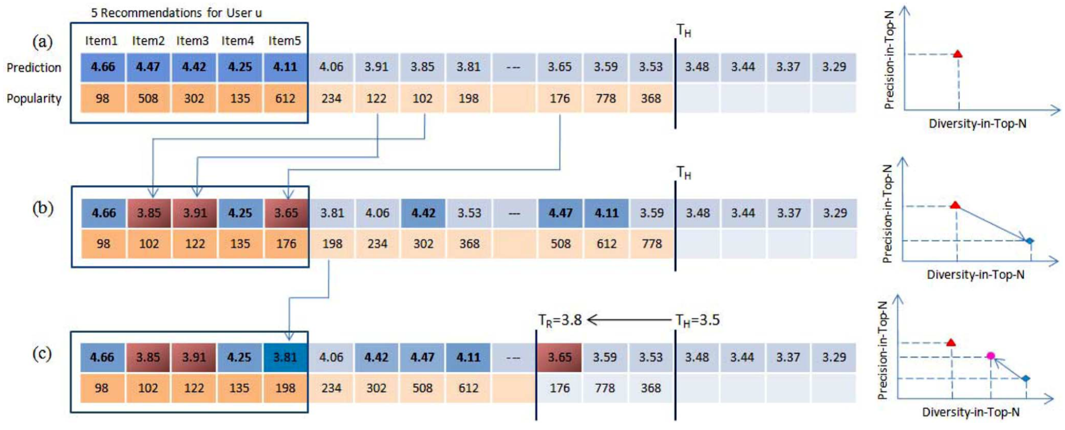
Figure 5.3.: This figure shows a line chart illustrating results (y-axis) of the functions that can be seen at the bottom of the figure for values between 0 and 75 (x-axis), which can be used to calculate the decrease-multiplier.

5.2. User-Specific Re-Ranking Approach II

The second implemented algorithm is based on an approach presented in G. Adomavicius and Y. Kwon, 2012. In this article, the authors propose a re-ranking technique, which aims to improve the aggregated diversity of a recommender system while maintaining acceptable levels of accuracy. Their re-ranking method makes it possible to control the trade-off between accuracy and aggregated diversity. This section begins by outlining the re-ranking approach by G. Adomavicius and Y. Kwon (2012). Then the implementation of a similar re-ranking approach for Catrobat's recommender system based on the approach in G. Adomavicius and Y. Kwon, 2012 is presented.

Basically, Adomavicius and Kwon achieve their objectives by considering item popularity in addition to predicted ratings in the recommendation process. A rating threshold T_H is defined which categorizes a predicted item rating as *high rating* if it is above T_H and as *non-high rating* if it is below T_H. For instance, for a rating scale with possible ratings between 1 and 5, a rating threshold can be defined as $T_H = 3.5$. In a traditional recommendation list, which is usually sorted by predicted ratings in descending order, items which have a predicted rating above T_H can be re-ranked on the basis of an alternative ranking approach which yields a higher aggregated diversity than the standard ranking approach. One such approach would be to rank items with predicted ratings above T_H by popularity in ascending order. Items with predicted ratings below T_H are ranked by the standard ranking approach. All items with predicted ratings above T_H are placed before items with predicted ratings below T_H in the final recommendation list.

However, Adomavicius and Kwon discovered that the resulting decrease in accuracy might be too high, especially for commercial systems. Therefore, the authors introduced a second threshold $T_R \in [T_H, T_{max}]$, where T_{max} denotes the highest possible rating on the rating scale (for instance $T_{max} = 5$ for possible ratings between 1 and 5), in order to allow users to choose the level of recommendation accuracy themselves by adjusting T_R. Items with a predicted rating above T_R are ranked by the alternative ranking approach and items with a predicted rating below T_R are ranked by the standard ranking approach. In the final recommendation list, all items with predicted



(a) Recommending top-*N* highly predicted items for user *u*, according to standard ranking approach
(b) Recommending top-*N* items, according to some other ranking approach for better diversity
(c) Confining re-ranked recommendations to the items above new ranking threshold T_R (e.g., ≥ 3.8) for better accuracy

Figure 5.4.: This figure illustrates the re-ranking approach by G. Adomavicius and Y. Kwon, 2012. The picture has been adopted from G. Adomavicius and Y. Kwon, 2012, ©2012 IEEE, with kind permission of the authors.

ratings above T_R are placed before all items with predicted ratings below T_R. Thus, tweaking T_R to be higher, increases the accuracy of the set of items above the threshold, as items with predicted ratings lower than T_R are placed after all programs with predicted ratings above T_R in the final recommendation list. Tweaking T_R to be lower increases the aggregated diversity, as the set of items, which can be ranked by the alternative ranking approach, is larger. The described re-ranking approach by G. Adomavicius and Y. Kwon (2012) is illustrated in figure 5.4.

Furthermore, Adomavicius and Kwon define and compare various alternative ranking approaches. Exemplary results on the MovieLens data set are shown in table 5.1 (G. Adomavicius and Y. Kwon, 2012). The authors define precision by the popular *precision-in-top-N* metric, which is shown in equation (5.3), where $L_N(u) = i_1, i_2, ..., i_N$ denotes a list of recommended items and $R^*(u, i_k) \geq T_H$ denotes a predicted rating above T_H from a user $u \in U$ for an item $i \in I$ for all $k \in \{1, 2, ..., N\}$. A correct prediction is denoted by $correct(L_N(u)) = \{i \in L_N(u) \,|\, R(u,i) \geq T_H\}$, where $R(u,i)$ describes an actual user rating.

Precision Loss	Diversity Gain in Numbers	Multiple in %
-0.1	+800	3.078
-0.05	+594	2.543
-0.025	+411	2.068
-0.01	+270	1.701
-0.005	+189	1.491
-0.001	+93	1.242
Standard: 0.892	385	1.000

Table 5.1.: This table shows the increase in diversity in numbers and percentage compared to precision loss, as found in G. Adomavicius and Y. Kwon, 2012. Thereby the standard ranking approach was an item-based collaborative filtering approach on the MovieLens data set, where the 5 highest ranked items were recommended. This table is a reproduction of a similar table in G. Adomavicius and Y. Kwon, 2012, © 2012 IEEE, in order to showcase the trade-off between aggregated diversity and precision when items above T_R are re-ranked by popularity in ascending order. More data about the trade-off can be found in G. Adomavicius and Y. Kwon, 2012.

$$precision\text{-}in\text{-}top\text{-}N = \frac{\sum_{u \in U} |correct(L_N(u))|}{\sum_{u \in U} |L_N(u)|} \tag{5.3}$$

Given that the approach by G. Adomavicius and Y. Kwon (2012) can be implemented on top of an existing recommendation approach and the results shown in table 5.1 yield a high improvement in aggregated diversity for a reasonable trade-off in precision, an algorithm based on the re-ranking approach in G. Adomavicius and Y. Kwon, 2012 has been implemented. However, the approach had to be adjusted in order to be compatible with Catrobat's recommender system, since positive-only, unary feedback is used by Catrobat's recommender system, instead of a rating scale, which is used in G. Adomavicius and Y. Kwon, 2012. This necessitated choosing an alternative metric by which the thresholds could be defined. As described in section 3.5.3, Catrobat's recommender system uses the similarities of other users $u \in U$, where U denotes the set of all users, to the active user u_a as basis to weight programs $p \in P$, where P denotes the set of all programs. This is based on the assumption that generally a user's taste remain the same over time. Thus, it is presumed that users who shared a similar taste in the past will also share a similar taste in the future. A high user similarity between u_a and another user u_1 therefore indicates a high probability that u_a will like programs that u_1 likes and u_a has not seen yet. On the other hand, a low user similarity between u_a and another user u_2 indicates that u_a and u_2 are similar to a certain extend, but the probability that u_a will like programs that u_2 likes and u_a has not seen yet is much lower than the probability, that u_a will like programs that u_1 liked and u_a has not seen yet. Thus, the average user similarity $\overline{sim(u_a, p)}$ between u_a and all other users who like a program p can be used alternatively to total weights to describe how likely it is that u_a will like p. Formally, the average user similarity for a program is shown in equation (5.4), where M denotes the set of users who like p and $C(M)$ denotes the number of users in M. The formula which is used to calculate the average user similarity for u_a is shown in equation (5.5), where N denotes the set of similar users of u_a, formally $\{n \in N \,|\, sim(u_a, u_n) > 0\}$.

$$\overline{sim(u_a, p)} = \frac{\sum_{m \in M} sim(u_a, u_m)}{C(M)} \tag{5.4}$$

$$\overline{sim(u_a, u_n)} = \frac{\sum_{n \in N} sim(u_a, u_n)}{C(N)} \tag{5.5}$$

However, average user similarities of programs can be highly different depending on a user's similarities to other users. Therefore, the thresholds T_H and T_R are not defined as static values, as in G. Adomavicius and Y. Kwon, 2012, but as a multiple of the average user similarity of a user to other users ($\overline{sim(u_a, u_n)}$). For the implementation of the re-ranking approach two thresholds T_H and T_R have been defined. They are shown in equation (5.6) and equation (5.7) respectively.

$$T_H = \overline{sim(u_a, u_n)} * 1.25 \tag{5.6}$$

$$T_R = \overline{sim(u_a, u_n)} * 1.5 \tag{5.7}$$

The values of the multipliers have been chosen arbitrarily, since testing multiple different values is beyond the scope of this thesis. However, if the presented modification to the algorithm proves fruitful in the online experiment, further experiments regarding the trade-off between accuracy and aggregated diversity that might result from different values for the multipliers are advised.

Furthermore, in contrast to the re-ranking method in G. Adomavicius and Y. Kwon, 2012, T_H and T_R are both used at the same time. Thus, all programs with $\overline{sim(u_a, p)} \geq T_R$ are ranked ahead of all programs with $T_H \leq \overline{sim(u_a, p)} < T_R$, which are in turn ranked ahead of all programs with $\overline{sim(u_a, p)} < T_H$. This decision has been made because Catrobat's section for recommended programs on the sharing platform can be expanded until the end of the recommendation list is reached. Therefore, it is likely that some users will expand the section to the point where programs are shown with $\overline{sim(u_a, p)} < T_R$. Following the re-ranking approach in G. Adomavicius and Y. Kwon, 2012, items with a predicted rating below T_R are ranked by the standard ranking approach. However, given that highly ranked Catrobat programs ranked by the standard ranking approach are likely to be hugely popular and therefore easy to find for a user in the sections of most downloaded and most viewed programs, it is assumed that programs with $T_H \leq \overline{sim(u_a, p)} < T_R$, which are ranked by the alternative ranking approach, are more valuable recommendations for a user than programs with $\overline{sim(u_a, p)} < T_R$, ranked by the standard ranking approach. Thus, all programs with $T_H \leq \overline{sim(u_a, p)} < T_R$ are ranked by the alternative ranking approach and placed before all programs with $\overline{sim(u_a, p)} < T_H$ in the recommendation list, which are ranked by the standard ranking approach. Although ranking more programs by the alternative ranking approach is likely to further decrease accuracy, it is believed that the expected additional increase in aggregated diversity is worth the trade-off.

The alternative ranking approach used is to rank programs by their number of likes in ascending order, since the lower a program's number of likes are, the less popular the program is. It has also been considered to use the program's rank in the list of most downloaded programs, similarly as in the implementation of the first re-ranking approach (see section 5.1). However, it is assumed that a program's rank in the list of most downloaded programs is less likely to change than a program's number of likes (and thus its rank in the list of most liked programs), given that one additional like reflects a considerable raise in popularity, while, for instance, ten additional downloads do not indicate growing popularity. This means that if a program captures the users' attention, the increase in attention is not reflected as quickly by the popularity rank of the program in the list of most downloaded programs as it is in the program's number of likes.

The re-ranking process of the original recommendation list can be divided into three major steps:

1. The first step is to assign programs to different lists depending on their average user similarity $\overline{sim(u_a, p)}$.
   If $\overline{sim(u_a, p)} \geq T_R$, the program is placed in the list L_{top}.
   Else, if $\overline{sim(u_a, p)} \geq T_H$, the program is placed in the list L_{high}.
   Else, if $\overline{sim(u_a, p)} < T_H$, the program remains in the original recommendation list $L_{standard}$.
2. The next step is to re-rank the lists. L_{top} and L_{high} are ranked by the alternative ranking approach, which is by the programs' number of likes in ascending order. $L_{standard}$ is ranked by the programs' total weights in descending order, which is the standard ranking approach.
3. In the third step the three recommendation lists are merged into one final recommendation list. L_{top} is put before L_{high}, which is placed before $L_{standard}$.

It is noted that in the special case that the number of programs in L_{top} is less than 12, L_{top} is not placed before L_{high} in the final recommendation list, but both lists are merged and then re-ranked by the alternative ranking approach. The reason for this is that if the number of programs in L_{top} is very small, the chance that the set consist only of popular programs or a large number of popular programs is relatively high, which would lead to recommendations with popular programs relatively near the top of the recommendation list. With the underlying goal in mind to increase the recommender system's aggregated diversity, this is generally not desirable. The number 12 has been chosen due to the fact that it is double the number of programs that are usually shown in the section of recommended programs on Catrobat's sharing platform when accessed by a mobile device.

Therefore, up to half of the programs in L_{top} can be popular ones and the unexpanded recommendation list still consists only of unpopular programs, when accessed by a mobile device.

After the re-ranking process is completed, the final recommendation list usually starts with a large number of unpopular programs. Despite their unpopularity, it can be assumed that the programs will be received well by the users, given the high average user similarities of the programs. Performance-wise the additional computations are negligible. The code of the re-ranking approach can be found in appendix B (code B.2).

5.3. Non-Personalized Re-Ranking Approach

In the following the implementation of the re-ranking of non-personalized recommendations is presented. Generally methods of re-ranking non-personalized recommendation lists are limited, in the light of the fact that the lists must appeal to the majority of users. As described in section 3.5.3, Catrobat's recommender system recommends a list of programs with the highest number of likes in descending order to guest users, since it is assumed that these programs appeal to the majority of users. As shown in figure 4.4, many programs in the section of recommended programs are identical to programs shown in the section of most downloaded programs. Given that this overlap is likely to be one of the reasons for the low aggregated diversity of Catrobat's recommender system, the focus of the re-ranking approach presented in this section is on reducing the number of overlapping programs between the section of most downloaded and recommended programs on Catrobat's sharing platform.

The re-ranking approach for non-personalized recommendations is based on the implementation presented in section 5.1. Basically, the list of programs with the highest number of likes is used as the recommendation list, but popular programs are ranked lower than their number of likes would indicate. A program's popularity is defined by the program's rank in the list of most downloaded programs, equally to the definition provided in section 5.1. During the ranking process, the number of likes of programs within the set of the 45 most downloaded programs is modified to make sure that the recommendation list does not start with programs from the top of the list of most downloaded programs. To do so, the number of likes of a popular program is multiplied by a number smaller than 1. The multiplier is referred to as *decrease-multiplier*.

The formula for calculating the decrease-multiplier is similar to the one used in the first re-ranking approach of user-specific programs (presented in equation (5.1)). It was slightly modified in order to ensure that no programs from the top of the list of most downloaded programs appear at the beginning of the recommendation list. Therefore, the weights of programs which are in the beginning of the list of most downloaded programs had to be further reduced and the number of programs for which the weights are reduced is decreased from 75 to 45. Equation (5.8) shows the formula, whereby dm denotes the decrease-multiplier of a program p with rank $n \in \{0, 1, ..., 44\}$ in the list of most downloaded programs.

$$dm(p_n) = \cos_{deg}(70 - n * 1.5)^2 \tag{5.8}$$

Figure 5.5 shows a graph that performs a comparison between the formula presented in equation (5.8) and the formula presented in equation (5.1). There, possible values for the decrease-multipliers are illustrated.

The decreased number of likes dnl of a program p is calculated by multiplying the original number of likes of a program nl_p by dm, as shown in equation (5.9).

$$dnl_p = nl_p * dm(p_n) \tag{5.9}$$

Therefore, at first the number of likes of all programs within the set of the 45 most downloaded programs is decreased. In the next step the recommendation list is ranked by the number of likes in descending order. Thus, programs within the set of the 45 most downloaded programs are not excluded from the recommendation list, but ranked significantly lower than before the implemented changes. However, the number of likes on the details page of a program is not affected by the decrease during the re-ranking process.

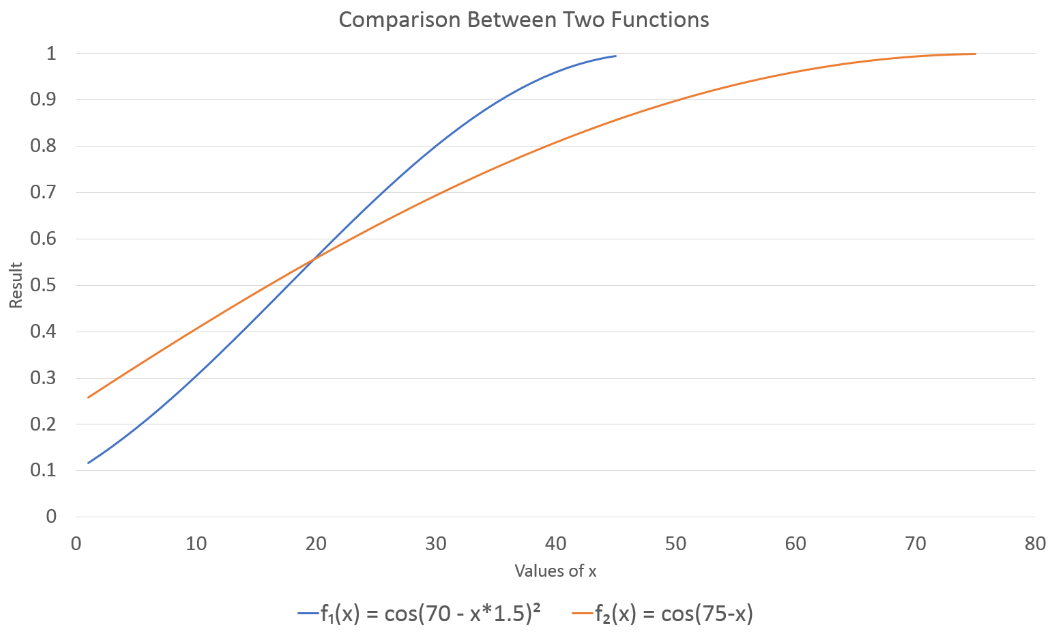


Figure 5.5.: This figure shows a line chart illustrating results (y-axis) of the functions that can be seen at the bottom of the figure for values between 0 and 75 (x-axis), which is used to calculate the decrease-multiplier.

Figure 5.6 shows the sections of recommended and most downloaded programs for guest users on Catrobat's sharing platform after the changes have come into effect. It can be seen that only 5 out of 18 programs can be found in both sections, in comparison with 9 out of 18 programs without the re-ranking. The programs that can be found in both sections are ranked on the places 6th, 10th, 11th, 12th and 18th in the recommendation list after the re-ranking. As shown in figure 4.3, before the re-ranking of the recommendation list, programs that could be found in both sections were ranked 1st, 3rd, 4th, 7th, 8th, 9th, 12th, 15th, 16th, 18th in the recommendation list. For mobile users, who usually only see the first 6 programs on the smaller sized screens, only 1 program can be found in both sections, in comparison with 3 out of 6 programs without the re-ranking.

The impact of the additional computations on the re-ranking process is negligible. The code of the re-ranking approach is shown in appendix B (code B.3).

5.4. User Groups

When conducting an online experiment, in which multiple versions of a system are tested simultaneously, a method of assigning users to different test groups is needed. There are multiple ways in which users can be assigned to groups, for instance users can be assigned to groups based on their location, or the set of users can be divided into equal parts and each part can then be assigned to a group. However, most of these methods may result in biased groups. In order to eliminate possible biases, it is common practice to randomly assign users to groups. Therefore, an entity called *User Test Group* has been implemented, which allows to randomly assign users to groups and to look up to which group a user belongs. As mentioned in section 3.4, an entity represents a class which is mapped by doctrine.

The representation of the entity in the database can be seen in figure 5.7. It features three fields: *user_id*, which is used to identify a user by a unique identifier, *group_number*, which describes to which group a user is assigned and *created_at*, which indicates the time at which the user was assigned to

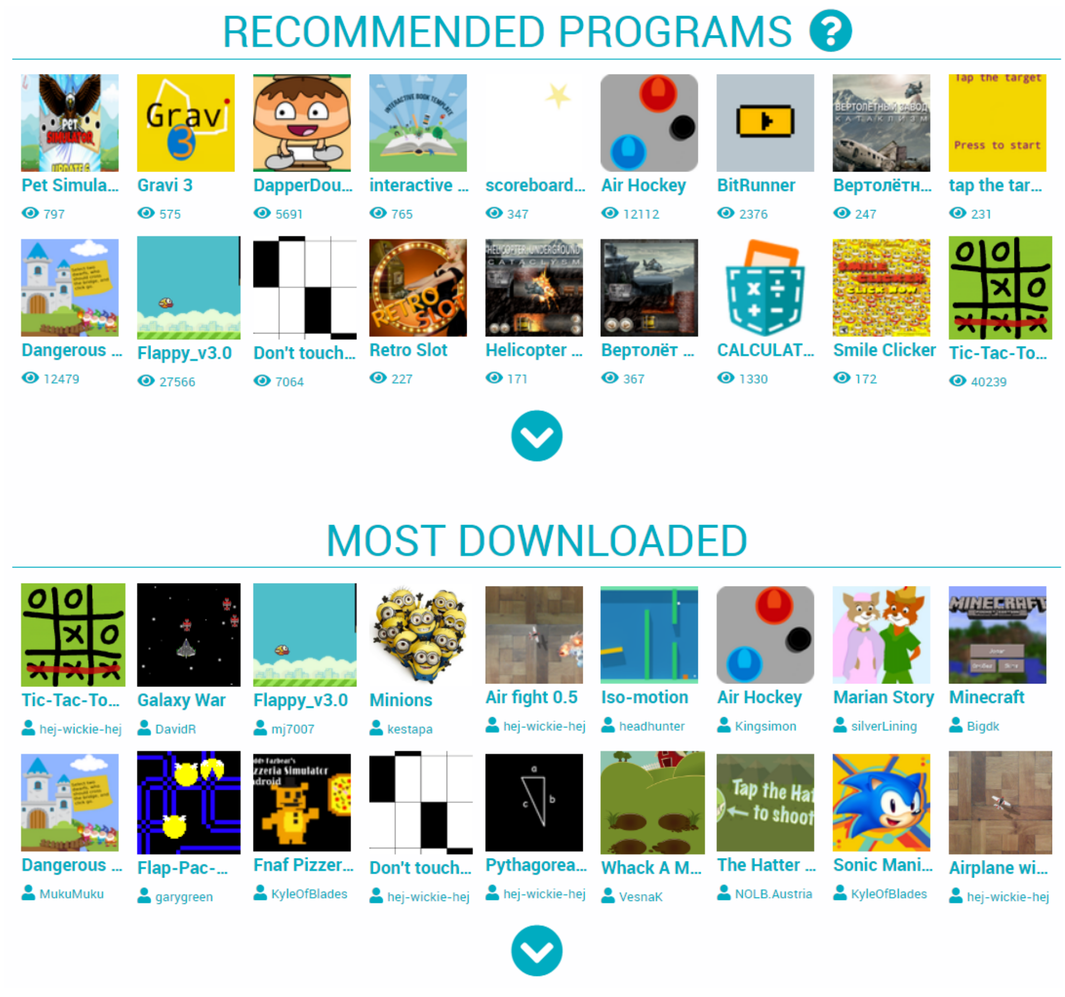
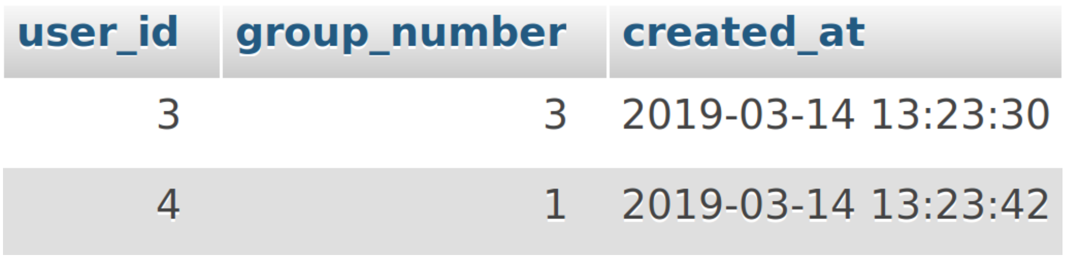


Figure 5.6.: This figure shows the sections of recommended programs and most downloaded programs for guest users on Catrobat's sharing platform, whereby the recommendation list was re-ranked by the approach described in section 5.3. The website (International Catrobat Association, 2019b) was accessed on a 21″ display.



user_id	group_number	created_at
3	3	2019-03-14 13:23:30
4	1	2019-03-14 13:23:42

Figure 5.7.: This figure shows the entity *User Test Group* in the database.

the group. The implementation of the entity can be found in appendix C (code C.1).

The decision to create a new entity rather than simply creating a new field for the existing user entity is based on two reasons. First and foremost, Catrobat is likely to carry out more online experiments in the future, which might require different designs as those used in the online experiments conducted in this thesis, for example, additional fields might be needed. Thus, one reason for the creation of a new entity is that it is easy to modify according to specific needs. The second reason why a new entity was created is to decouple the test groups from the user entity in order to reduce the risk of breaking the existing and future code.

6. Evaluation and Results

This chapter discusses the evaluations of the re-ranking approaches presented in chapter 5. The implemented changes to Catrobat's recommender system have been evaluated in two online experiments on Catrobat's sharing platform, with the aim of answering the following research questions.

Research Question 1: *Can the aggregated diversity of Catrobat's recommender system be significantly improved by one of the implemented approaches while maintaining an acceptable level of accuracy?*

The recommender system's aggregated diversity is measured by the number of different programs that have been downloaded from the section of recommended programs. A significant improvement of aggregated diversity is achieved by an improvement of at least 25%. An acceptable level of accuracy loss is specified as 15% or less, measured by the conversion rate of recommended programs. The conversion rate is defined as the ratio of how many of the viewed programs have been downloaded (see section 2.1.1).

Research Question 2: *How do the implemented approaches impact the novelty and serendipity levels of Catrobat's recommender system?*

Novelty is defined as the rate of how many recommended programs had been unknown to a user until they were viewed. Thus, a novel program is a program that has not been viewed by a user before. On the other hand, serendipity is defined as the rate at which programs that had been unknown to a user before (novel programs) were downloaded. It can be viewed as a conversion rate for novel programs only.

In order to answer the research question, two online experiments have been conducted. The first online experiment is outlined in section 6.1. It is an evaluation of the two user-specific re-ranking approaches, which are described in section 5.1 and section 5.2. Section 6.2 presents the second online experiment, which was conducted in order to evaluate the non-personalized re-ranking approach that was implemented, as outlined in section 5.3.

6.1. User-Specific Re-Ranking Approaches

This section starts with an overview of the test scenario of the first online experiment. Afterwards the observed results are presented.

6.1.1. Test Scenario I

In the first online experiment three different versions of user-specific recommendations are compared on the home page of Catrobat's sharing platform. The first version, which is referred to as *Version A*, is the original version of the recommender system by Samer (2017), as described in section 3.5.3. Version A is used as a baseline for the online experiment. *Version B* refers to the straightforward re-ranking approach presented in section 5.1. *Version C* refers to the re-ranking approach described in section 5.2, which considers average user similarities. All three versions of the system (*A, B, C*) only differ as to how the recommendations are ranked in the section of recommended programs on Catrobat's sharing platform. During the testing period, users were randomly assigned to one of three groups (*1, 2, 3*). Depending on the user group that a user was assigned to, the user was presented a different version of the system. Users were assigned to a group the first time they logged in during the testing period and the relevant information was stored in the database with the help of the implemented user groups, which are described in section 5.4. Which group a user was assigned to was determined randomly. Once a user had been assigned to a group, the user remained in the same group until the end of the online experiment. The online experiment lasted 34 days, from April 30, 2019 to June 2, 2019. Table 6.1 shows the presented versions of the system for each user group and the number of users in each group at the end of the testing period. In total, 491 users took part in the online experiment.

User Group	Presented Version	Number of Users
Group 1	Version A	156
Group 2	Version B	158
Group 3	Version C	177

Table 6.1.: This table shows which versions of the user-specific recommender system were presented to which user groups.

Criterion	Group 1	Group 2	Group 3
Views	1	5	4
Downloads	0	0	2
Conversion Rate	0	0	50
Different Views	1	4	3
Different Downloads	0	0	2

Table 6.2.: This table shows the results of the first online experiment. The results include the number of views and downloads for each group from the section of recommended programs on Catrobat's sharing platform, as well as the conversion rates. Furthermore, the number of different programs that have been viewed or downloaded during the first online experiment is shown for all three groups.

6.1.2. Results

Table 6.2 shows the results of the online experiment. A visual representation of the observed number of views and downloads for each group can be seen in figure 6.1. Within this context, a *view* corresponded to the action of clicking on a program in the section of recommended programs on Catrobat's sharing platform and thus viewing the details page of a program. A *download* was registered when a viewed program was downloaded. In contrast to the description in Samer, 2017, it was not possible to collect data about the number of visits of Catrobat's sharing platform. This was due to the fact that users were randomly assigned to test groups, which are not represented in Google Analytics[1].

The results shown in table 6.2 and figure 6.1 indicate that recommendations from Version C performed best, given that 100% of the downloads during the online experiment were performed by members of Group 3. In terms

[1]Google LLC, 2019b.

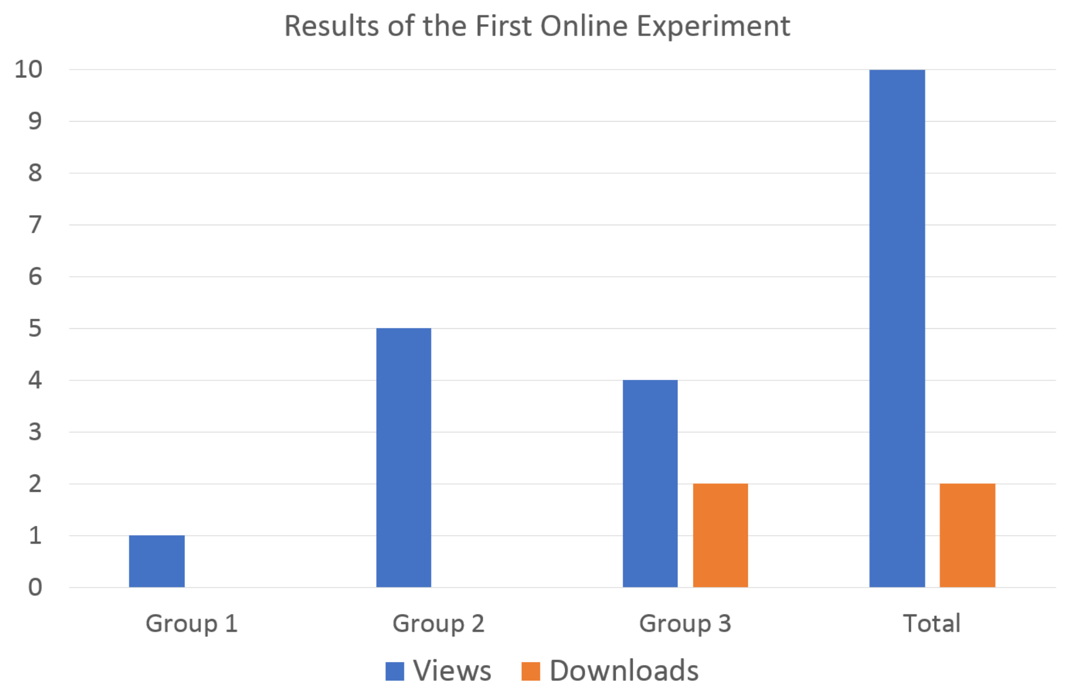


Figure 6.1.: This figure is a visualization of the results of the first online experiment. It shows the number of views and downloads of recommended programs of the different groups of the first online experiment.

of views, Version B and Version C both performed similarly. The baseline version of the system, Version A, performed poorly in terms of views and downloads. Table 6.2 also shows the number of different programs that have been viewed or downloaded. It can be seen that Version B and Version C achieved acceptable levels of diversity. However, insufficient data was generated by any group in order to be able to draw reasonable conclusions about the different versions of the recommender system in terms of diversity, accuracy, novelty and serendipity. Although in percentage the number of views generated by Group 2 and 3 and the number of downloads generated by Group 3 are much higher than the numbers generated by Group 1, the raw numbers only show a slight difference. In fact, one user in either group would have been able to strongly influence the results, as it is common for a user to view or download multiple programs over a span of 34 days.

Overall, an unexpectedly small number of views and downloads of user-specific recommendations took place during the testing period. In particular, taking into consideration that 491 users took part in the online experiment, it seems unlikely that only a total of 10 views and 2 downloads of user-specific recommendations took place within a time frame of 34 days. By comparison, non-personalized recommendations, which are shown to guest users and logged in users who are not eligible for user-specific recommendations, received much more attention with 2329 views and 1234 downloads during the testing period, as shown in table 6.3. Moreover, table 6.3 shows that the conversion rate of non-personalized recommendations compared to user-specific recommendations is much higher. From these numbers 62 views and 18 downloads were created by logged in users who were not eligible for recommendation, either because they have not liked at least one program or the user similarities have not been updated since they liked their first program. Although these numbers are higher than the total views and downloads of user-specific recommendations, they are still low compared to the number of views and downloads of non-personalized recommendations during the online experiment. The low download rate of logged in users persists if all downloads from the home page of Catrobat's sharing platform during the testing period are considered, with 179 downloads from logged in users, compared to 6572 downloads from guest users.

	Views	Downloads	Conversion Rate (in %)
User-Specific	10	2	20
Non-Personalized	2329	1234	52.98

Table 6.3.: This table shows a comparison between the activity of guest users and logged in users during the testing period of the online experiment. Shown are the number of views and downloads of programs of the section of recommended programs on Catrobat's sharing platform, as well as the corresponding approximate conversion rates in percentage.

6.2. Non-Personalized Re-Ranking Approach

This section describes the test scenario of the second online experiment and subsequently presents the results.

6.2.1. Test Scenario II

The second online experiment was conducted in order to compare two different versions of non-personalized recommendations on Catrobat's sharing platform. Version A is represented by the original version by Samer (2017) and Version B is represented by the re-ranking approach, which has been implemented as a part of this thesis. The implementation of Version B is described in section 5.3. Unfortunately, at the time of writing this thesis, it was not possible to present Version A to one part of the guest users and Version B to another part simultaneously. However, given that Version A is the baseline that Version B is compared to and due to the fact that no changes have been introduced to Catrobat's sharing platform recently, historical data was used for the evaluation of Version A. The online experiment for Version B was performed over a period of approximately 10 days, from June 4, 2019 to June 13, 2019. The evaluation of Version A was carried out within an equal time frame, namely from May 25, 2019 to June 3, 2019.

6.2.2. Results

This section presents and discusses the results obtained from the second online experiment. At first, the accuracy of the results is examined. Afterwards, the aggregated diversity of the two versions of the system, Version A and Version B, is reviewed.

Table 6.4 shows the number of views and downloads, as well as the approximate conversion rates in percentage of the section of recommended programs on Catrobat's sharing platform during the respective testing periods. Both versions of the system performed very similar regarding the number of views and downloads. Surprisingly, no decrease in the conversion rate of Version B was found. In fact, with 56.27% the conversion rate of Version B is even slightly higher than the conversion rate of Version A, amounting to 54.99%. Thus, the assumption seems to be reasonable that there is no significant difference regarding the conversion rates of Version A and Version B of Catrobat's recommender system. In order to provide further evidence for this assumption, a chi-square goodness of fit test based on the number of downloads was used, taking into consideration the conversion rates of Version A and Version B. The test considered the expected number of downloads of Version B and the observed downloads of Version B. The expected number of downloads of Version B was calculated by applying the conversion rate of Version A to the number of views of Version B. To do so, the error rate is defined as $\alpha = 0.05$. The calculation of the p-value shows that the probability that the observed differences in the number of downloads are due to chance is approximately 69.2%. This indicates that there is no statistically significant difference between the conversion rates of Version A and Version B. Thus, the conversion rate lies within the predefined margin of up to -15% of the first research question.

However, the number of views and downloads can also be examined taking into account the total number of views and downloads from all sections of the home page of Catrobat's sharing platform, namely: *Featured*, *Newest Programs*, *Recommended Programs*, *Most Downloaded*, *Most Viewed* and *Random Programs*. In doing so, further information about the users' interest in the section of recommended programs compared to the other sections of the home page of Catrobat's sharing platform can be gained. Table 6.5 shows

	Views	Downloads	Conversion Rate (in %)
Version A	551	303	54.99
Version B	526	296	56.27

Table 6.4.: This table shows a comparison between the two versions (A, B) of the recommender system during the testing periods. It specifies the views and downloads of the section of recommended programs on Catrobat's sharing platform, as well as the approximate conversion rates in percentage.

	Total Views	Total Downloads	% of Views from Rec.	% of Downloads from Rec.
Version A	6996	1651	7.88	18.35
Version B	8851	1831	5.94	16.17

Table 6.5.: This table shows a comparison between the total number of views and downloads of the home page of Catrobat's sharing platform of Version A and Version B during the testing periods. It specifies the total number of views and downloads of the section of recommended programs on Catrobat's sharing platform, as well as the percentage of total views and downloads which were recorded in the section of recommended programs.

that the views and downloads of recommended programs of Version B account for approximately 2% less of the total views and downloads during the online experiment than the respective number of Version A. This demonstrates that the rate at which programs from the section of recommended programs were viewed is approximately 24.6% lower in Version B than in Version A. In order to test whether the described lower rate of views of recommended programs of Version B was only found at random, a chi-square goodness of fit test was conducted with $\alpha = 0.05$. The test revealed that the probability that the results were found only by chance is lower than 0.1%. This indicates that users showed overall less interest in the section of recommended programs of Version B compared to Version A, even though the highly similar conversion rates of both versions of the system indicate that the recommendations are equally accurate. However, it is noted that the overall conversion rates of the home page of Catrobat's sharing platform of 23.6% for Version A and 20.69% for Version B are significantly lower than the conversion rates of the sections of recommended programs of 54.99% for Version A and 56.27% for Version B.

Criterion	Version A	Version B	Change (in %)
Number of Different Viewed Recommendations	56	69	+29.07
Number of Different Downloaded Recommendations	43	62	+47.6
Views of Top 6 Most Downloaded Programs	181	21	-87.85
Downloads of Top 6 Most Downloaded Programs	103	10	-90.06
Views of Top 18 Most Downloaded Programs	241	86	-62.62
Downloads of Top 18 Most Downloaded Programs	146	43	-69.85
Programs Only Viewed from the Section of Recommended Programs	30	40	+39.67
Programs Only Downloaded from the Section of Recommended Programs	16	35	+123.92

Table 6.6.: This table compares various criteria for the evaluation of the second online experiment between Version A and Version B. The change in percentage has been calculated taking into consideration the difference in total views and downloads of recommendations between Version A and Version B.

Moreover, the data indicates that the changes to Catrobat's non-personalized recommender system increased the aggregated diversity of the recommendations. This is reflected in the test results shown in table 6.6. The columns *Number of Differently Viewed Recommendations* and *Number of Differently Downloaded Recommendations* specify the number of different programs which have been viewed or downloaded at least once during the testing periods. Both numbers are higher for Version B and the difference is statistically significant with a confidence level of over 99% according to a chi-square goodness of fit test with $\alpha = 0.05$.

The main motivation for implementing the re-ranking of non-personalized recommendations was to reduce the overlap between programs in the sections of most downloaded programs and the section of recommended programs. Therefore, the number of views and downloads of the top 6 and top 18 most downloaded programs[2] in the section of recommended programs during the testing periods was observed. As illustrated in table 6.6, these numbers differ greatly for both system versions. In Version A 181 programs of the set of the top 6 most downloaded programs were viewed within the section of recommended programs and 103 programs were downloaded. This accounts for roughly a third of the total number of viewed and downloaded recommendations of Version A during the testing period. On the other hand, the observed number of views and downloads of the top 6 most downloaded programs within the section of recommended programs is significantly lower in Version B, with 21 views and only 10 downloads. In order to verify whether the values were only found by chance, a chi-square goodness of fit test was conducted with $\alpha = 0.05$. The test revealed a confidence level of over 99%. As shown in table 6.6, the difference persists if the number of observed most downloaded programs increases from 6 to 18. However, as expected, the difference between the two system versions has become smaller for the top 18 most downloaded programs, although the difference is still statistically significant with a confidence level of over 99,99% due to a chi-square goodness of fit test with $\alpha = 0.05$.

Furthermore, it has been found that the number of programs that have been viewed or downloaded only within the section of recommended programs and in no other section of the home page of Catrobat's sharing platform is significantly higher in Version B compared to Version A. As shown in table 6.6 the corresponding number of viewed programs increased by approximately 39.67% and the corresponding number of downloaded programs by approximately 123.92% in Version B if compared to Version A. In order to verify the statistical significance of the data, a chi-square goodness of fit test based on the number of programs viewed and the number of programs downloaded only in the section of recommended programs has been conducted with $\alpha = 0.05$. The test indicated a confidence level of over 99.99%. A further discussion of the results obtained from the second online experiment with regard to the research question is presented

[2]The numbers 6 and 18 have been chosen according to the corresponding number of programs that can usually be seen on the mobile and desktop version of Catrobat's sharing platform.

in section 6.3.

6.3. Discussion

The following is a discussion of the results of the two online experiments as regards the research questions presented in the beginning of chapter 6. It starts with the results of the first online experiment, outlined in section 6.1.2. Afterwards the results observed in the second online experiment are discussed, which are presented in section 6.2.2.

6.3.1. Results of Online Experiment I

Unfortunately, the data derived from the first online experiment was not sufficient to draw reasonable conclusions on any of both research questions. This is due to the fact that the number of viewed and downloaded programs of the section of recommended programs by logged in users was unexpectedly low during the testing period. Therefore, instead of discussing the results of the first online experiment with regard to the research questions, possible explanations for why the number of views and downloads found was so low are discussed.

One possible explanation why the number of views and downloads from the section of recommended programs during the first online experiment was that low is that users simply disliked the recommendations and preferred to download programs from other sections of the sharing platform. However, this explanation seems unlikely, given that there was a large overlap between non-personalized recommendations and user-specific recommendations of the system Version A. This overlap was based on the fact that non-personalized recommendations were ranked by the total number of likes of programs in descending order, while the ranking approach of user-specific recommendations favored programs with a high number of likes. Thus, both, the non-personalized recommendations and the user-specific recommendations of Version A, recommended similar programs. In light of this fact, the explanatory approach that users simply disliked the user-specific recommendations seems unlikely.

Generally speaking, the assumption that most users do not log into their user account when browsing the sharing platform for programs to download seems plausible based on the results found in the first online experiment. This can be due to multiple reasons, which are presented in the following.

- In all likelihood, most of the time users visit the sharing platform to search for new programs to download. In this scenario it is not necessary (or optional) for a user to log into a user account.
- Although users need to log into their user accounts in order to upload their programs, users do not remain logged in when they visit the sharing platform afterwards.
- It might be that there is not sufficient incentive for users to log into their user account when browsing the sharing platform. Currently offered incentives are the ability to comment on a program, to like a program and to receive notifications when an uploaded program receives a comment or a like from another user.

However, further studies about user behavior are needed in order to find evidence for or against the aforementioned assumptions, or to identify different factors that might explain the low number of views and downloads of user-specific recommendations.

6.3.2. Results of Online Experiment II

In the second online experiment enough data could collected to issue statistically significant statements about the two research questions. The first research question is:

Research Question 1: *Can the aggregated diversity of Catrobat's recommender system be significantly improved by one of the implemented approaches while maintaining an acceptable level of accuracy?*

Chapter 6 starts with the statement that aggregated diversity is measured by the number of different downloads of recommended programs. A significant improvement in aggregated diversity is defined by an increase of at least

25%. Accuracy is defined as conversion rate, whereby an acceptable loss is specified as a decrease of up to 15%.

The online experiment showed that the performance of Version B was similar to Version A in terms of accuracy. With approximately 55% and 56% for Version A and B respectively, both system versions have highly similar conversion rates. This strongly indicates that Version B maintained an acceptable level of accuracy.

Although the conversion rates were highly similar in both versions, the data in table 6.5 indicates that the number of views of the section of recommended programs, compared to the total views of programs on Catrobat's home page, was lower in Version B than in Version A. This might be due to the reason that in some instances users preferred to view more popular programs from other sections instead of the less popular programs in the section of recommended programs. As their number of views and downloads suggests, these highly popular programs seem to attract much user attention. However, from the perspective of Catrobat it seems to be of little importance from which section of the home page a program is viewed or downloaded. As long as the overall activity and the respective conversion rates remain stable or increase, it is believed that a modest redistribution of views is unproblematic.

Furthermore, the observed data suggests that the changes made to the non-personalized part of Catrobat's recommender system increased its aggregated diversity by a considerable amount. As shown in table 6.6, the number of different programs downloaded from the section of recommended programs during the online experiment was 47.6% higher in Version B than in Version A. This exceeds the defined minimum increase in aggregated diversity of 25% of the research question by a solid amount. The increase might be due to various factors. Table 6.6 shows that in Version A the top 6 most downloaded programs accounted for 103 of the downloads during the testing period, which is more than a third of the total number of downloads of Version A during the testing period. It might be the case that these highly popular programs suppress other programs, in such a way that they are commonly favored by users over less popular programs. On the other hand, in Version B the top 6 most downloaded programs were ranked significantly lower and thus were only downloaded 10 times, which is less than 3.5% of the total downloads of Version B. Nonetheless, due to the fact that no data about the users' motivation are collected in online experiments, the exact reason for the increase in aggregated diversity cannot be ascertained without further studies. However, from the perspective of the non-personalized recommender system the first research question can be answered positively.

The second research question is:

Research Question 2: *How do the implemented approaches impact the novelty and serendipity levels of Catrobat's recommender system?*

Novelty was defined as the rate of recommended programs that had been unknown to a user until they were viewed. Serendipity was defined as the conversion rate of novel programs, which are programs that were previously unknown to a user.

The properties novelty and serendipity are usually evaluated for user-specific recommendations, as it is essential to know which items have been viewed by a user in the past in order to determine which items will be perceived as novel by a user. As described in section 6.3.1, it was not possible to collect sufficient data for the user-specific recommendation approaches in order to draw reasonable conclusions on the novelty and serendipity levels of Catrobat's recommender system. However, recommendations from non-personalized recommendation approaches also vary in their levels of novelty and thus serendipity. A statement about these properties can be made if the following premise regarding programs on Catrobat's sharing platform is accepted.

P_1: *The chance that popular programs are novel or serendipitous recommendations is lower than the chance that less popular programs are novel or serendipitous recommendations.*

The premise was formulated on the basis of the following reasoning. Version A of the second online experiment presented the most liked programs in descending order as recommendations. These recommended programs are highly similar to programs shown in other sections of Catrobat's home page, namely the sections of the most downloaded and most viewed programs. If a user has already explored one of these sections, the user will already be familiar with many of the programs that can be found in the section of recommended programs. This might cause low novelty and serendipity levels.

On the other hand, the re-ranking approach of Version B ranks highly popular programs significantly lower than they are ranked in Version A. As a result, a considerable amount of programs that are less popular are ranked higher in Version B than in Version A. Consequently it is argued that if P_1 is true, the levels of novelty and serendipity are likely to be higher in Version B than in Version A. This assumption is further backed by the number of views and downloads of programs that have been viewed or downloaded only from the section of recommended programs and from no other section of the home page of Catrobat's sharing platform during the testing period. As shown in table 6.6, in Version B, 40 different programs have been viewed only from the section of recommended programs, compared to 30 different programs for the baseline Version A. Considering the number of all views of recommended programs of Version A and B during the testing periods, this represents an increase of nearly 40%. Regarding the number of different programs that have only been downloaded from the section of recommended programs and from no other sections during the testing periods, it is evident that the number of Version B is more than two times higher than the number of Version A. Taking into consideration the number of all downloads of recommended programs during the testing periods, this means an increase of approximately 123.9%. Although it cannot be known which programs have already been viewed by guest users, it seems reasonable that programs that have been viewed or downloaded only in the section of recommended programs are more likely to be novel views or serendipitous downloads than programs which have also been viewed or downloaded in other sections of the home page of the sharing platform. Moreover, on the assumption that highly popular programs are in general less likely to be novel and serendipitous recommendations, the number of views and downloads of the top 6 and top 18 most downloaded programs in table 6.6 further indicates that Version B has higher levels of novelty and serendipity than the baseline Version A.

Therefore, on the basis of the data collected in the second online experiment, the assumption seems reasonable that the implemented re-ranking approach for the non-personalized part of Catrobat's recommender system has a substantial positive impact on its levels of novelty and serendipity.

7. Conclusions and Future Work

Finally, the previous chapters are recapitulated and the conclusions that can be drawn from this diploma thesis are presented in order to summarize the findings. To sum up, this chapter gives an overview of potential future work on Catrobat's recommender system and the like rating system.

7.1. Conclusions

The objective of this diploma thesis was to analyze and improve the recommender system of Catrobat's sharing platform. To do so, the necessary steps were divided into a theoretical and a practical part. The aim of the theoretical part was to acquire and present knowledge about recommender systems and their different types. Furthermore, Catrobat as an organization and as a programming language was introduced and its recommender system was analyzed. Based on this analysis, various potential improvements to the recommender system were identified and outlined. In a discussion, it was decided to increase the aggregated diversity of Catrobat's recommender system.

In the practical part of this thesis, which built on the knowledge gained in the theoretical part, three different re-ranking approaches were implemented in addition to the existing recommendation algorithms. Two approaches were employed for the user-specific part of the recommender system and one for the non-personalized part. The purpose of these re-ranking approaches was to increase the aggregated diversity of the recommender system. Additionally, a versatile and adaptable method of assigning users to groups based on predefined criteria was implemented, in order to avoid biases in the process.

Two online experiments were conducted in order to test the impact of the approaches on the system. Unfortunately, it was not possible to collect sufficient data to carry out a comprehensive evaluation of the two user-specific approaches. However, statistically significant data were found for the evaluation of the non-personalized approach. The data indicated that the re-ranking approach of non-personalized recommendations was successful in increasing the aggregated diversity of Catrobat's recommender system, without lowering its accuracy. Moreover, it is argued that the aforementioned approach most probably increased the levels of novelty and serendipity of Catrobat's recommender system.

7.2. Future Work

As discussed in chapter 4, there are a number of ways that are likely to further improve Catrobat's recommender system. However, this section focuses on potential improvements based on the results of the online experiments and experiences gained in the process.

First and foremost, due to the positive results achieved by the re-ranking approach for the non-personalized part of the recommender system, it is suggested that Catrobat should adopt said re-ranking approach.

Nonetheless, the first online experiment showed that the number of users who log into their user account when browsing the sharing platform is unexpectedly small. Possible reasons are stated in section 6.3.1. Given that there are over 88,000 programs on the sharing platform as of June 2019, user-specific recommendations offer great potential for Catrobat's users. Therefore, Catrobat is strongly advised to either create stronger incentives for users to log into their user account when browsing the sharing platform, or to introduce changes to the Pocket Code App so that once users are logged into Pocket Code, they would remain logged in when accessing the sharing platform. From the perspective of Catrobat's recommender system, it is believed that increasing the rate at which users log into their user account when browsing the sharing platform should be given the highest priority. In addition, once this issue is resolved, Catrobat is recommended to repeat the evaluation of the two user-specific re-ranking approaches, which were implemented as part of this thesis.

Furthermore, since the recommendation quality of collaborative filtering techniques depends heavily on the amount of feedback that users provided, it might be problematic that most of Catrobat's users who are eligible for recommendations only liked one program (as shown in table 3.1). At the time of writing this thesis, the action of liking a program requires a user to locate the program on the sharing platform because a program can only be liked on its details page. Making the like rating feature more easily accessible could help increase the overall rating activity of users. This could be achieved by giving an option to like downloaded programs (or the program's original source if the program has been modified). This way users would not need to visit the sharing platform and they would no longer have to search for the program they want to like themselves, thus the like rating system could be used with very little effort.

As outlined in section 4.1.2, even though Catrobat's recommender system has no critical scaling issues, the computation of user similarities is a highly resource-intensive process. However, it is questionable whether the current approach to calculating user similarities offline (as presented in section 3.5.2) will remain feasible if the number of users who are eligible for recommendation increases significantly. Therefore, it might be a wise decision to work on Catrobat's scalability before it becomes an immediate problem.

Another issue that is not directly related to the recommender system is the suboptimal use of space on the home page of Catrobat's sharing platform. This thesis aimed to implement measures to avoid the substantial overlap of programs between the sections of most downloaded and recommended programs. Although this issue has been taken care of, the large overlap of programs between the sections of most downloaded and most viewed programs persists, as illustrated in figure 4.3. Therefore, it is suggested that either both sections should be combined or one of them should be eliminated. A possible way of combining both sections would be to consider the number of views and the number of downloads in the ranking process by forming the mean. The freed up space could be used in many different ways, for instance by introducing a section of trending programs or a section with the purpose of promoting remixes.

Appendix

Appendix A. Abbreviations

ACM	Association of Computing Machinery
App	Application
ID	Identifier
IDE	Integrated Development Environment
PHP	PHP: Hypertext Preprocessor
XML	Extensible Markup Language

Appendix B.

Code of the Re-Ranking Approaches

In the practical part of this thesis, three re-ranking approaches have been applied to Catrobat's recommender system. The original recommendation process is presented in section 3.5.3. A version of the original code including comments is shown in code 3.1.

The first re-ranking approach is described in section 5.1 and the corresponding code is shown in code B.1. Section 5.2 outlines the second re-ranking approach; the corresponding code can be seen in code B.2. The re-ranking approach of the non-personalized recommendations can be found in section 5.3 and the corresponding code is shown in code B.3.

```
$most_downloaded_programs = $this->program_repository->
  getMostDownloadedPrograms($flavor, 75);
$ids_of_most_downloaded_programs = array_map(function ($program){
  return $program->getId();
}, $most_downloaded_programs);

foreach ($recommendation_weights as $key => $weight)
{
  $rank_in_top_downloads = array_search($key,
    $ids_of_most_downloaded_programs);
  if ($rank_in_top_downloads !== false)
  {
    $recommendation_weights[$key] = $weight * cos(deg2rad(
      75 - $rank_in_top_downloads));
  }
}

arsort($recommendation_weights);
```

Code B.1: This extract shows the code of the first re-ranking approach of user-specific recommendations, as described in section 5.1.

```php
arsort($recommendation_weights);

$recommendations_by_id = array_keys($recommendation_weights);

$average_user_similarity = array_sum(
  $similar_user_similarity_mapping) /
    count($similar_user_similarity_mapping);

$threshold_above_average_weight = $average_user_similarity * 1.25;
$threshold_high_weight = $average_user_similarity * 1.5;

$average_recommendation_weight = [];
$above_average_recommendation = [];
$top_recommendation = [];

foreach ($recommendations_by_id as $key => $recommendation_id)
{
  $average_recommendation_weight[$recommendation_id] =
    $recommendation_weights[$recommendation_id] /
      $number_of_recommendations[$recommendation_id];

  switch ($average_recommendation_weight[$recommendation_id])
  {
    case $average_recommendation_weight[$recommendation_id] >=
      $threshold_high_weight:
      $top_recommendation[$recommendation_id] =
        $this->program_like_repository->totalLikeCount(
          $recommendation_id);
      break;

    case $average_recommendation_weight[$recommendation_id] >=
      $threshold_above_average_weight:
      $above_average_recommendation[$recommendation_id] =
        $this->program_like_repository->totalLikeCount(
          $recommendation_id);
      break;

    default:
      // do nothing
  }
}

if (count($top_recommendation) >= 12)
{
  asort($top_recommendation);
  sort($above_average_recommendation);

  $recommendations_by_id = array_merge(array_keys(
    $above_average_recommendation), $recommendations_by_id);
  $recommendations_by_id = array_merge(
    array_keys($top_recommendation), $recommendations_by_id);

  $recommendations_by_id = array_unique($recommendations_by_id);
}

elseif (count($above_average_recommendation) > 0 ||
  count($top_recommendation) > 0)
{
  $above_average_recommendation = $above_average_recommendation +
    $top_recommendation;

  asort($above_average_recommendation);

  $recommendations_by_id = array_merge(array_keys(
    $above_average_recommendation), $recommendations_by_id);

  $recommendations_by_id = array_unique($recommendations_by_id);
}
```

Code B.2: This extract shows the code of the second re-ranking approach of user-specific recommendations, as described in section 5.2.

```
$most_liked_programs = $this->program_repository->
  getMostLikedPrograms($flavor, 0, 0);

$programs_total_likes = [];
foreach ($most_liked_programs as $most_liked_program)
{
  $program_id = $most_liked_program->getId();
  $programs_total_likes[$program_id] = $this->
    program_like_repository->totalLikeCount($program_id);
}

$most_downloaded_programs = $this->program_repository->
  getMostDownloadedPrograms($flavor, 45);
$ids_of_most_downloaded_programs = array_map(function ($program)
{
  return $program->getId();
}, $most_downloaded_programs);

foreach ($programs_total_likes as $program_id =>
  $number_of_likes)
{
  $rank_in_top_downloads = array_search($program_id,
    $ids_of_most_downloaded_programs);
  if ($rank_in_top_downloads !== false)
  {
    $programs_total_likes[$program_id] = $number_of_likes *
      cos(deg2rad(70 - $rank_in_top_downloads * 1.5))**2;
  }
}

arsort($programs_total_likes);
```

Code B.3: This extract shows the code of the re-ranking approach of non-personalized recommendations, as described in section 5.3.

Appendix C.

Code of the User Groups Entity

Code C.1 shows the code of the entity *User Test Group*. Section 5.4 describes the entity, which is designed to assign users to certain or random groups for testing purposes. It has been implemented so that users are not assigned to groups based on their language or location, hence eliminating possible biases. The entity has been used in an online experiment for the practical part of this thesis.

```
[...]
/**
 *
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 * @ORM\Table(name="user_test_group")
 *  @UniqueEntity("$user")
 *
 */

class UserTestGroup
{
  /**
   * @ORM\Id
   * @ORM\Column(type="integer", unique=true, nullable=false)
   */
  protected $user_id;

  /**
   * @ORM\Column(type="integer")
   */
  protected $group_number;

  /**
   * @ORM\Column(type="datetime")
   */
  protected $created_at;

  /**
   * @param int $user_id
   * @param int $group_number
   */
  public function __construct($user_id, $group_number)
  {
    if ($user_id !== null)
    {
      $this->setUserId($user_id);
      $this->setGroupNumber($group_number);
    }
  }

  /**
   * @ORM\PrePersist
   */
  public function updateTimestamps()
  {
    if ($this->getCreatedAt() === null)
    {
      $this->setCreatedAt(new \DateTime());
    }
  }

  /**
   * @param int $user_id
   */
  public function setUserId($user_id)
  {
    $this->user_id = $user_id;
  }

  /**
   * @return int
   */
  public function getUserId()
  {
    return $this->user_id;
    }

    /**
     * @param int $group_number
     */
    public function setGroupNumber($group_number)
    {
        $this->group_number = $group_number;
    }

    /**
     * @return int
     */
    public function getGroupNumber()
    {
        return $this->group_number;
    }

    /**
     * @param \DateTime $created_at
     *
     * @return $this
     */
    public function setCreatedAt(\DateTime $created_at)
    {
        $this->created_at = $created_at;
        return $this;
    }

    /**
     * @return \DateTime
     */
    public function getCreatedAt()
    {
        return $this->created_at;
    }
}
```

Code C.1: This extract shows the code of the entity *User Test Group*.

Appendix D.

Code of the User Similarity Computation

User similarities are represented by the Jaccard distance between two users, as described in section 3.5.2. The formula for the Jaccard distance is shown in equation (3.1). In the following, the code for the user similarity calculation, written by Samer (2017), is presented. The user similarities are calculated offline and updated regularly (Samer, 2017).

```
/**
 * @param $array1
 * @param $array2
 */
private function imitateMerge(&$array1, &$array2)
{
  foreach ($array2 as $i)
  {
    $array1[] = $i;
  }
}

/**
 *
 * Collaborative Filtering by using Jaccard Distance
 * As in this case we have to deal with TRUE/FALSE ratings (i.e.
 * user liked the program OR has not seen/liked it yet) the
 * Jaccard distance is used to measure the similarity between two
 * users.
 *
 *   n ... total number of users that have liked at least one
 *          program
 *   m ... total number of liked programs
 *
 * @see : http://infolab.stanford.edu/~ullman/mmds/ch9.pdf
 *        (section 9.3)
 * @time_complexity: O(n^2 * m)
 *
 * @param ProgressBar $progress_bar
 *
 * @throws \Doctrine\ORM\ORMException
 * @throws \Doctrine\ORM\OptimisticLockException
 */

public function computeUserLikeSimilarities($progress_bar = null)
{
  $users = $this->user_manager->findAll();
  $rated_users = array_unique(array_filter($users, function (
    $user) {
    /**
     * @var $user User
     */
    return (count($this->program_like_repository->findBy(
      ['user_id' => $user->getId()])) > 0);
  }));

  $already_added_relations = [];
  /**
   * @var $first_user User
   * @var $second_user User
   */
    foreach ($rated_users as $first_user)
    {
      if ($progress_bar != null)
      {
        $progress_bar->setMessage('Computing like similarity of
          user (#' . $first_user->getId() . ')');
      }

      $first_user_likes = $this->program_like_repository->findBy(
        ['user_id' => $first_user->getId()]);

      $ids_of_programs_liked_by_first_user = array_map(
        function ($like) {
        /**
```

```
         * @var $like ProgramLike
         */
        return $like->getProgramId();
      }, $first_user_likes);

      foreach ($rated_users as $second_user)
      {
        $key = $first_user->getId() . '_' . $second_user->getId();
        $reverse_key = $second_user->getId() . '_' .
          $first_user->getId();
        if (($first_user->getId() == $second_user->getId()) ||
         in_array($key, $already_added_relations)
          || in_array($reverse_key, $already_added_relations)
        )
        {
          continue;
        }

        $already_added_relations[] = $key;
        $second_user_likes = $this->program_like_repository->
          findBy(['user_id' => $second_user->getId()]);

        $ids_of_programs_liked_by_second_user = array_map(
          function ($like) {
          /**
           * @var $like ProgramLike
           */
          return $like->getProgramId();
        }, $second_user_likes);

        $ids_of_same_programs_liked_by_both = array_unique(
          array_intersect($ids_of_programs_liked_by_first_user,
            $ids_of_programs_liked_by_second_user));

        // make copy of array -> merge with empty array is fast
        // shortcut!
        $temp = array_merge([],
          $ids_of_programs_liked_by_first_user);

        // this imitate merge is way more faster than using
        // array_merge() with huge arrays!
        // -> this has a significant impact on performance here!
        $this->imitateMerge($temp,
          $ids_of_programs_liked_by_second_user);
        $ids_of_all_programs_liked_by_any_of_both = array_unique(
          $temp);

        $number_of_same_programs_liked_by_both =
          count($ids_of_same_programs_liked_by_both);
        $number_of_all_programs_liked_by_any_of_both =
          count($ids_of_all_programs_liked_by_any_of_both);

        if ($number_of_same_programs_liked_by_both == 0)
        {
          continue;
        }

        $jaccard_similarity = floatval(
          $number_of_same_programs_liked_by_both) /
           floatval($number_of_all_programs_liked_by_any_of_both);

        $similarity_relation = new UserLikeSimilarityRelation(
          $first_user, $second_user, $jaccard_similarity);

        $this->entity_manager->persist($similarity_relation);
        $this->entity_manager->flush($similarity_relation);
      }

      if ($progress_bar != null)
      {
        $progress_bar->clear();
        $progress_bar->advance();
        $progress_bar->display();
      }
    }
  }
```

Code D.1: This extract shows the computation of user similarities of Catrobat's recommender system.

Bibliography

Adomavicius, G. and Y. Kwon (May 2012). "Improving Aggregate Recommendation Diversity Using Ranking-Based Techniques." In: *IEEE Transactions on Knowledge and Data Engineering* 24.5, pp. 896–911. ISSN: 1041-4347. DOI: 10.1109/TKDE.2011.15 (cit. on pp. 15, 76, 77, 80, 86–90).

Adomavicius, Gediminas and Youngok Kwon (2009). "Towards more diverse recommendations: Item re-ranking methods for recommender systems." In: *In Workshop on Information Technologies and Systems* (cit. on p. 14).

Ahn, Hyung Jun (Jan. 2008). "A New Similarity Measure for Collaborative Filtering to Alleviate the New User Cold-starting Problem." In: *Inf. Sci.* 178.1, pp. 37–51. ISSN: 0020-0255. DOI: 10.1016/j.ins.2007.07.024. URL: https://doi.org/10.1016/j.ins.2007.07.024 (cit. on p. 71).

Amazon (2019). URL: https://www.amazon.com/ (visited on 03/20/2019) (cit. on pp. 1, 12, 18).

Andreas Töscher Michael Jahrer, Robert M. Bell (2009). *The BigChaos Solution to the Netflix Grand Prize* (cit. on p. 7).

Balabanović, Marko and Yoav Shoham (Mar. 1997). "Fab: Content-based, Collaborative Recommendation." In: *Commun. ACM* 40.3, pp. 66–72. ISSN: 0001-0782. DOI: 10.1145/245108.245124. URL: http://doi.acm.org/10.1145/245108.245124 (cit. on p. 6).

Black Duck Software, Inc. (2019). *Openhub Catrobat Project Site*. URL: https://www.openhub.net/p/catrobat (visited on 03/01/2019) (cit. on p. 53).

Breese, John S., David Heckerman, and Carl Kadie (1998). "Empirical Analysis of Predictive Algorithms for Collaborative Filtering." In: *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*. UAI'98. Madison, Wisconsin: Morgan Kaufmann Publishers Inc., pp. 43–52. ISBN: 1-55860-555-X. URL: http://dl.acm.org/citation.cfm?id=2074094.2074100 (cit. on p. 27).

Burke, Robin (May 2000). "Knowledge-Based Recommender Systems." In: *Encyclopedia of library and information systems* 69 (cit. on p. 39).

Burke, Robin (Nov. 2002). “Hybrid Recommender Systems: Survey and Experiments.” In: *User Modeling and User-Adapted Interaction* 12. DOI: 10.1023/A:1021240730564 (cit. on pp. 28, 31, 43, 45–47, 71).

Burke, Robin (2007). “Hybrid Web Recommender Systems.” In: *The Adaptive Web: Methods and Strategies of Web Personalization*. Ed. by Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 377–408. ISBN: 978-3-540-72079-9. DOI: 10.1007/978-3-540-72079-9_12. URL: https://doi.org/10.1007/978-3-540-72079-9_12 (cit. on p. 28).

Celma, Òscar and Perfecto Herrera (2008). “A New Approach to Evaluating Novel Recommendations.” In: *Proceedings of the 2008 ACM Conference on Recommender Systems*. RecSys ’08. Lausanne, Switzerland: ACM, pp. 179–186. ISBN: 978-1-60558-093-7. DOI: 10.1145/1454008.1454038. URL: http://doi.acm.org/10.1145/1454008.1454038 (cit. on p. 14).

Claypool, Mark et al. (1999). “Combining Content-Based and Collaborative Filters in an Online Newspaper.” In: *In Proceedings of ACM SIGIR Workshop on Recommender Systems* (cit. on pp. 28, 46).

Doctrine (2019). URL: https://www.doctrine-project.org/index.html (visited on 03/16/2019) (cit. on pp. 58, 61).

Ebadi, Ashkan and Adam Krzyzak (Jan. 2016). “A hybrid multi-criteria hotel recommender system using explicit and implicit feedbacks.” In: *International Scholarly and Scientific Research & Innovation*. Vol. 10. 8, pp. 1450–1458 (cit. on p. 21).

Ekstrand, Michael D. et al. (2014). “User Perception of Differences in Recommender Algorithms.” In: *Proceedings of the 8th ACM Conference on Recommender Systems*. RecSys ’14. Foster City, Silicon Valley, California, USA: ACM, pp. 161–168. ISBN: 978-1-4503-2668-1. DOI: 10.1145/2645710.2645737. URL: http://doi.acm.org/10.1145/2645710.2645737 (cit. on pp. 14, 22).

Felfernig, Alexander and Robin Burke (Jan. 2008). “Constraint-based recommender systems: Technologies and research issues.” In: *ACM International Conference Proceeding Series*, p. 3. DOI: 10.1145/1409540.1409544 (cit. on pp. 38–40, 43, 44).

Felfernig, Alexander, Michael Jeran, et al. (Dec. 2014). “Basic Approaches in Recommendation Systems.” In: *Recommendation Systems in Software Engineering*, pp. 15–37. DOI: 10.1007/978-3-642-45135-5__2 (cit. on pp. 31, 35, 36).

Fornaciari, Tommaso and Massimo Poesio (Apr. 2014). “Identifying fake Amazon reviews as learning from crowds.” In: *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*. Gothenburg, Sweden: Association for Computational Linguistics, pp. 279–287. DOI: 10.3115/v1/E14-1030. URL: https://www.aclweb.org/anthology/E14-1030 (cit. on p. 12).

Goldberg, David et al. (Dec. 1992). “Using Collaborative Filtering to Weave an Information Tapestry.” In: *Commun. ACM* 35.12, pp. 61–70. ISSN: 0001-0782. DOI: 10.1145/138859.138867. URL: http://doi.acm.org/10.1145/138859.138867 (cit. on p. 5).

Goodreads Homepage (2019). URL: https://www.goodreads.com/ (visited on 04/02/2019) (cit. on p. 18).

Goodreads Recommendations (2019). URL: https://www.goodreads.com/recommendations (visited on 05/25/2019) (cit. on p. 19).

Google LLC (2019a). *Google - Terms of Service*. URL: https://play.google.com/intl/en-us_us/about/play-terms/index.html (visited on 03/04/2019) (cit. on p. 58).

Google LLC (2019b). *Google Analytics*. URL: https://analytics.google.com/analytics/web/ (visited on 06/04/2019) (cit. on p. 101).

Google LLC (2019c). *Google Play Store*. URL: https://play.google.com/store (visited on 03/04/2019) (cit. on p. 58).

Herlocker, Jonathan L., Joseph A. Konstan, Al Borchers, et al. (1999). “An Algorithmic Framework for Performing Collaborative Filtering.” In: *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’99. Berkeley, California, USA: ACM, pp. 230–237. ISBN: 1-58113-096-1. DOI: 10.1145/312624.312682. URL: http://doi.acm.org/10.1145/312624.312682 (cit. on pp. 25, 27, 70).

Herlocker, Jonathan L., Joseph A. Konstan, and John Riedl (2000). “Explaining Collaborative Filtering Recommendations.” In: *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*. CSCW ’00. Philadelphia, Pennsylvania, USA: ACM, pp. 241–250. ISBN: 1-58113-222-0. DOI: 10.1145/358916.358995. URL: http://doi.acm.org/10.1145/358916.358995 (cit. on p. 14).

Herlocker, Jonathan L., Joseph A. Konstan, Loren G. Terveen, et al. (Jan. 2004). “Evaluating Collaborative Filtering Recommender Systems.” In: *ACM Trans. Inf. Syst.* 22.1, pp. 5–53. ISSN: 1046-8188. DOI: 10.1145/

963770.963772. URL: http://doi.acm.org/10.1145/963770.963772 (cit. on pp. 9–11).

International Catrobat Association (2019a). *Catrobat - German Project Home Page*. URL: https://www.catrobat.org/de/#mission (visited on 03/01/2019) (cit. on pp. 1, 53).

International Catrobat Association (2019b). *Catrobat - Sharing Platform*. URL: https://share.catrob.at/pocketcode/ (visited on 03/04/2019) (cit. on pp. 53, 56, 75, 78, 79, 96).

International Catrobat Association (2019c). *Catrobat - Terms of Service*. URL: https://share.catrob.at/pocketcode/termsOfUse (visited on 03/04/2019) (cit. on p. 59).

Jannach, Dietmar, Lukas Lerche, and Markus Zanker (2018). "Recommending Based on Implicit Feedback." In: *Social Information Access: Systems and Technologies*. Ed. by Peter Brusilovsky and Daqing He. Cham: Springer International Publishing, pp. 510–569. ISBN: 978-3-319-90092-6. DOI: 10.1007/978-3-319-90092-6_14. URL: https://doi.org/10.1007/978-3-319-90092-6_14 (cit. on pp. 17, 18, 20).

Jannach, Dietmar, Markus Zanker, et al. (2010). *Recommender Systems - An Introduction*. English. 1st ed. United Kingdom: Cambridge University Press. ISBN: 978-0-521-49336-9 (cit. on pp. 4, 17, 23, 24, 26, 28, 29, 35, 36, 38–40, 44–46).

Javari, Amin and Mahdi Jalili (June 2014). "A probabilistic model to resolve diversity-accuracy challenge of recommendation systems." In: *Knowledge and Information Systems*. DOI: 10.1007/s10115-014-0779-2 (cit. on p. 80).

Karakaya, Mahmut and Tevfik Aytekin (Nov. 2017). "Effective methods for increasing aggregate diversity in recommender systems." In: *Knowledge and Information Systems*. DOI: 10.1007/s10115-017-1135-0 (cit. on p. 80).

Lifelong Kindergarten Group, MIT Media Lab (2019). *Scratch*. URL: https://scratch.mit.edu (visited on 03/01/2019) (cit. on p. 55).

Linden, Greg, Brent Smith, and Jeremy York (Jan. 2003). "Amazon.Com Recommendations: Item-to-Item Collaborative Filtering." In: *IEEE Internet Computing* 7.1, pp. 76–80. ISSN: 1089-7801. DOI: 10.1109/MIC.2003.1167344. URL: http://dx.doi.org/10.1109/MIC.2003.1167344 (cit. on p. 29).

Liu, Haifeng et al. (2014). "A new user similarity model to improve the accuracy of collaborative filtering." In: *Knowledge-Based Systems* 56, pp. 156–166. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2013.

11.006. URL: http://www.sciencedirect.com/science/article/pii/S0950705113003560 (cit. on p. 62).

Lops, Pasquale, Marco de Gemmis, and Giovanni Semeraro (Jan. 2011). "Content-based Recommender Systems: State of the Art and Trends." In: *Recommender Systems Handbook*, pp. 73–105. DOI: 10.1007/978-0-387-85820-3_3 (cit. on pp. 36–38).

McNee, Sean M., John Riedl, and Joseph A. Konstan (2006). "Being Accurate is Not Enough: How Accuracy Metrics Have Hurt Recommender Systems." In: *CHI '06 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '06. Montreal, Quebec, Canada: ACM, pp. 1097–1101. ISBN: 1-59593-298-4. DOI: 10.1145/1125451.1125659. URL: http://doi.acm.org/10.1145/1125451.1125659 (cit. on p. 76).

Netflix, Inc. (2009a). *Netflix Prize*. URL: https://www.netflixprize.com/ (visited on 03/22/2019) (cit. on p. 7).

Netflix, Inc. (2009b). *Netflix Prize Leaderboard*. URL: https://www.netflixprize.com/leaderboard.html (visited on 03/22/2019) (cit. on p. 7).

Nielsen, Jakob (Nov. 2013). *Conversion rate*. URL: https://www.nngroup.com/articles/conversion-rates/ (visited on 05/13/2019) (cit. on p. 5).

Niemann, Katja and Martin Wolpers (2013). "A New Collaborative Filtering Approach for Increasing the Aggregate Diversity of Recommender Systems." In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. Chicago, Illinois, USA: ACM, pp. 955–963. ISBN: 978-1-4503-2174-7. DOI: 10.1145/2487575.2487656. URL: http://doi.acm.org/10.1145/2487575.2487656 (cit. on p. 80).

Oracle Corporation (2019). *MySQL*. URL: https://www.mysql.com/ (visited on 03/16/2019) (cit. on p. 58).

Painsi, Robert (Mar. 2019). *Catrobat Statistics*. URL: http://robertpainsi.github.io/catrobat/statistics/?period=overall#Catrobat-Statistics (visited on 03/04/2019) (cit. on pp. 53, 59).

Patil, C. B. and R. B. Wagh (Jan. 2014). "A multi-attributed hybrid re-ranking technique for diversified recommendations." In: *2014 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pp. 1–6. DOI: 10.1109/CONECCT.2014.6740332 (cit. on p. 80).

Pazzani, Michael J. and Daniel Billsus (2007). "Content-Based Recommendation Systems." In: *The Adaptive Web: Methods and Strategies of Web*

Personalization. Ed. by Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 325–341. ISBN: 978-3-540-72079-9. DOI: 10.1007/978-3-540-72079-9_10. URL: https://doi.org/10.1007/978-3-540-72079-9_10 (cit. on p. 35).

PHP Group (2019). *PHP*. URL: http://www.php.net/ (visited on 03/16/2019) (cit. on p. 58).

RecSys Community (2019). *ACM Recommender System Conference*. URL: https://recsys.acm.org/ (visited on 03/22/2019) (cit. on p. 7).

Resnick, Paul, Neophytos Iacovou, et al. (1994). "GroupLens: An Open Architecture for Collaborative Filtering of Netnews." In: *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*. CSCW '94. Chapel Hill, North Carolina, USA: ACM, pp. 175–186. ISBN: 0-89791-689-1. DOI: 10.1145/192844.192905. URL: http://doi.acm.org/10.1145/192844.192905 (cit. on p. 6).

Resnick, Paul and Hal R. Varian (Mar. 1997). "Recommender Systems." In: *Commun. ACM* 40.3, pp. 56–58. ISSN: 0001-0782. DOI: 10.1145/245108.245121. URL: http://doi.acm.org/10.1145/245108.245121 (cit. on p. 7).

Ricci, Francesco, Lior Rokach, and Bracha Shapira (2015). *Recommender Systems Handbook*. 2nd. Springer Publishing Company, Incorporated. ISBN: 9781489976369 (cit. on pp. 3–6, 8–17, 19, 21, 25, 45, 46).

Samer, Ralph (May 2017). "Construction of a Recommender System for Catrobat's Collaborative Web Community." MA thesis. Graz University of Technology (cit. on pp. 25, 59, 62, 63, 65, 67, 69, 70, 81, 100, 101, 104, 133).

Sarwar, Badrul et al. (2001). "Item-based Collaborative Filtering Recommendation Algorithms." In: *Proceedings of the 10th International Conference on World Wide Web*. WWW '01. Hong Kong, Hong Kong: ACM, pp. 285–295. ISBN: 1-58113-348-0. DOI: 10.1145/371920.372071. URL: http://doi.acm.org/10.1145/371920.372071 (cit. on pp. 27, 29, 30).

Shani, Guy and Asela Gunawardana (Jan. 2011). "Evaluating Recommendation Systems." In: *Recommender Systems Handbook*. Vol. 12, pp. 257–297. DOI: 10.1007/978-0-387-85820-3_8 (cit. on pp. 46, 48–50).

Shardanand, Upendra and Pattie Maes (1995). "Social Information Filtering: Algorithms for Automating 'Word of Mouth'." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '95. Denver, Colorado, USA: ACM Press/Addison-Wesley Publishing Co., pp. 210–217. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223931. URL: http://dx.doi.org/10.1145/223904.223931 (cit. on p. 25).

Son, Le (Dec. 2014). "Dealing with the new user cold-start problem in recommender systems: A comparative review." In: *Information Systems* 58. DOI: 10.1016/j.is.2014.10.001 (cit. on p. 71).

Symfony Homepage (2019). URL: https://symfony.com/ (visited on 03/15/2019) (cit. on p. 58).

Thi Do, Minh-Phung, Dung Van Nguyen, and Academic Network of Loc Nguyen (Aug. 2010). "Model-based approach for Collaborative Filtering." In: *The 6th International Conference on Information Technology for Education, 2010* (cit. on pp. 31–33).

Trivago (2019). URL: https://www.trivago.com/ (visited on 04/17/2019) (cit. on pp. 39, 41, 42).

Tsang, Edward (Jan. 1993). *Foundations of Constraint Satisfaction*. ISBN: 978-0-12-701610-8 (cit. on p. 43).

Ungar, Lyle and Dean P. Foster (1998). "Clustering Methods for Collaborative Filtering." In: *AAAI Technical Report WS-98-08*. AAAI Press (cit. on p. 33).

Wikimedia Commons, User: Chire (2010). *Iris flower data set, clustered using k means (left) and true species in the data set (right).* Used under public domain license. URL: https://commons.wikimedia.org/wiki/File:Iris_Flowers_Clustering_kMeans.svg (visited on 06/09/2019) (cit. on p. 34).

Xue, Gui-Rong et al. (2005). "Scalable Collaborative Filtering Using Cluster-based Smoothing." In: *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '05. Salvador, Brazil: ACM, pp. 114–121. ISBN: 1-59593-034-5. DOI: 10.1145/1076034.1076056. URL: http://doi.acm.org/10.1145/1076034.1076056 (cit. on p. 32).

YouTube Homepage (2019). URL: https://www.youtube.com/ (visited on 04/03/2019) (cit. on p. 22).