Ralph G. Samer, BSc

Construction of a Recommender System for Catrobat's Collaborative Web Community

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

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

Institute for Software Technology

Co-Supervisor
Bernadette Spieler, MSc.

Graz, May 2017

AFFIDAVIT

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Abstract

The introduction of the Internet to billions of users worldwide has led to an unprecedented trend in the digitalization of data, resulting in a growth of data that has expanded exponentially across cyberspace. Today, a huge amount of that data is generated by users via collaborative software, which is largely represented by social platforms like Facebook, Twitter, and Google+. With the increasing amount of produced collaborative data in such platforms, finding relevant pieces of information becomes a challenging task for a user. Hence, there is a significant demand for solutions that help users in finding desired content within those huge data masses. Recommender Systems represent one common scientific approach to tackle this problem. These systems are the main research topic of this Master's thesis. Since the research area of recommender systems is very broad and multi-disciplinary, the research scope of this thesis has to be narrowed down to only the most relevant examples of these systems. Thereby, this thesis primarily focuses on the practical use of *collaborative filtering* recommender techniques for the Catrobat project.

During the project portion of this thesis, three basic recommender approaches were implemented and evaluated. The first two approaches are based on a collaborative filtering technique and the third approach is a content-based recommender system which shows a graph of content-similar items (i.e., Catrobat programs). All approaches make use of either collaborative remixing data or collaborative data from like ratings which had to be collected by the system. The more collaborative data the system collects about its users, the better the system knows them. Therefore, user profiles containing an appropriate amount of user data are essential for making good user-specific recommendations. Moreover, the quality of these recommendations needs to be evaluated. Consequently, the main goal of this thesis is to use the results of the evaluation to advise the Catrobat project on how to improve the quality of the recommendations in order to increase overall user activity.

The main conclusion of this thesis is that recommendations based on a collaborative filtering technique have a positive impact on both download and remixing activity. Moreover, the findings reveal that the collected like rating data appears to be the preferred input source for a collaborative filtering approach in the context of Catrobat. The obtained results also indicate that the recommendations generated by a content-based recommender system can improve the navigation through a set of content-related items in Catrobat. Finally, these findings have been the basis for the suggestions given to Catrobat, which include the adaption of the existing system to a hybrid combination of both collaborative filtering approaches, the active promotion of the Scratch converter project, the inclusion of acknowledgement information in the detail data of remixed programs, and further statistical investigations of the implemented approaches.

Kurzfassung

Seit der Einführung des Internets steigt die Menge an digitalen Daten weltweit kontinuierlich an. Den Großteil dieser Daten stellen heutzutage jene Daten dar, die durch Benutzer und Benutzerinnen mit Hilfe von kollaborativer Software erzeugt werden. Repräsentative Beispiele für kollaborative Softwaresysteme sind vor allem soziale Plattformen wie etwa Facebook, Twitter oder Google+. Mit der rasant wachsenden Menge an solchen "kollaborativ-erzeugten" Daten wird es für Benutzer/innen immer schwerer, relevante Inhalte darin zu finden. Folglich steigt auch die Nachfrage nach passenden Lösungsmöglichkeiten zum Suchen solcher relevanten Informationen stark an. Ein bekannter Ansatz zur Lösung dieses Problems sind Recommender Systeme, die auch den primären Forschungsgegenstand dieser Masterarbeit darstellen. Aufgrund der großen Vielfalt an verschiedenen Einsatzmöglichkeiten von Recommender Systemen und des weitreichenden interdisziplinären Forschungsgebiets dieser Systeme ist es erforderlich den Fokus dieser Masterarbeit einzuschränken. Diese Masterarbeit beschäftigt sich daher mit dem praktischen Einsatz von Empfehlungsdiensten im Catrobat Projekt und legt den Fokus vor allem auf das *kollaborative Filtern* von Inhalten.

Im Rahmen dieser Arbeit wurden drei Varianten von Empfehlungsdiensten implementiert. Dabei basieren die ersten beiden Varianten auf der kollaborativen Filtermethode. Als dritte Variante wurde ein inhaltsbasiertes Recommender System implementiert, das als Ergebnis einen Graphen mit inhaltlich zusammenhängenden Objekten (Catrobat-Programme) abbildet. Diese zusammengehörigen Catrobat-Programme werden dann als Empfehlungen dem/der Benutzer/in vorgeschlagen. Alle drei Varianten verwenden dabei als Grundlage für ihre Empfehlungen vom System gesammelte kollaborative Daten. Dies sind entweder extrahierte Remixing-Daten von Catrobat-Programmen oder die gesammelten Bewertungsdaten der Benutzer und Benutzerinnen. Dabei ist vor allem die Menge an verfügbaren Daten von entscheidender

Bedeutung für die Qualität der Empfehlungen. Je mehr solcher kollaborativer Daten einem Recommender System zur Verfügung stehen, desto genauer kann es die Empfehlungen an die individuellen Vorlieben der Benutzer/innen anpassen. Daher ist das Erstellen von detaillierten Benutzerprofilen essentiell für die Generierung von userspezifischen Empfehlungen. Ferner spielt auch die Evaluierung der Empfehlungen eine bedeutende Rolle. Daher wurden die drei implementierten Varianten im Rahmen dieser Masterarbeit genauer untersucht. Aufbauend auf den erhaltenen Evaluierungsergebnissen ist das Ziel dieser Masterarbeit Lösungsvorschläge und Ideen für Catrobat zu präsentieren, um die Qualität der Empfehlungen zu verbessern und die Benutzeraktivität zu erhöhen.

Die grundlegende Erkenntnis dieser Masterarbeit ist, dass jene Empfehlungen, die auf der kollaborativen Filtermethode beruhen, einen positiven Einfluss auf die allgemeine Download- und Remixingaktivität der Benutzer und Benutzerinnen hat. Dabei deuten die Ergebnisse darauf hin, dass die Bewertungsdaten der Benutzer und Benutzerinnen eine hervorragende Eingabequelle für die in Catrobat eingesetzten Empfehlungsdienste darstellen. Eine weitere wichtige Beobachtung zeigt, dass die Empfehlungen des inhaltsbasierten Recommender Systems positiven Einfluss auf die Navigation innerhalb der inhaltlich zusammenhängenden Objekte hat. Basierend auf diesen Erkenntnissen werden dann Vorschläge an Catrobat gemacht. Diese umfassen unter anderem den Umstieg auf eine hybride Kombination beider vorhandener kollaborativer Varianten, die Förderung des Scratch Konverter Projekts, die Aufforderung zur Angabe eines Anmerkungs- bzw. Danksagungstextes vor dem Hochladen eines remixten Catrobat Programms, sowie eine erneute detaillierte Untersuchung aller implementierten Varianten nach Erreichen einer gewissen Datenmenge.

Contents

List of Figures

List of Tables

1. Introduction

With the introduction of the first personal computers in the 1970s and the invention of Arpanet, the early predecessor of today's Internet, a revolutionary new age of information was born. In this completely new era of human history, technology has drastically changed the way how people communicate and share information. The transition from the industrial age to the information age has gone hand in hand with the increasing digitalization of data and the growing importance of communication networks, like for example, the Internet. As a consequence, the need to collect, group and store information on servers has increased significantly as well.

In particular, huge masses of collaborative user data are created on common social platforms, like Facebook, Twitter, or Google+. Beyond that, there also exist other kinds of social platforms such as media sharing services which provide executable or playable content. Examples of media sharing platforms include Youtube, Netflix, Spotify, Apple iTunes Store, Google Play, Apple App Store, and many others.

Another example of media sharing platform is Catrobat's educational community website[1], which hosts programs created by children and teenagers on their mobile smartphones or tablet devices (see Section 5.4). These programs are called *Catrobat programs*. Unlike a typical app store, uploaded Catrobat programs can not only be commented and rated by other users, but also modified and uploaded again as new versions. This collaborative learning approach is called *remixing* and is explained in Section 3.4.

As the number of users is growing rapidly, so does the amount of Catrobat programs and thus the amount of collaborative data. This makes searching for programs more relevant and essential to users. Consequently, the demand

[1]Catrobat's Community Website: `http://share.catrob.at`

for finding a desired piece of information becomes a crucial obligation for a software collaboration system like Catrobat's community website.

Therefore, modern *collaborative software* systems strive to use *collective intelligence* to fully take advantage of their collaborative data. Furthermore, these collaborative applications can be extended with recommender systems in such a way that they are able to support their users in the finding process before the users even start searching. In other words, an important task is to recommend yet unseen items to users which might be of interest to them. The difficulty is not so much to figure out the preferences of users, but to automatically draw correct conclusions.

This Master's thesis aims to tackle this problem and focuses on the use of collaborative data in order to enhance the quality of results generated by recommender systems. Essentially, the fundamental goal of this thesis is to answer the following research questions:

- **Research Question 1:**
  Do recommendations of Catrobat programs have a positive impact on the overall download activity?

- **Research Question 2:**
  Can a user-based recommendation approach based on collaborative filtering produce recommendations of higher quality compared to those generated by naive recommendation approaches? Which of the two implemented user-based collaborative filtering approaches performs best: (a) the first approach, which is based on remixing data, or (b) the second approach, which is based on like ratings?

- **Research Question 3:**
  Given the set of Catrobat programs remixed by all users, can these remixed programs be used to generate high quality content-based recommendations for other users in order to increase the overall remixing activity?

This thesis is structured into the following chapters. Chapter 2 defines the most important technical terms used in this thesis. Chapter 3 highlights different aspects of collaborative software and emphasises the deep relationship between collective intelligence and collaboration. In addition, an overview of the large spectrum of collaborative software is given and approaches on how to classify such software are presented. At the end of this chapter, the focus moves towards collaborative learning and the aforementioned *remixing* learning approach is discussed in depth. Chapter 4 presents those systems and appropriate modes for predicting and recommending undiscovered programs based on collaborative data gathered by such collaborative systems. In particular, the focus lies on collaborative recommendation approaches, which are also referred to as *collaborative filtering* recommendation approaches. Chapter 5 introduces the Catrobat organisation and project. This chapter initiates the practical part of this Master's thesis. Chapter 6 discusses the implementation of the practical project of this thesis and the used evaluation techniques. This chapter concentrates on the implementation of two different collaborative filtering recommendation approaches and also assesses the performance of an alternative content-based recommendation approach which is based on remixing. At the end of the chapter, online tests are described which have been conducted in order to evaluate the performance of the implemented approaches. Chapter 7 presents the evaluation results of these tests and discusses the significance of these findings. Finally, Chapter 8 provides a brief recapitulation of all previous chapters, emphasizes the outcome of this thesis, and provides ideas for future work.

2. Glossary

This chapter defines relevant technical terms used in this thesis.

Ajax: As described by Garrett, 2005, the term Ajax is a combination of several different technologies. It consists of a standards-based presentation using XHTML and CSS, dynamic display and interaction using the Document Object Model, data interchange and manipulation using XML and XSLT, asynchronous data retrieval using XMLHttpRequest, and JavaScript binding everything together.

Algorithm: In computer programming, an algorithm defines a function which implements a sequence of operations in order to perform a task (Terms, 2017a). In its simplest form, an algorithm can be a primitive operation, such as adding two numbers. Normally, an algorithm has more complex instructions, such as playing a compressed multimedia file.

Click-Through Rate: The Click-Through Rate (CTR) is a measure that describes the ratio of the number of clicks (for example, on a link) to the number of page impressions as described by *OnPageWiki* 2017.

Conversion Rate: According to *Nielsen Norman Group* 2017, the conversion rate is the percentage of users who take a desired action. A typical example of the conversion rate defines the percentage of website visitors who buy something on the site. Other examples are: buying something on an e-commerce site, becoming a registered user, downloading software, etc.

Hyperlink: As defined by Terms, 2017b, a hyperlink can represent an image, a phrase, or a single word which can be clicked on in order to jump to a different section on a page or different page. Hyperlinks, which are also known as “links”, allow users to navigate within a website and can be found in almost every web page.

Overfitting: According to Techopedia, 2017a, overfitting is an effect that occurs in situations where a model tries to find a trend in data which is extremely noisy. This means that the model is unable to generate any general and reliable predictions. Often the reason for overfitting is that data is represented by a very complex model with too many model parameters. The predictions of an overfitted model are inaccurate since the model does not reflect the real trend in the data.

Programmable Media: According to Monroy-Hernández, 2007, programmable media refers to media content (such as audio files, videos, images, and text) which is controlled by a certain behavior. An example is an image of an animal for which an behavior is defined like “move the animal 10 steps forward when the space key is pressed”. Examples of platforms for creating programmable media content include Scratch and Catrobat.

Representational State Transfer: As described by Techopedia, 2017b, representational state transfer, often abbreviated as REST, is a distributed framework which uses web technologies and web protocols. Its architecture supports server and client interactions which accomplish the transfer of resources. REST is a programming paradigm often used in the context of web services.

Web 2.0 According to Terms, 2017c, Web 2.0 is the second generation of the world wide web and provides a variety of new functionality and features for user interaction on the web. With the introduction of Web 2.0 in 2004, web pages have become more dynamic. Web 2.0 allows users to collaboratively work together and interact with others in real-time on the web. Examples include wikis, blogs, social networking and web applications.

3. Collaborative Software

The main goal of this chapter is to shed light on the different aspects of collaborative software. At the beginning of this chapter, the concept *collective intelligence* is examined and its importance in the context of the web as a result of a knowledge-based human superorganism is discussed in more detail. This leads to classification approaches and examples of collaborative software tools which aim to exploit the full potential of the collective intelligence process behind this human superorganism. Finally, a collaborative learning method for (group) project work, called *remixing*, is presented which forms the basis for later chapters of this thesis.

3.1. Collective Intelligence

In 1911, the entomologist William Morton Wheeler (Wheeler, 1911) made the observation that individual ants worked as part of a cooperating single working unit, namely the colony, which represented a "superorganism" that was indistinguishable from a single organism.

Figure 3.1 shows an experiment conducted by researchers at Princeton University (Reid et al., 2015). The experiment illustrates an example on how columns of ants can instinctively co-operate to intelligently bypass obstacles in order to forage for food or supplies.

The scientific findings of Wheeler can be understood as a predefinition of the term *collective intelligence*, which has since been redefined and further developed by many other sociologists like Durkheim, 1912 or Wells, 1938. For example, Durkheim interpreted society as the sole source of human logical thought. In fact, a lot of people's knowledge originates from several sources, e.g., from books, scientific papers, seminars, etc. Obviously, these sources

Figure 3.1.: Ant workers searching for food (*Army ants' "living" bridges span collective intelligence, "swarm" robotics* 2015)

Source: https://blogs.princeton.edu/research/2015/11/24/army-ants-living-bridges-span-collective-intelligence-swarm-robotics-pnas

are based on other people's knowledge, which perfectly demonstrates the presence of a human superorganism.

More precisely, according to Levy, 1997, p. 13:

> *"Collective Intelligence is a form of universally distributed intelligence, constantly enhanced, coordinated in real time, and resulting in the effective mobilization of skills. [...] The basis and goal of collective intelligence is mutual recognition and enrichment of individuals rather than the cult of fetishized or hypostatized communities."*

3.1.1. Principles of Collective Intelligence

Tapscott and Williams, 2006 interpreted collective intelligence as some kind of mass collaboration. In their book the authors also introduced the term *Wikinomics*, which describes a new way of doing business in a revolutionary collaborative context. The basic idea behind this concept is, that people work independently on a joint project, meaning their work has to be organized individually by themselves and hence no central work hierarchies exist (Tapscott and Williams, 2006). Examples are the social contribution of users to social networks like Facebook or Twitter, the development of open source software projects such as the Linux kernel, the sharing of knowledge on various collaborative platforms such as video platforms like YouTube, online encyclopedias like Wikipedia, or the knowledge contribution to scientific projects such as the human genome project.

According to Tapscott and Williams, 2006, Wikinomics consists of the following four central concepts.

i. **Openness:**
   The Openness principle references the *willingness* of individuals to cooperate and share ideas and intellectual property.
   For example, big internet companies like Google, eBay, Facebook, or Amazon open up their platforms to extend the scope and accelerate the process of innovation. These platforms can be enriched with information by millions of customers and partners to take advantage of synergy effects (Tapscott and Williams, 2006).

ii. **Peering:**
   In order to exploit the full potential of human ingenuity and human skills, individuals need to be able to freely modify and extend content created by others. Consequently, peering encourages self-organization, leading to better results for certain tasks compared to those results achieved by hierarchically organized work. For example, IBM invests a large amount of money every year to support the Linux community[1]. In return, the company saves a vast amount of development costs and earns billions in revenue with the use of Linux-related software each year. This

[1]Members of Linux Foundation: https://www.linuxfoundation.org/members/corporate

way, the Linux community can be seen as an additional extension added to IBM's human capital, whereas IBM is unable to control what the developers in the community actually do (Tapscott and Williams, 2006).

iii. **Sharing:**
In contrast to the secrecy of all intellectual property, the exchange of some valuable ideas with the community may open up new horizons for a company, e.g., market expansion, lowering costs or building new communities. Moreover, a high willingness of a number of organizations and individuals to share ideas results in powerful share-communities. Another important aspect in the context of sharing is licensing.
Examples of popular licenses for open source software projects include Apache license, BSD license, GNU General Public license, MIT license, and many others. By choosing the appropriate license one can, for instance, protect the freedom of the project, but open it for use, redistribution and modification. Nowadays, a countless number of smart firms makes use of sharing their ideas, but still keeps their potential patent ideas secret. For example, after the company Lego opened up and shared the source code for their Lego Mindstorm products, some customers made valuable contributions to the project (Tapscott and Williams, 2006).

iv. **Acting Globally:**
The invention of the Internet, the expansion of communication infrastructures such as mobile networks and the increased personal mobility, has literally pushed globalization forward. Furthermore, the communication afford and costs to operate businesses across the globe decreased significantly. The globalization process leads on to blurred economic and geographic borders that previously insulated companies and governments. Moreover, global alliances enable businesses to access new markets and to benefit from new ideas and technologies (Tapscott and Williams, 2006). This way, acting globally results in economic growth, makes companies more competitive and facilitates collaboration between people across many different businesses or business units (Tapscott and Williams, 2006).
For example, in 2003 Boeing[2] announced to replace its producer-supplier

[2]Boeing: `http://www.boeing.com`

relationship hierarchy with a network of technological partners from several different countries for designing and building their new "Boeing 787 Dreamliner" aircraft[3]. Having a global network decreases development costs for Boeing and reduces the risks of this large-scale project while collaboration with the partners is close.

3.1.2. Collective Intelligence on the Web

Although collective intelligence is not limited to the web, today's human collective intelligence is almost totally driven by data collected on the web. In this context it can be understood as some piece of shared intelligence that relies on online data generated by user activities. Examples of user activities on the web include creating websites, writing blog posts, submitting search requests, making online purchases, visiting web pages, watching videos online, listening to online music, user actions on a specific web page, etc. As described by Segaran, 2007, these user actions can be recorded, collected and used to automatically derive valuable information without having to ask the user explicit questions.

The two key aspects that enabled the development of collective intelligence on the web are first hyperlinking and second the switch from static HTML pages to dynamic Web 2.0 applications (see Section 3.3). On the one hand, as new web content is created by users, hyperlinks can be used to connect to already existing content (see also Jaindl, 2016). On the other hand, Web 2.0 technologies allow user collaboration in real-time.

The search engine Google [4], as depicted in Figure 3.2, is a well-known example of collective intelligence on the web. Like any other search engine, Google uses so-called *web crawlers* (also known as *spiders* or *Internet bots*) to automatically browse the web and analyze hundreds of millions of websites created by people from all over the world. Thereby, the crawlers extract relevant information from the visited websites and collect this data in order to build a search index. Whenever Google receives a search request from a user, the search engine has to look up the search index to find those web pages that best match the user's

[3]*Article of the Boeing 787 Dreamliner* 2017.

[4]Google Search Engine: `http://www.google.com`

input. This look-up process requires very powerful and optimized algorithms. Such algorithms aim to exploit the full potential of collective intelligence. In case of Google, the key factor of their success was their Page Rank algorithm which precomputes the ranking order of all web pages in advance based on their relative importance (Page et al., 1999). The relative importance of a single web page is measured by the quality and number of hyperlinks referring to it (see *Facts about Google and Competition* 2011). Thereby, a graph, referred to as the *webgraph*, is introduced that models web pages as nodes and hyperlinks as edges (see Example in Figure 3.3).

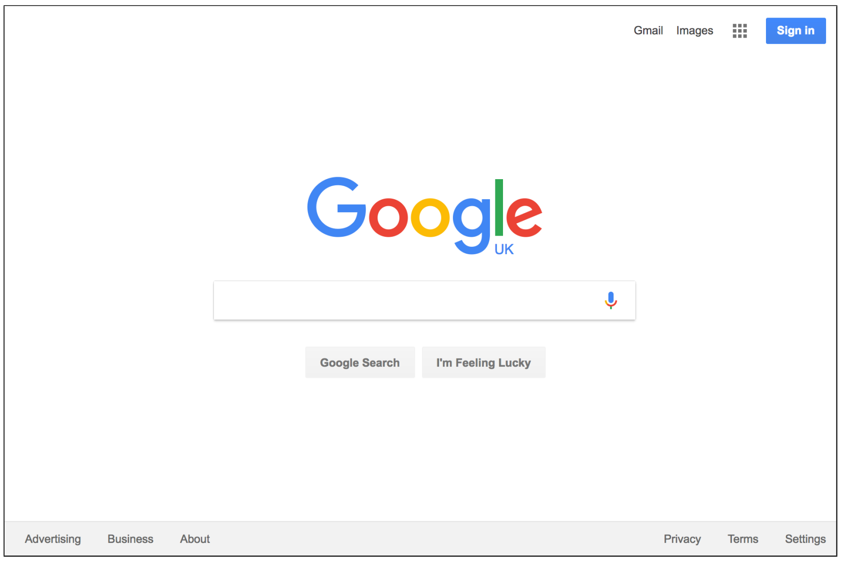


Figure 3.2.: Example of collective intelligence on the Internet (*The Google Search Engine* 2017)

The Page Rank algorithm then uses this graph as input in order to compute the importance of each single web page (i.e., node). Page et al., 1999 have shown that this algorithm works well on large scale graphs like the webgraph. For the sake of brevity, the mathematical details of the Page Rank algorithm are not explained in this thesis (see more Page et al., 1999).

On a very abstract level, both the search index of the web and its corresponding webgraph represent a large collection of knowledge which has been created by a human superorganism. Furthermore, the search engine provides an

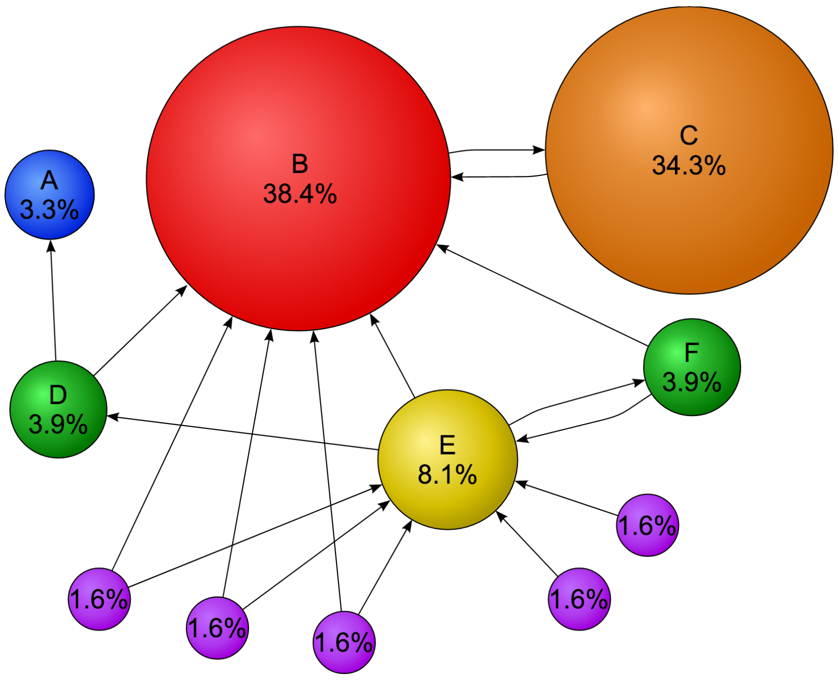


Figure 3.3.: Basic example of a simple "webgraph" (see *Article about "PageRank Algorithm"* 2017). Each node in the graph represents a single web page and is assigned an importance value determined by the Page Rank algorithm (Page et al., 1999). Page C has only one incoming link but a higher rank value than page E. This is because this single incoming link comes from page B, which has high importance, hence page C is of high importance too.
Source: `https://en.wikipedia.org/w/index.php?title=PageRank&oldid=774470009`

interface for users to directly communicate with the "brain" behind this kind of collective intelligence.

Another typical example of collective intelligence on the web is the online encyclopedia Wikipedia[5]. It is a collaborative *wiki* tool that allows users to freely create new and edit existing articles. Wiki tools are typical examples of collaborative software and are discussed in Section 3.3.2. Basic taxonomies on how to classify collaborative software are presented in the next Section 3.2.

[5]Wikipedia: `http://wikipedia.org`

3.2. Classification of Collaborative Software

Group work is proven to be an efficient form of cooperation in businesses and has also been successfully applied across different company boundaries many times in the past. Nowadays, this form of collaborative work is becoming more and more popular in companies where an increasing amount of work is done via software. In practice, group work is also often performed over spatial and temporal distances. In order to meet today's demands of cooperative group work, collaborative software is used.

The objective of *collaborative software*, also known as *groupware*[6], is to assist groups of users while they work on a common task or goal. Thereby, these systems provide an interface to a shared work environment for its user groups (Ellis, Gibbs, and Rein, 1991). Richman and Slovak, 1987 describe groupware as a revolutionary working concept that places the computer directly into the middle of the communications among a group of users. In other words, collaborative software systems are tools used to perform computer-supported cooperative work (Carstensen and Schmidt, 1999).

Computer-supported cooperative work (CSCW), also known as *computer-supported collaborative work*, is an umbrella term coined by Greif and Cashman at a workshop in 1984 (Sarin and Greif, 1984; Grudin, 1994). It defines a broad concept that constitutes an interdisciplinary research area for a variety of different disciplines including computer science, economics, media science, psychology, sociology, and many other disciplines. Due to its large research spectrum, a detailed description and analysis of CSCW would be beyond the scope of this Master's thesis. However, a brief introduction of the concept is given.

Basically, CSCW addresses cooperative group work and collaboration. Furthermore, it involves all information and communication technologies that support group work as well as evaluation methods used to assess them (Ellis, Gibbs, and Rein, 1991). Moreover, CSCW tries to answer the arising question of how computer systems can coordinate and support collaborative activities (Carstensen and Schmidt, 1999). Hence, a groupware system can be seen as a CSCW tool aiming to improve the productivity of a user group. One

[6]Groupware is a portmanteau term that comprises the words *group* and *software*.

important thing is that the decision of which groupware system to choose needs to be made very carefully. Basically, this selection mainly depends on the context and individual requirements of the user group, and the collaborative tasks that have to be performed by that group. For that reason, many different taxonomies exist in order to classify groupware. Two main classification approaches are briefly explained below.

3.2.1. Time/Space Classification of Groupware

Based on the context of a collaborative software system's use and the notions of time and space, groupware can be classified into four categories (Johansen, 1988; Ellis, Gibbs, and Rein, 1991; Baecker, 1995). Figure 3.4 shows a matrix which is attributed to Johansen, 1988.

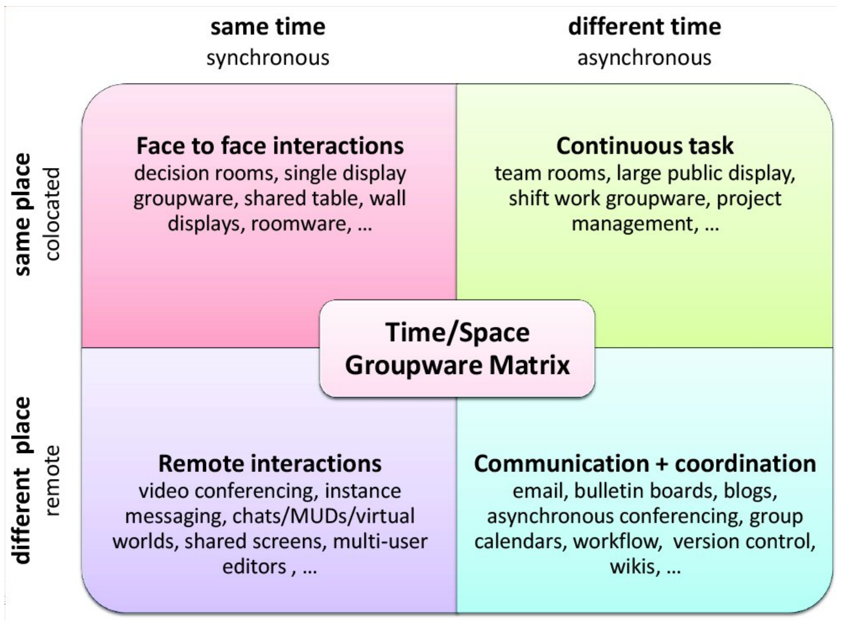

same time
synchronous
different time
asynchronous
same place
colocated
different place
remote
Face to face interactions
decision rooms, single display groupware, shared table, wall displays, roomware, ...
Continuous task
team rooms, large public display, shift work groupware, project management, ...
Time/Space Groupware Matrix
Remote interactions
video conferencing, instance messaging, chats/MUDs/virtual worlds, shared screens, multi-user editors , ...
Communication + coordination
email, bulletin boards, blogs, asynchronous conferencing, group calendars, workflow, version control, wikis, ...

Figure 3.4.: Time/Space Groupware Matrix (Johansen, 1988)

Source: https://en.wikipedia.org/w/index.php?title=Computer-supported_cooperative_work&oldid=765868253

1. **Same Place and Same Time:**
   The upper left quadrant of the matrix describes groupware that lets users of a working group interact face to face with the system. Thereby, the collaborative work takes place in the same room (i.e. same place) at some specific moment in time (i.e. same time). Examples of face-to-face groupware include meeting room systems such as digital whiteboards, wall displays, shared tables, single display groupware, group decision support systems, roomware, electronic meeting systems, and many others.

2. **Same Place and Different Times:**
   A collaborative software system which falls into the upper right cell of the matrix allows its users to work on an ongoing task in the same room but at different times. Typical examples of groupware for such continuous tasks are large public displays, team rooms, and shift work

groupware.

3. **Different Places and Same Time:**
   The lower left quadrant of the matrix represents groupware which allows users to interact remotely with the system. Examples include video conference solutions, electronic meeting software, real-time groupware, instant messaging groupware (SMS, chat, etc.), and many others.

4. **Different Places and Different Times:**
   Groupware which falls into the lower right group of the matrix supports coordinated communication. Typical examples are email, version control systems, wikis, blogs, group calendars, and asynchronous conferencing solutions.

In summary, according to Ellis, Gibbs, and Rein, 1991, a smart collaborative software system should fit the different needs of all four matrix cells. To name an example, Google's online office suite *"G Suite"*[7] (which includes applications such as *Google Docs*, *Sheets*, and *Slides*) is able to switch between online (real-time mode with multiple users at same time) and offline (non-real-time mode with only one active user) editing while the user interface and basic functionality remains the same.

As of today, many features of Catrobat's community website, an online collaborative software system for sharing Catrobat programs (see Section 5.4), fall into the lower right quadrant of the time/space matrix shown in Figure 3.4.

3.2.2. The 3C Model

The *3C model* was first coined by Teufel et al., 1995 and up to now it has also appeared in many other books (e.g. see Borghoff and Schlichter, 2000). Thereby, *communication*, *coordination* and *cooperation* define the three C's of the model's name and represent the three centers of the triangle's corners shown in Figure 3.5. Each of these three terms expresses the degree of intensity of collaboration within a group (Teufel et al., 1995):

[7]Google's Office Suite: `http://gsuite.google.com`

- *Coordination* expresses the coordination of task-related activities.
- *Cooperation* involves solving common tasks together.
- *Communication* describes a reliable and sufficiently fast exchange of information between the users of a group.

Usually, groupware implements a variety of different functions and can therefore not be clearly assigned to one these three C's. Consequently, groupware is classified depending on whether it primarily supports *communication*, *cooperation* or *coordination*. To phrase it differently, depending on the extent to which the functional areas are supported by a collaborative software system, the system can be located at different positions within the triangle. Furthermore, the different groupware systems can be grouped into four (overlapping) *system classes* (see ellipses in Figure 3.5):

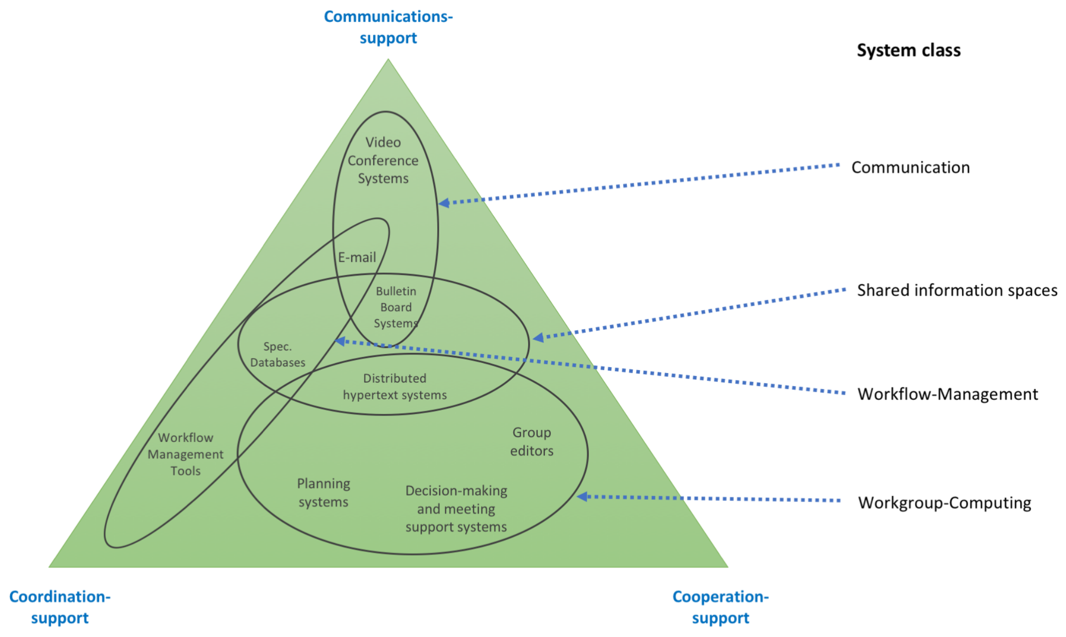


Figure 3.5.: The 3C classification model (Teufel et al., 1995)

- *Communication class:*
  The main task of a communication system is to allow the explicit exchange of information between different communication partners and to automatically bridge spatial distances and temporal differences. Typical examples are email systems, bulletin board systems, and video conferencing systems.

- *Shared Information Spaces class:*
  This class provides a shared information space for a user group in which relevant information is maintained, modified and stored over a long period of time. Thereby, the access to this information is protected with appropriate access mechanisms. The shared information spaces class includes distributed hypertext systems, bulletin board systems and special databases whose data can be fetched simultaneously by different users.

- *Workflow-Management class:*
  Workflow management encompasses all tasks that must be performed during the modeling, execution, control and simulation of workflows. Basically, workflows are organization-wide work processes that involve a large number of actors. Typical examples of the workflow-management class are email systems, workflow management tools, and special database systems.

- *Workgroup-Computing class:*
  Workgroup computing systems aim to support the co-operation of individual users who work in teams and have to solve tasks which are hard to divide and structure. In other words, groupware that falls into this category typically provides functionality to users in order to collaboratively work together on a common task. This class includes, for example, planning systems such as scheduling systems, group editors, and decision-making and meeting support systems.

It is obvious that the 3C model provides a clear overview of various forms of groupware. Moreover, the aforementioned system classes completely cover the whole area of known groupware applications. However, similar to the time-space approach (see Section 3.2.1), the 3C model can not completely avoid overlaps.

In terms of Catrobat's community website (see Section 5.4), considering the 3C model, there are many features such as *remixing* (see Section 3.4) which fall into different system classes. However, the entire system itself as a distributed website, also referred to as distributed hypertext system, would therefore most likely be placed rather into the middle of the triangle in Figure 3.5.

3.3. Examples of Collaborative Software

Due to the increasing importance of team work, a large number of software vendors for groupware and a huge variety of collaborative software applications already exists today. Typical examples of modern groupware applications include *email systems*, *video conferencing software*, *workflow systems*, *distributed websites*, *group calendars*, *version control systems*, and many others. Since the beginning of the *Web 2.0* era in 2003, which provided the foundation for the development of interactive and collaborative web applications, an increasing number of collaborative web applications have been developed. Today, these *Web 2.0 applications*, also known as *social software*, represent the majority of collaborative software applications. They can be directly run in web browsers without any software installations and enable collaboration among different users over the Internet. As a result, the user not only consumes available content, but also produces new content as a so-called "prosumer" (Fuchs, 2011, Toffler, 1980, p. 267). To limit the scope of this thesis, only a few representative examples of web 2.0 applications will be presented in more detail now.

3.3.1. Blogs

Blogs, also known as weblogs, are publicly accessible web diaries or journals. Basically, an online blog is a collection of many blog entries, also referred to as *blog posts*. Blog posts are written by an author, the *blogger*, and usually contain discussions or individual thoughts about specific threads or topics. The term *blogosphere* describes the entire collection of all weblogs and their interconnections on the web.

Sometimes a blog contains posts created by many bloggers or same blog posts of the blog have been edited by different bloggers. Such blogs are called *collaborative blogs* or *group blogs* and are an example of a collaborative software system. In particular, collaborative blogs have become more popular over the last several years. For example, one of the largest blog providers, Tumblr[8], supports group blogs.

Another growing type of blogs worth mentioning are *microblogs*. In contrast to the typically verbose posts written in normal blogs, posts of microblogs are very short. For example, Twitter, one of the most popular providers of microblog services, allows only 140 characters per message.

3.3.2. Wikis

A *wiki*[9] is a collaborative website made up of multiple wiki entries, which are article pages. The visitors of a wiki can read and directly edit these entries from their web browser. Different articles can be connected by using hyperlinks which have to be embedded into the article's text. Usually, authors elaborate texts, which may be supplemented with images, animations, or any other type of media. This way, articles written by one author can be extended or modified by other users. Wikis are used to collect experience, knowledge, and information, or to phrase it differently, wikis are collaborative software tools that strive to take advantage of collective intelligence (see Section 3.1). The articles of a wiki system are the result of collaborative work which has been made possible by allowing users to freely edit and reuse existing content (Richardson, 2006, pp. 55-58).

The most popular example of a wiki is *Wikipedia*. It is the largest encyclopedia on the web and relies entirely on user contributions. Wikipedia is based on the MediaWiki[10] system, which is a free open-source software project. Catrobat also provides a wiki[11] for its users, which is based on MediaWiki. At the time of writing this thesis, the wiki is used only internally. However, it is intended to allow users to write and edit tutorials, and to describe components of the Catrobat programming language such as bricks, scripts, etc. (see Section 5.2) in the future.

[8]Tumblr: `http://www.tumblr.com`

[9]The term *wiki* comes from the Hawaiian word *wikiwiki* and means *fast*.

[10]MediaWiki: `http://www.mediawiki.org`

[11]Catrobat Wiki: `http://wiki.catrob.at`

3.3.3. Social Networks

The most popular example of social applications are *social networks* such as Facebook, Twitter, Google+, LinkedIn, or XING. The basic idea of a social networking service is to group related users into clusters which are created based on the relationship between the users, e.g., friendships, acquaintances, workmates, or individuals who share similar interests. Thereby, each user creates a user profile by answering a series of questions once she or he joins the social network (Boyd and Ellison, 2007). The profile stores this personal data about the user in a structured way. Such personal data typically includes user information such as name, age, location, interests, and a short description about the user (Boyd and Ellison, 2007). One of the key characteristics of social networks is that they allow instant communication and exchange of multimedia content between users over any arbitrary distance (Jaindl, 2016).

Apart from typical social networking platforms, there are many other types of social applications such as online communities, social news websites, web forums, instant messaging platforms or media sharing services. To limit the scope of this Master's thesis, these examples are not described in detail. However, media sharing services are relevant for this thesis and are hence briefly discussed below.

3.3.4. Media Sharing Services

Media sharing services, also referred to as *content sharing services*, are platforms that allow users to upload and share media files like music, photos, videos or any other type of media content. Usually, a provider of such a service, hosts these media files on a central server or on a cloud network. Sometimes, such media sharing services are part of social networking platforms (Gadea et al., 2011). For example, the social networking sites Facebook or Twitter allow their users to upload photos and videos. The largest sharing platform on the web is YouTube, a video hosting service. Other well-known examples are Vimeo, Netflix, Spotify, Dropbox, Google Drive, Apple iTunes Store, Apple Music, Google Play Music, and Wikipedia (see also Section 3.3.2).

A special type of a media sharing service is a controlled digital distribution service. Such a controlled service specializes in hosting security sensitive or high-quality content such as executable software and buyable content. In order to guarantee reliable quality and security of the provided content, these platforms only allow a small set of creators (e.g., mobile app developers) or employed editors to upload and maintain digital content. Often, after content gets uploaded by a user, it is also automatically tested and checked with special tools again. Typical examples include the Google Play Store and the Apple App Store. Both are so-called *app store platforms* that host applications which were programmed for mobile devices (also referred to as *mobile apps*) by software companies or private software developers.

In contrast to such commercial app stores, there also exist free e-learning platforms for sharing *programmable media content* such as Scratch (Monroy-Hernández, 2007) and Catrobat's community website (see Section 5.4) on the Internet. In the case of Catrobat, its community website represents a sharing platform quite similar to Scratch. Both platforms let their users easily upload and share programmable media projects.

3.4. Collaborative Learning and Remixing

Collaborative learning has become an emerging elearning concept in online education over the last several years (Dalsgaard, 2006). In general, collaborative learning is about learning together in groups and can either happen *offline* (e.g., in classrooms of schools, at the workplace, etc.) or *online* (also known as *collaborative online learning* or *collaborative e-learning*). The main goal of collaborative learning is to increase the learning outcome for each member of a group by combining the unique strengths of each individual. Moreover, collaborative learning can be seen as a counter-concept to contests or competitions. While members of a group try to beat each other in competitions, collaborative learning aims to find a consensus through group collaboration (Laal and Ghodsi, 2012). Further, collaborative learning has many other advantages over competitive work or work performed by individuals. It is a supportive learning process that may result in increased productivity, higher achievement, and more social competence (Laal and Ghodsi, 2012). Collaborative learning activities include group projects, solving common problems or creating a product together, cooperative writing, completing a common task, etc.

Due to its increasing importance, and to limit the scope of topics to those relevant to this thesis, the remainder of this section only focuses on collaborative *online* learning. Actually, collaborative online learning is a special form of *computer-supported cooperative work* which has already been discussed in Section 3.2. Such collaborative work can be performed by using *collaborative software systems*. In an online educational context, these systems are also referred to as *collaborative e-learning systems* or *social (e-learning) software*. Basically, a collaborative e-learning system is an online learning solution that supports its users (e.g., students and teachers) by letting them (intuitively) interact with each other and allowing them to share and exchange learned knowledge (Marusteri et al., 2015).

While traditional non-collaborative e-learning approaches such as *learning management systems*, in which learning is organized as courses, have some limitations (e.g., learners cannot individually control the learning process, interaction is limited, etc.), collaborative e-learning views the learning process as a collaborative, problem-based, and self-governed social process (Dalsgaard, 2006; Du et al., 2013). Typical examples of groupware used in collaborative e-learning are wikis, blogs, social networks, social bookmarking websites, and many others (see Dalsgaard, 2006).

In the context of the project of this thesis (see Chapter 6), Catrobat's community website, described in Section 5.4, represents a collaborative e-learning platform which enables its users to develop and work on joint programmable media projects. This e-learning platform tries to facilitate the collaborative learning process by introducing a new powerful e-learning technique called *remixing* which is discussed in more detail below.

Remixing: The basic idea of remixing in Catrobat consists in creating a refined or extended version of an existing media project. The main advantage of remixing is the reuse of existing work. Thereby, users (called "remixers") download existing projects created by other users from Catrobat's community website and modify them. Once changes are made to the project, a copy of the original (i.e. existing) project is automatically created in the background in order to avoid conflicts. In other words, this copy is now a remix of the original project. After all modifications are finished, the "remixer" can upload the copy to share it on the community website. Later on, other "remixers" may create remixes of this remixed project. This cycle of remixing continues as soon as other users create other remixes based on these new remixed remixes.

It is important to be aware that Scratch follows a very similar remixing approach; it acted as a source of inspiration for Catrobat. Dasgupta et al., 2016, show how remixing in Scratch encourages novice users to learn how to program and demonstrate that highly active remixers usually achieve more advanced and extended programming skills through remixing.

In conclusion, this chapter discussed the fundamental concepts of collaborative software. At the beginning of this chapter, the underlying relationship between collaborative software and collective intelligence was described and different classification approaches were presented. Subsequently, an overview of examples of collaborative applications was given. Finally, the focus moved towards collaborative learning and remixing. Remixing is a key feature for collaborative learning in online environments such as the Catrobat community platform. Remixing lets users build upon work from others, thereby enhancing the collaborative learning process. This learning process can be further improved, supported, and promoted by using recommender systems. Chapter 4 provides an overview of recommender systems and gives the necessary background knowledge needed for the following practical part of this thesis.

4. Recommender Systems

This chapter gives an overview of basic concepts and ideas for Recommender Systems, which form the basis for the next chapters of this thesis. Moreover, common issues (such as the "Cold Start Problem"), which frequently occur in the context of recommender systems, are briefly discussed. Finally, the chapter concludes with the presentation of various methods for evaluating the performance of such a system.

4.1. Motivation and Principle of Recommender Systems

Recommender systems are a very popular research area in the field of computer science. Every year many international conferences about recommender systems take place in different parts of the world and are supported by many big companies. One of the largest scientific conferences is the well-known ACM RecSys conference[1].

According to Ricci et al., 2010; Mahmood and Ricci, 2009; Burke, 2007; Resnick and Varian, 1997, a recommender system is a combination of several software techniques and software tools which makes suggestions for items that are relevant to a user. The so-called "items" represent kinds of objects, e.g., buyable goods, news articles, music albums, holiday trips, or any other arbitrary kind of an intangible product. In the context of this thesis, "items" are always programs created by users on mobile devices, including smartphones and tablets, which are then uploaded to the Catrobat community website (see Chapter 5).

[1]The ACM Conference Series on Recommender Systems: `http://recsys.acm.org`

Usually, a recommender system mainly focuses on recommending a specific type of item as described by Ricci et al., 2010. Many e-commerce platforms (e.g. web shops) and online collaborative software systems (such as Facebook, Youtube, Google Play Store or Catrobat's community website) often have to deal with a huge number of items and a fast growing number of users.

Moreover, many e-commerce systems such as Amazon or Netflix, tend to reveal only a small number of very popular products (e.g. top 100 most purchased books) and a majority of less favoured items (i.e. niche products). This is known as the "long tail" phenomenon, which can be approximated by using the Pareto principle (Brynjolfsson, Y. (Hu, and Simester, 2011; Yin et al., 2012).

Figure 4.1 shows the curve of a typical long tailed sorted popularity distribution of products. It is obvious that the overall popularity is mainly driven only by the most popular items located in the "head" (orange-colored area). However, even though the contribution of a single niche product to the total revenue may not be significant, the vast amount of these items (i.e. the complete blue-colored area) can have an impact on the total revenue in aggregate as explained by Anderson, 2006. Due to the high popularity of the items in the "head", these products would consequently be suggested more often by a recommender system that does not pay close attention to niche items. Therefore, exploiting the full potential of these niche items is an important optimization task for a recommender system in order to increase diversity of recommendations and, consequently, the number of sales.

The great variety of different interests is another important aspect that has to be taken into account, as each user represents an individual person with their own preferences. Moreover, many users also lack competence or sufficient personal experience to evaluate the overwhelming set of items (Ricci et al., 2010; Resnick and Varian, 1997). Thus, automatically searching for and finding the right items that are of interest for a single user is a vital challenge for such platforms. This is exactly where recommender systems now come into play in order to tackle this kind of search task.

The main goal of a recommender system is to separate and recommend only those items which are relevant to the respective user from an otherwise overwhelming set of items (Resnick and Varian, 1997). In other words, these suggested items are expected to be tailored to the preferences and personal

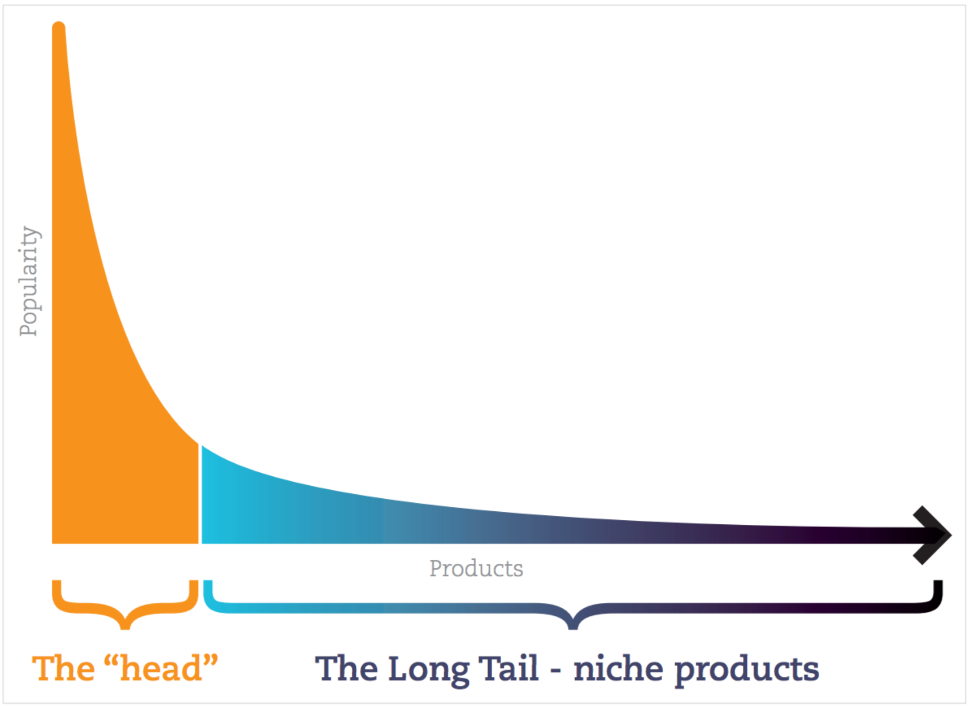


Figure 4.1.: The "long tail" phenomenon
Source: `http://blog.softcube.com/?p=17`

tastes of the individual user. Basically, in order to make such user-specific decisions, a recommender system must: (1) identify the user; (2) collect data about the user's preferences over a sufficiently long period of time; and (3) structure the information to finally create a user profile.

More precisely, according to Isinkaye, Folajimi, and Ojokoh, 2015, the recommendation process consists of three different phases. Figure 4.2 illustrates the three basic steps of the recommendation process.

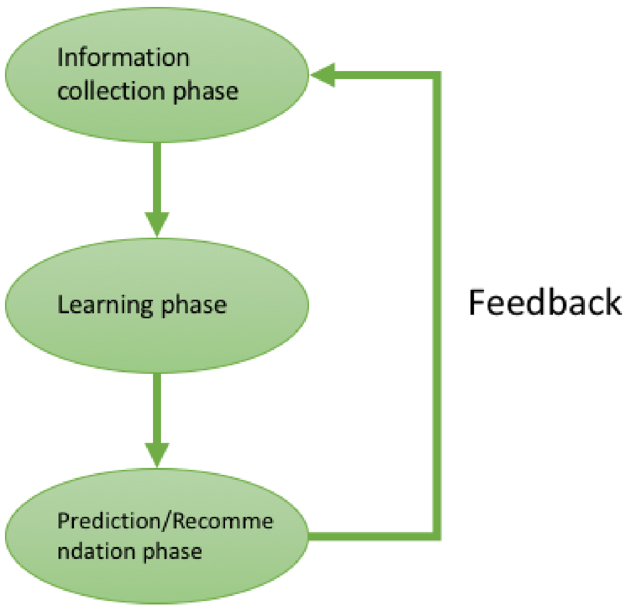


Figure 4.2.: Phases of the recommendation process (Isinkaye, Folajimi, and Ojokoh, 2015)

The three phases of the recommendation process are:

i. **Information Collection Phase:**
This is the initial phase of the recommendation process. During this phase relevant information about the user is collected and a user profile or model is created for the prediction phase. User data includes user behavior, relevant characteristics about the user, and the content of the resources the user has accessed. There are two different types of input data (explicit and implicit feedback) which are discussed in Section 4.2.

ii. **Learning Phase:**
Once enough data has been collected, appropriate learning algorithms can be applied. These algorithms remove non-relevant data and try to extract useful features from the existing feedback.

iii. **Prediction/Recommendation Phase:**
In the final prediction phase (also known as recommendation phase), those items which the user most likely prefers are predicted and suggested. The suggestions are either based on the data gathered in the information collection phase (model- or memory-based, see Section 4.4.2) or based on observed data from past user activities.

4.2. Input Sources

The overall performance of the recommender process heavily relies on the quantity and quality of its input data. Defined below are explicit and implicit feedback, the two fundamental types of input sources.

4.2.1. Explicit Feedback

Explicit feedback is obtained from explicit data such as ratings, personal interests, or user preferences. Since it directly reflects the user's opinion, it is of high-quality. Therefore, the majority of the recommender system's literature focuses on examining explicit feedback data. However, in some scenarios no explicit data is available, or it cannot be collected at all, e.g., due to system limitations. In such cases, implicit feedback is often used for recommendation purposes instead.

4.2.2. Implicit Feedback

Contrary to explicit data, implicit feedback is extracted from observed user behavior and then used to make indirect conclusions about possible preferences of the user (Oard and Kim, 1998). A few examples for implicit feedback are the user's purchase history, items visited by the user, the duration the user spends viewing individual web pages, search patterns, or even mouse clicks and mouse movements.

Although implicit information can be an appropriate alternative input source for recommender systems, in comparison to explicit data, it has some drawbacks as described by Y. Hu, Koren, and Volinsky, 2008. These drawbacks, as well as some benefits, are briefly discussed below.

4.2.3. Comparison between Explicit and Implicit Feedback

Explicit feedback requires the willingness of the users to disclose their preferences. However, not every user wants to supply a moderate amount of private information about themselves, if any at all. Consequently, explicit data such as rating data is often very rare and sparse. On the one hand, this can be seen as a benefit as well, since it makes the recommendation process more transparent, resulting in an increased recommendation quality and more confident suggestions (Isinkaye, Folajimi, and Ojokoh, 2015; Buder and Schwind, 2012). On the other hand, no explicit user input is needed in order to record implicit signals and therefore this kind of data can be gathered automatically. Hence, the amount of implicit data can be huge, which can have a positive impact on the recommendation quality.

Implicit data represents positive-only feedback, since negative feedback can not be correctly inferred from past user-behavior observations. For example, just because the user has not viewed a certain item does not necessarily mean that she or he dislikes it. Perhaps the reason is a different one, e.g., that she or he did not have enough time to look at it or does not know about the item at all. Another downside is that implicit feedback may be noisy, due to the fact that user behavior is passively tracked. The resulting inferred conclusions about user preferences can only include rough estimations, which may be less accurate than those based off of purposeful selection. Moreover, implicit feedback includes confidence users have placed in certain items but is unable to describe the level of preference. On the contrary, in the case of explicit feedback, users are able to express how much they like or dislike an item, e.g., by simply choosing a different number of stars in a star rating system. Finally, the evaluation of a recommender system using implicit data differs from those using explicit feedback in such a way that alternative measures are needed. Due to these drawbacks, it is necessary to interpret implicit data very carefully.

4.3. User-Specific vs. Non-Personalized Recommendations

Based on the collected user data, a recommender system is able to make user-specific suggestions by applying several recommendation techniques (see Section 4.4). The more data the system collects about its users, the better the system knows them. Therefore, user profiles containing an appropriate amount of user data are essential for making good user-specific recommendations. To phrase it differently, user-specific suggestions are based on the past behavior of the respective user. This past behavior can also be used to calculate similarities between users or items (see Section 4.4.2).

In sharp contrast to this method of compiling past behaviors to formulate new suggestions, there also exist very specific application areas of recommender systems where no user data can be collected at all. Various examples include recommendations featured in newspapers, magazines, or on TV shows. In this case, recommender systems are forced to propose only non-personalized recommendations. They can either be based on automatically aggregated data (e.g. top selling books, most trending movies, etc.) or manually created by editors (e.g. a list of featured products). Non-personalized advices are general item-references that are independent of the user and are based on the average feedback of other users (Schafer, J. Konstan, and J. Riedl, 1999). As there is no relation to the user, such non-personalized suggestions are typically very straight-forward to create and are therefore rarely considered by recommender systems research as described by Ricci et al., 2010.

However, in practice, they might be quite useful and are needed in certain situations where no user data is available. For example, former research by Tsuji et al., 2012, shows that Amazon's state-of-the-art non-personalized book recommendations can outperform evaluation results of recommendations based on collaborative filtering methods.

4.4. Recommendation Methods

Recommender Systems can be broadly classified into the three categories: content-based, collaborative filtering and hybrid systems. In addition, there are also some other alternative approaches, such as knowledge-based (Burke, 2000; Felfernig et al., 2014), time-aware (Campos, Diez, and Cantador, 2014), context-based (Adomavicius and Tuzhilin, 2008) or demographic (Wang, Chan, and Ngai, 2012) recommender systems. Since these alternative approaches would be outside the scope of this thesis, they are not discussed in more detail.

4.4.1. Content-based Recommenders

The content-based recommendation technique is usually best applied to the recommendation of text documents, such as newspaper articles, publications, newsgroup messages, web pages, etc. (Pazzani and Billsus, 1997; Meteren and Someren, 2000; Pazzani and Billsus, 2007). It can also be successfully used in any other application domain where items are characterized by attributes (e.g. keywords, tags, metadata, title, author, words of description, etc.).

This technique aims to suggest items to users that are similar to the ones they preferred in the past. Figure 4.3 shows the basic principle of a content-based recommender. The underlying algorithm extracts feature values from the content of those items the user has previously highly rated or liked. In order to do so, a user profile needs to be created. The user profile describes what kind of items appeal to the user (Pazzani and Billsus, 2007) and can be understood as a model explaining the user's interests based on the features of their rated items.

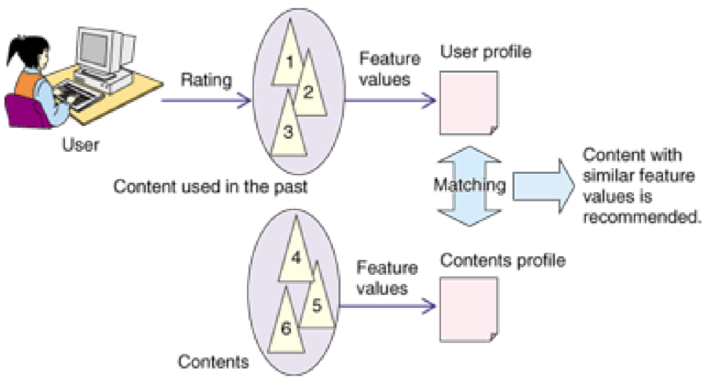


Figure 4.3.: Basic principle of a content-based recommender
Source: https://www.ntt-review.jp/archive_html/200804/images/le1_fig02.gif

The profile is gained by using an appropriate learning model such as, e.g., Nearest Neighbour, decision trees, Bayesian Classifiers, neural networks, clustering, etc. (Pazzani and Billsus, 1997). In other words, the problem is treated as a classification problem where the model is fitted to training data by the algorithm. Since there is no general rule of thumb, deciding which learning model to choose depends on the concrete application's use case as well as on the given context. After the user profile is computed, content-features of other items are then used to find similar ones the user has not rated yet. More formally expressed, the correlation between the preferences of a user's profile and the item's features is analyzed. Finally, the most similar items are recommended to the user. Such content-based suggestions in terms of movies can be, e.g., other movies with the same actor, director, or genre as the ones the user has previously liked.

4.4.2. Collaborative Filtering

Since content-based prediction techniques can only be applied in some domains where items are associated with a decent amount of content, research focuses primarily on collaborative filtering methods (Ekstrand, J. T. Riedl, and J. A. Konstan, 2011; Goldberg et al., 1992; J. A. Konstan et al., 1997). The authors of *Tapestry* (Goldberg et al., 1992; Resnick, Iacovou, et al., 1994), the first online recommender system, introduced the term *collaborative filtering* as a concept based on social collaboration between users. Tapestry was designed to recommend documents from newsgroups in order to help users to find relevant articles in a large collection of documents.

Basically, a recommender system based on collaborative filtering represents nothing else than a collaborative software system which aims to exploit the full potential of collective intelligence (see Chapter 3). Thereby, collaborative filtering can be easily integrated into all possible forms of applications, which are able to observe and capture relationships between users. This includes explicit feedback (e.g. user ratings or likes) as well as implicit information (e.g. purchase history of users). The collaborative filtering approach has become the most popular recommendation method since it was promoted by Resnick and Varian, 1997. Nowadays, many big internet companies like Netflix, Facebook, YouTube, and Twitter make use of this approach to recommend unseen items (e.g. movies, friends, web pages) to their users.

In short, the aim of collaborative filtering is to model the concept of word-of-mouth suggestions between humans. In everyday life, people tend to base their decisions on recommendations from other persons they know and trust, apart from their own experiences. Thereby, the algorithm follows this concept by introducing a rating matrix R (Definition 4.1), also known as user-item matrix, describing the user preferences for items.

$$R = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ r_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m1} & r_{m2} & \cdots & r_{mn} \end{pmatrix} \tag{4.1}$$

R is a $m \times n$ matrix, where m denotes the total number of users and n the total number of items. We define the item i as an element in the set of all items I (formally, $i \in I$) and the user u as an element of the set of all users U (formally, $u \in U$). The individual element $r_{u,i}$ of the $m \times n$ matrix R, represents the rating of the item i by user u. For simplicity reasons, we make the assumption that only non-negative integer values for ratings (e.g. 5 star rating) are allowed (i.e. $r_{u,i} \in \mathbb{N}_0^n$) and the rating value of the corresponding matrix element is always zero for items the user has not rated yet.

Figure 4.4 gives an overview of the collaborative filtering process as described by Yao and Cai, 2015. The goal of the algorithm is to predict an item-rating r_{u_a,i_j} for the active user u_a and a specified item i_j which has not yet been rated by the active user. In the picture the item-rating r_{u_a,i_j} represents the black cell within the rating matrix. Depending on the concrete collaborative filtering method the algorithm either uses the user-vector (highlighted row) or item-vector (highlighted column) in order to compute and return the rating value for item i_j. Another task of the algorithm can be to recommend a so-called "top-N list" of unrated items to the active user u_a. In this case, the prediction step is repeated for every other unrated item in the active user's row. Finally, no more cells in the row of the active user are empty and then the algorithm returns a sorted list of unrated items containing the highest prediction values. These are the "top-N recommendations".

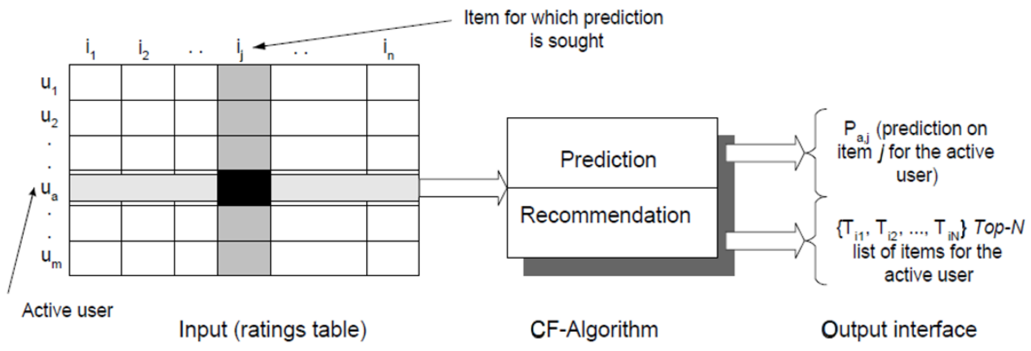

Item for which prediction is sought
i_1
i_2
i_j
i_n
u_1
u_2
u_a
u_m
Active user
Input (ratings table)
Prediction
Recommendation
CF-Algorithm
$P_{a,j}$ (prediction on item j for the active user)
$\{T_{i1}, T_{i2}, ..., T_{iN}\}$ Top-N list of items for the active user
Output interface

Figure 4.4.: The collaborative filtering process as described by Yao and Cai, 2015
Source: https://www.semanticscholar.org/paper/User-Based-and-Item-Based-Collaborative-Filtering-Yao-Cai/c4b2363ba7c15c67a2c6b2d079e325f1a33d6b75

Basically, there exist two different collaborative filtering methods which are described below.

Memory-based Collaborative Filtering

The memory-based (or neighbourhood-based) approach generates recommendations by heuristically analyzing the rating matrix (Definition 4.1). It aims to find neighbour users that share similar tastes with the active user. Memory-based systems use various algorithms to group cohesive preferences of different neighbour users and then generate top-N recommendations or predictions for the user. There are two types of approaches which are described as follows.

User-based: User based filtering is the simplest implementation of this method (Schafer, Frankowski, et al., 2007). It suggests items based on user-to-user correlation. This way, users receive recommendations of items that other like-minded users have previously expressed interest in ("rating history"). To phrase it differently, the main idea is to find those users, who share similar taste with the active user, and predict ratings for all items the active user has not rated yet. The prediction is achieved by calculating weighted averages of the ratings of all similar users. Subsequently, the top-N predicted items are recommended to the active user.

Under the hood the algorithm computes the similarity value between the active user's matrix row and the row of another user. This process is repeated for all other users in order to obtain all similarity values between the active user and the other users.

More formally expressed, an item-rating $\hat{r}_{u,i}$ for the active user u and an unrated item i is predicted (Yao and Cai, 2015; Zhao and Shang, 2010):

- In the first step, every individual similarity factor $sim_{u,v}$ between the active user u and the other user v is computed for all other users. The similarity factor $sim_{u,v}$ is determined by a similarity function s (see definition and examples in Section 4.4.2) which expresses how similar the active user u is to the other person v.

- Given these similarity values, the predicted value $\hat{r}_{u,i}$ (see Formula[2] 4.2) is then calculated by iterating over each[3] other similar user $v \in V$ and summing up the products of the similarity factor $sim_{u,v}$ (between the active user u and the similar user v) and his/her rating value $r_{v,i}$[4]. Finally, the sum is divided by the absolute sum of all same similarities in order to correctly consider dissimilarities (i.e. negative similarities) between items as well.

$$\hat{r}_{u,i} = \frac{\sum\limits_{v \in V} (sim_{u,v} * r_{v,i})}{\sum\limits_{v \in V} \mid sim_{u,v} \mid} \tag{4.2}$$

Formula 4.2 defines a basic example of an aggregation function for user-based predictions. It weights the ratings of other similar users with their similarity factor. However, different users tend to have other "baselines" around which their ratings may vary (Garcin, Faltings, Jurca, et al., 2009), meaning their average rating of an item can differ. Therefore, a slightly adapted function (Formula 4.3) is commonly used which takes this aspect into account.

$$\hat{r}_{u,i} = \bar{r_u} + \frac{\sum\limits_{v \in V} (sim_{u,v} * (r_{v,i} - \bar{r_v}))}{\sum\limits_{v \in V} \mid sim_{u,v} \mid} \tag{4.3}$$

In practice, the application does not only need the prediction value for a single unrated item, but often explicitly requests a top-N recommendation list. In order to create such a list, the two steps shown above have to be repeated for all other unrated items. This then iteratively replaces all zero cells of the active user's row in the matrix with predicted values for the corresponding item. Thereafter, the unrated items can be sorted to a list of top-N recommendations according to their predicted values and presented to the active user.

[2]Note: $V \subset U \setminus \{u\} \leftrightarrow \forall v \in V : sim_{u,v} \neq 0$.

[3]In case there are too many similar users, this is often limited to k most similar users.

[4]If the similar user did not rate the item the product will yield zero, since $r_{v,i} = 0$.

Item-based: Item-based algorithms have become very popular since Amazon heavily promoted this approach (Linden, Smith, and York, 2003). For example, in the case of Amazon, as soon as a user places a product in the shopping basket, similar products are suggested, which have been purchased by other users who have also bought this product. In contrast to user-based filtering, the item-based approach recommends items based on item-to-item correlation, meaning the similarity between two items given the rating matrix are computed for each item-item pair.

The basic idea of the item-based approach is to find items, which have been rated similarly by users, and predict ratings for all items the active user has not rated yet. The prediction is achieved by calculating weighted averages of the ratings of all similar items. Afterwards, the top-N predicted items are recommended to the active user.

An item-rating $\hat{r}_{u,i}$ for the active user u and an unrated item i is predicted in the following way (Yao and Cai, 2015; Zhao and Shang, 2010):

- First, every similarity factor $sim_{i,j}$ between the unrated item i of the active user and another evaluated item j is computed for all other evaluated items of the active user. The similarity factor $sim_{i,j}$ is determined by the similarity function s (see Section 4.4.2) which expresses how similar the rated item i is to the other item j.

- Given these similarity values, the predicted value $\hat{r}_{u,i}$ (see Formula 4.4) is then calculated by iterating over each[5] other similar item $j \in S$ and summing up the products of the similarity factor $sim_{i,j}$ (between the unrated item i and the evaluated similar item j) and the active user's rating value $r_{v,j}$ for the similar item j. Finally, the sum is divided by the absolute sum of all same similarities in order to correctly consider dissimilarities between items as well.

Formula 4.4 gives a basic example of an aggregation function for item-based predictions, where S is defined as a set of items similar to item i. In practice, S is often very large and therefore reduced to the k most similar items of item j that has been rated by user u. According to J. Herlocker, J. A. Konstan,

[5]In case there are too many similar items, this should be limited to k most similar items.

and J. Riedl, 2002, setting the size of the neighbourhood k between 20 and 50 provides enough neighbours to average out extremes.

$$\hat{r}_{u,i} = \frac{\sum\limits_{j \in S} (sim_{i,j} * r_{u,j})}{\sum\limits_{j \in S} \mid sim_{i,j} \mid} \tag{4.4}$$

In order to recommend a top-N recommendation list to the active user, the two steps described above have to be repeated for all other unrated items.

User-based vs. Item-based: Figure 4.5 graphically illustrates the difference between user-based and item-based collaborative filtering with a basic example for binary, positive-only feedback. For instance, binary feedback can represent explicit user likes or implicit feedback such as the user purchased an item. In this example, we consider the binary feedback as explicit positive-only ratings. That is, all elements of a user vector are either 0 (i.e. user has not rated the item) or 1 (i.e. user likes the item). More formally expressed, $r_{u,i} \in \{0,1\}$.

In Figure 4.5 there are three users who like four different items. The directed black-colored lines indicate which person likes which item. For the sake of simplicity, we make the assumption that all likes are equally important, meaning we have binary ratings. The task is to recommend items to the last user based on the individual ratings, first by using user-based and then by using item-based collaborative filtering. As a measure of similarity, we use the Jaccard distance (see Formula 4.17), which is suitable for binary evaluations.

On the left side of the image a user-based example is shown. The topmost user denoted as u_1 prefers item 1 (i_1), item 2 (i_2), item 3 (i_3) and item 4 (i_4), i.e. $R_{u_1} = \{r_{u_1,i_1}, r_{u_1,i_2}, r_{u_1,i_3}, r_{u_1,i_4}\} = \{1,1,1,1\}$). The user in the middle has only rated item i_2 (i.e. $R_{u_2} = \{r_{u_2,i_1}, r_{u_2,i_2}, r_{u_2,i_3}, r_{u_2,i_4}\} = \{0,1,0,0\}$) and the bottommost user u_3 likes item i_3 (i.e. $R_{u_3} = \{r_{u_3,i_1}, r_{u_3,i_2}, r_{u_3,i_3}, r_{u_3,i_4}\} = \{0,0,1,0\}$). In order to keep this example simple, we assume that only items from the most like-minded person are recommended.

Using the Jaccard distance the taste-overlap between u_1 and u_3 is:

$$J(R_{u_1}, R_{u_3}) = \frac{\mid \{R_{u_1} \cap R_{u_3}\} \mid}{\mid \{R_{u_1} \cup R_{u_3}\} \mid} = \frac{\mid \{1,1,1,1\} \cap \{0,0,1,0\} \mid}{\mid \{1,1,1,1\} \cup \{0,0,1,0\} \mid} = \frac{1}{4} \tag{4.5}$$

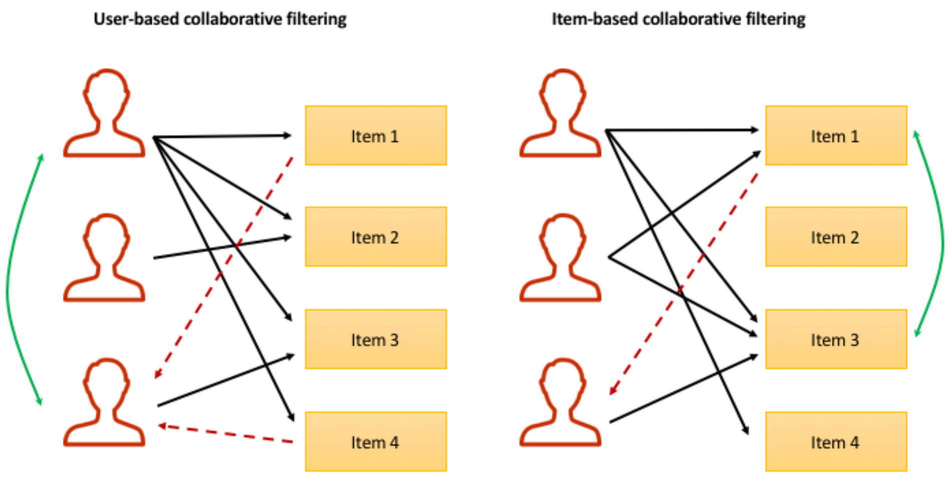

User-based collaborative filtering
Item-based collaborative filtering
Item 1
Item 2
Item 3
Item 4
Item 1
Item 2
Item 3
Item 4

Figure 4.5.: User-based vs. item-based collaborative filtering

Consequently, the user-similarity between u_2 and u_3 is:

$$J(R_{u_2}, R_{u_3}) = \frac{|\{R_{u_2} \cap R_{u_3}\}|}{|\{R_{u_2} \cup R_{u_3}\}|} = \frac{|\{0,1,0,0\} \cap \{0,0,1,0\}|}{|\{0,1,0,0\} \cup \{0,0,1,0\}|} = 0 \tag{4.6}$$

Since the taste-overlap between user u_1 and user u_3 ($\frac{1}{4}$) is greater than the overlap between u_2 and u_3 (0), the bottommost user u_3 will receive recommendations of all other items the topmost user u_1 has liked (see green link in Figure 4.5). These are item i_1 and item i_4 (indicated by a red-colored dashed line).

On the right side of the figure the item-based approach is visually expressed. In this case, the topmost user u_1 now prefers item i_1, item i_3 and item i_4, the user in the middle u_2 likes item i_1 and item i_3 and the bottommost user u_3 has only evaluated item i_3. For reasons of simplicity, we assume that only the most similar item is recommended.

In terms of items we now have the following four rating sets:

$$R_{i_1} = \{r_{u_1,i_1}, r_{u_2,i_1}, r_{u_3,i_1}, r_{u_4,i_1}\} = \{1,1,0\} \tag{4.7}$$

$$R_{i_2} = \{r_{u_1,i_2}, r_{u_2,i_2}, r_{u_3,i_2}, r_{u_4,i_2}\} = \{0,0,0\} \tag{4.8}$$

$$R_{i_3} = \{r_{u_1,i_3}, r_{u_2,i_3}, r_{u_3,i_3}, r_{u_4,i_3}\} = \{1,1,1\} \tag{4.9}$$

$$R_{i_4} = \{r_{u_1,i_4}, r_{u_2,i_4}, r_{u_3,i_4}, r_{u_4,i_4}\} = \{1,0,0\} \tag{4.10}$$

Because the last user has only evaluated item i_3, we only need to consider all item similarities between i_3 and the other items. By using the Jaccard measure for calculating the similarities to item i_3, we obtain the following values:

$$J(R_{i_3}, R_{i_1}) = \frac{|\{R_{i_3} \cap R_{i_1}\}|}{|\{R_{i_3} \cup R_{i_1}\}|} = \frac{|\{1,1,1\} \cap \{1,1,0\}|}{|\{1,1,1\} \cup \{1,1,0\}|} = \frac{2}{3} \tag{4.11}$$

$$J(R_{i_3}, R_{i_2}) = \frac{|\{R_{i_3} \cap R_{i_2}\}|}{|\{R_{i_3} \cup R_{i_2}\}|} = \frac{|\{1,1,1\} \cap \{0,0,0\}|}{|\{1,1,1\} \cup \{0,0,0\}|} = 0 \tag{4.12}$$

$$J(R_{i_3}, R_{i_4}) = \frac{|\{R_{i_3} \cap R_{i_4}\}|}{|\{R_{i_3} \cup R_{i_4}\}|} = \frac{|\{1,1,1\} \cap \{1,0,0\}|}{|\{1,1,1\} \cup \{1,0,0\}|} = \frac{1}{3} \tag{4.13}$$

Obviously, a quick inspection reveals that the maximal Jaccard-correlation between item i_3 and all other items is the similarity between i_3 and i_1, meaning that item i_1 is the most similar to i_3 (see green link in Figure 4.5). In other words, the item-pair (i_3, i_1) has been rated by a larger common user group than the item-pairs (i_3, i_2) and (i_3, i_4). Thus, item i_1 is recommended to user u_3.

Comparison: Although item-based collaborative filtering is more popular than user-based collaborative filtering, the decision of which to choose greatly depends on the number of items and users the system has (J. L. Herlocker et al., 2004). On most websites (e.g. Amazon, Netflix, etc.) the number of users is much higher than the number of items. Under these conditions item-based collaborative filtering will expectably perform better. This is due to the fact that the vector of an item contains the ratings from all users that have rated this item regardless of whether they are similar to the active user u or not. Contrary to user-based filtering, the calculation for an item-based recommendation takes only item-vectors (instead of user-vectors) into account. Hence, the item-based approach finds similar items even if the user is new and has not rated any items before (see Section 4.4.3). Therefore, an item-based approach will produce better and more stable recommendations for newly registered users that have rated or purchased only a few products so far.

However, in the case of the project of this Master's thesis, no such huge difference between the number of users and items exists. Therefore, a user-based collaborative algorithm will be a more appropriate algorithm for the project's implementation (see Section 6.3).

Similarity Functions

For the calculation of user-user similarities a similarity function is defined by $s : U \times U \mapsto \mathbb{R}$, where $U \in \mathbb{N}_0^n$ denotes the set of all possible user-vectors (i.e. possible row vectors of the rating matrix) containing only positive integer rating values (including zero for non-rated items).

In case of item-item similarities, both user vectors are simply replaced with two item vectors. For the sake of brevity, only the user-based method is described in the following. There is a great variety of different similarity functions which can be chosen depending on the given rating data. In order to limit the scope of this thesis, only the most relevant are explained below.

The Pearson correlation (Garcin, Faltings, Jurca, et al., 2009) is a statistical measure and expresses the linear correlation between two statistical variables which are in this case two user-vectors $\vec{u}$ and $\vec{v}$. Let I_{uv} be the set of items that have been rated by both users. Then the Pearson correlation coefficient is defined as:

$$s(\vec{u},\vec{v}) = \frac{\sum\limits_{i \in I_{uv}} (r_{u,i} - \bar{r_u})(r_{v,i} - \bar{r_v})}{\sqrt{\sum\limits_{i \in I_{uv}} (r_{u,i} - \bar{r_u})^2 \sum\limits_{i \in I_{uv}} (r_{v,i} - \bar{r_v})^2}} \tag{4.14}$$

Another typically used similarity function is the cosine similarity which describes the angle between two user vectors ($\vec{u}$ and $\vec{v}$) in Euclidean space:

$$s(\vec{u},\vec{v}) = \cos(\vec{u},\vec{v}) = \frac{\vec{u}\cdot\vec{v}}{||\vec{u}|| \times ||\vec{v}||} = \frac{\sum\limits_{i\in I_{uv}} r_{u,i}r_{v,i}}{\sqrt{\sum\limits_{i\in I_u} r_{u,i}^2}\sqrt{\sum\limits_{i\in I_v} r_{v,i}^2}} \tag{4.15}$$

The above Formula 4.15 does not consider the fact that users tend to have other "baselines" around which their ratings may vary. Therefore, the adjusted cosine similarity (Formula 4.16), which is a slightly modified version of this formula, should be used instead. The formula subtracts each of both user's individual average of all their ratings from each rating of that user.

$$s(\vec{u},\vec{v}) = \frac{\sum\limits_{i\in I_{uv}} (r_{u,i}-\bar{r_u})(r_{v,i}-\bar{r_v})}{\sqrt{\sum\limits_{i\in I_u} (r_{u,i}-\bar{r_u})^2}\sqrt{\sum\limits_{i\in I_v} (r_{v,i}-\bar{r_v})^2}} \tag{4.16}$$

All aforementioned similarity measures return a scalar value that ranges between -1 and 1, meaning $-1 \leq s(\vec{u},\vec{v}) \leq 1$. This scalar result describes either how similar ($0 < s(\vec{u},\vec{v}) \leq 1$) or dissimilar ($-1 \leq s(\vec{u},\vec{v}) < 0$) the user u is to the user v. If $s(\vec{u},\vec{v}) = 0$, then both users are neither similar nor dissimilar.

Although these measures are very popular for calculating the similarity between two users, they do not always work well for binary feedback which includes explicit positive-only ratings and various kinds of implicit feedback (see Section 4.2). In case of binary feedback the user vector is a bit-vector containing only 1's for liked items and 0's for unseen or unrated items (i.e., $r_{u,i} \in \{0,1\}$). Since no negative likes are available, only a positive similarity correlation between two users can be determined. An appropriate similarity measure for binary feedback is the Jaccard distance (Liu et al., 2014; Candillier, Meyer, and Fessant, 2008), which measures the taste-overlap of the rating set R_u of the user u and R_v of the user v:

$$s(R_u,R_v) = J(R_u,R_v) = \frac{|\ \{R_u \cap R_v\}\ |}{|\ \{R_u \cup R_v\}\ |} \tag{4.17}$$

The set R_u defines the set of all items liked by user u. The result of the Jaccard distance ranges between 0 and 1 ($0 \leq J(R_u, R_v) \leq 1$) and can only describe how similar the user u is to the user v. In contrast to the previous formulas, a dissimilarity between both users (i.e. negative similarity) can not be obtained from positive-only feedback.

As the Pearson coefficient (Formula 4.14) does not consider the number of items two users have in common, a more suitable version for non-binary ratings exists that combines it with the Jaccard similarity measure (Formula 4.17) (Candillier, Meyer, and Fessant, 2008; Garcin, Faltings, Jurca, et al., 2009):

$$s(R_u, R_v, \vec{u}, \vec{v}) = \frac{|\ \{R_u \cap R_v\}\ |}{|\ \{R_u \cup R_v\}\ |} * \frac{\sum\limits_{i \in I_{uv}} (r_{u,i} - \bar{r_u})(r_{v,i} - \bar{r_v})}{\sqrt{\sum\limits_{i \in I_{uv}} (r_{u,i} - \bar{r_u})^2 \sum\limits_{i \in I_{uv}} (r_{v,i} - \bar{r_v})^2}} \tag{4.18}$$

Model-based Collaborative Filtering

In practice, the rating matrix R (see Definition 4.1) is expected to be very sparse, meaning the majority of elements (or cells) in the matrix is set to zero[6]. This is simply due to the fact that the number of items in a system is often usually so huge (sometimes thousands or even millions of items) that most users have evaluated a few items and only a small fraction of users have seen and rated almost all items. As soon as the data volume of the rating matrix becomes very large, memory-based recommender systems often reach their limits of scalability and speed. This can lead to several problems when it comes to real-time recommendations.

Instead of a memory-based system, a model-based technique can be used. The basic idea of the model-based approach consists in building a model based on the rating matrix, which is then used to make recommendations (Breese, Heckerman, and Kadie, 1998). More precisely explained, relevant pieces of information are extracted from the rating data in order to learn or estimate a

[6]Note: In literature these empty cells are often left blank instead of set to zero.

model. These pieces reveal latent factors in the rating matrix which encapsulate the existing ratings (Koren, 2010), but omit all these zeros at the same time. This model is pre-computed and then used to make recommendations to the user. Once this model is built, the complete rating matrix is not needed anymore to generate recommendations. To phrase it differently, the prediction algorithm later operates on a compressed version of the rating matrix which is represented by the model. Hence, model-based recommender systems are usually faster and more scalable. Compared to memory-based approaches, the recommendations produced by model-based systems are often of similar quality.

From a probabilistic point of view, the process of collaborative filtering can be considered as the estimation of the expected value of a rating, given all the information about the active user we have collected so far (Breese, Heckerman, and Kadie, 1998). The goal is to build a model that can predict these rating values for items the user has not evaluated yet.

In order to generate a model, a data mining or machine learning technique is applied on the training data. Some examples of such techniques include latent semantic analysis, singular value decomposition, association rules, restricted Boltzmann machines, Bayesian networks, decision trees, and clustering (Hofmann, 2004; Brand, 2002; Salakhutdinov, Mnih, and Hinton, 2007; Miyahara and Pazzani, 2000). In case the rating matrix contains binary, positive-only ratings (i.e. zeros and ones), or more appropriate algorithms, e.g., specially adapted matrix factorization techniques, Bayesian Personalized Ranking or more probabilistic models have to be used (Y. Hu, Koren, and Volinsky, 2008; Verstrepen, 2015; Johnson, 2014; Rendle et al., 2009; Mnih and Teh, 2011).

Even if model-based recommender systems have many benefits as explained above, there are also some drawbacks. These are mainly the high costs to create the model in advance, the difficult interpretability of the predictions, and the loss of useful information due to the data reduction process.

4.4.3. Challenges of Collaborative Filtering Approaches

In this section, some of the major challenges of implementing a recommender system as described by Su and Khoshgoftaar, 2009 are briefly discussed. Since most systems are based on collaborative filtering, including the implementation of the project of this thesis, the problems explained in the following subsections mainly focus on such systems but may also be relevant for other kinds of recommender systems.

Data Sparsity

In practice, there are often situations where not enough rating data is available for the items. Usually, in these contexts the rating matrices can get very large and sparse. Especially in case of collaborative filtering based systems, the quality of the predictions depends on the number of ratings made by other users. If an item does not have enough ratings, then it is probably not recommended very often by a collaborative filtering system. However, the rating matrices are often very large and therefore enough recommendable items can still be found for which a decent amount of ratings exists.

Scalability and Limited Computational Power

The computational complexity of neighbourhood-based techniques such as memory-based collaborative filtering approaches often grows with the number of items and users in the rating table. Many e-commerce websites often have hundreds of thousands or millions of users and items, but require real time feedback from a recommender system. In such cases, even algorithms with linear or polynomial time complexity will work very slowly. Therefore, several model-based collaborative filtering techniques (e.g., incremental singular value decomposition) exist that perform dimensionality reduction in advance using existing users and are able to generate suggestions of comparable accuracy more quickly (Sarwar et al., 2002).

Cold Start Problem

Besides having sparse rating matrices, there is the cold start problem which particularly affects recommender systems based on collaborative filtering. One can distinguish two types of the cold start problem, the *new item problem* and the *new user problem* (Yu et al., 2004; Adomavicius and Tuzhilin, 2005).

The *new item problem* occurs every time a new item is introduced. New items are added as a new empty column to the rating matrix, meaning no user has rated that item yet. Thus, a collaborative filtering algorithm can not predict any ratings of this item for a user since there is no rating of any other user yet. As a consequence, the new item can not be recommended to anyone until some users have rated it. One way to fix this issue would be to use a different approach, e.g., a content-based recommender which finds similar items based on attributes instead of ratings (see Section 4.4.1).

Another type of the cold start problem is the *new user problem*. It occurs every time a new user is added to the system by appending a new empty row to the rating matrix. Similarly to the previous problem, collaborative filtering systems are unable to predict any ratings for a new user since no rating data and no purchase history of that user exists. In this case, even a content-based approach, which recommends items based on a user's preferences, will not produce reasonable predictions either. Therefore, more appropriate techniques (e.g., knowledge-based recommender systems) have to be used in order to solve this kind of problem.

Shilling attacks

In recommender systems where the predictions are solely based on user ratings, undesired side effects may happen. This is because users may rate all their own items positively and those items from their competitors negatively. In particular for collaborative systems, it is therefore often necessary to introduce sophisticated precautions in order to discourage such shilling attacks (Lam and J. Riedl, 2004; Resnick and Varian, 1997).

Similar and Duplicate Items

In systems with a large number of items, it is very likely that there are many identical or very similar items that have different names or duplicate items. There will also be duplicates of the same item that share the same name but have a different *identifying number* (ID) than the original item. If a recommender system does not take this fact into consideration, such a system would treat these items as different ones. Consequently, this would have a negative impact on the overall performance of the system. A naive approach to tackle this problem would be, for example, the automatic use of a thesaurus to find alternative phrases. However, even similar terms and synonyms may have different meanings. Possible solutions that can overcome this problem in a more intelligent way are latent discovering algorithms such as *latent semantic indexing* (Deerwester et al., 1990) (see also Section 4.4.2).

Gray and Black Sheep

The term *gray sheep* describes a user whose opinion does not overlap with any other group of people in terms of similarity or dissimilarity. As collaborative filtering approaches are based on finding similar users or items, they can not make useful recommendations to such *gray sheep* users (Claypool et al., 1999). However, other techniques such as content-based recommender systems can be applied instead. A *black sheep* in turn is a user who shares the same opinion with too many different user groups. In other words, their taste completely varies and thus, making predictions for a recommender system very difficult. Even though this can be interpreted as an issue of a recommender system, it is not a problem specific to recommender systems as it also affects people in real life who want to recommend something orally to other people.

4.4.4. Comparison between Content-based and Collaborative Filtering

Many content-based recommender systems generate suggestions based on keywords. Consequently, these content-based suggestions tend to be more understandable for users (Aggarwal, 2016). In particular, such content-based systems are usually good at recommending new items for which no sufficient rating data exists (cold start problem for new items, see Section 4.4.3), but are not as appropriate for recommendations to new users (cold start problem for new users, see Section 4.4.3) who have not rated many items yet (Aggarwal, 2016). In order to achieve high-quality results and to avoid overfitting, the system normally needs a high number of ratings of the active user.

In contrast to content-based approaches, which clearly represent the individual character of a user, collaborative methods focus on the user as an interacting individual which makes this technique more attractive for certain use cases. However, since content-based methods only take similarities between the user's profile and other not yet rated items into account, their major benefit is that they are not required to have a user community, as opposed to collaborative filtering. On the other hand, collaborative filtering completely relies on user ratings and can still be used in situations where no content information about the items is available.

Furthermore, a collaborative filtering system, unlike a content-based one, is able to discover and recommend relevant "serendipitous" items to the user even without having related content in a user's profile (Schafer, Frankowski, et al., 2007). Relevant "serendipitous" items are unseen items in the collection that are rarely liked or bought by other users (see "niche products" in Figure 4.1) and may have high relevance to the active user. However, in case of a content-based system, if a user has never seen an item that is mapped to several keywords, a related item with the same keywords can not be recommended. This is because the content-based model only sees the active user's preferences, but does not include other interests of similar users, resulting in reduced diversity of recommended items (Aggarwal, 2016).

4.4.5. Hybrid Approaches

After the strengths of the aforementioned approaches have been presented, the question arises whether they can be combined into one single recommender system which is able to exploit the synergy potential. A hybrid system aims to tackle this challenge by using various recommendation techniques to achieve higher prediction accuracy (Burke, 2002). In other words, a hybrid recommender system is made up of several subsystems (or components) having different advantages and disadvantages and disparate types of input.

A common example in practice is the combination of a content-based and a collaborative filtering system. In this case, the content-based subsystem could act as a fallback system for the collaborative subsystem to find content-related items whenever the cold start problem for a new item occurs. The collaborative component may in turn reveal "serendipitous" items rated by similar users which would not have been recognized by the content-based approach. There are also hybrid system which use the *same* recommendation approach in each subsystem (e.g. different models of collaborative-based recommenders).

Based on their approach, hybrid systems can be classified into weighted, switching, mixed, feature combination, feature augmentation, cascade, and meta-level systems (for further information see Burke, 2002). Cunningham et al., 2001 show a basic and easy-to-install hybrid system that consists of a collaborative filtering and a content-based component.

4.5. Evaluation Methods

In order to study the quality and effectiveness of a recommender system, its predictions have to be evaluated. Basically, there are three classes of evaluation methods which are briefly covered in this section.

4.5.1. Offline Evaluation

Offline testing allows the designers of a recommender system to evaluate the performance of the recommender system by using historical user data (e.g., user ratings or purchase history) of an online system. The offline evaluation method is primarily intended to be used to efficiently preselect a small set of candidates from a large set of different recommendation algorithms (Shani and Gunawardana, 2009). This is mainly because, unlike other evaluation techniques, offline evaluation runs very quickly on huge data sets and does not require real users to be involved in the testing process. Instead the user behavior and interaction with the recommender system is simulated by using the historical data.

Usually, the dataset is either split randomly or based on the temporal order of the data (if the timestamps of user interactions/ratings are available), into two disjoint subsets, a training set, and a test set. During the training phase, the model of the recommender system is fitted to the training data by using an algorithm from the set of available candidates. It is necessary to ensure that both, the training set and the test set, do not overlap (i.e. disjoint sets) in order to avoid *model overfitting*. After training, the trained model is used to make predictions for all instances of the test set. Then, the accuracy of these predictions is measured based on the expected results in the test set. In the next step, the parameters of the model are tuned and the updated model is retrained and retested in the next iteration. The prediction quality is measured again and again after each iteration until the accuracy increases to a certain maximum value. At the end, the same sequence of actions is repeated with the next algorithm from the set of available candidates. This model selection process can be further improved by using more sophisticated validation techniques such as *k-fold cross validation* or *leave-one-out cross validation* (see Wit, 2008).

According to J. L. Herlocker et al., 2004, there are three different classes of accuracy metrics: *predictive accuracy* metrics, *classification accuracy* metrics, and *rank accuracy* metrics. For the sake of brevity, only some of the most frequently used metrics are listed below.

Predictive Accuracy Metrics

Predictive accuracy metrics aim to evaluate the accuracy of the predicted ratings. The *mean absolute error* (MAE) is a widely used predictive accuracy metric and computes the deviation between the predicted ratings (denoted as $\hat{r}_{u,i}$) and the actual ratings (denoted as $r_{u,i}$).

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |\hat{r}_{u,i} - r_{u,i}| \tag{4.19}$$

The *root mean square error* (RMSE) is another well-known predictive accuracy metric. In contrast to MAE, it puts more emphasis on larger deviations between the predicted and actual ratings by squaring the differences.

$$\text{RMSE} = \frac{1}{n} \sum_{i=1}^{n} (\hat{r}_{u,i} - r_{u,i})^2 \tag{4.20}$$

Classification Accuracy Metrics

In many recommender systems, the focus is not on increasing the prediction accuracy, but on relevance of the proposed items. The goal of classification accuracy metrics is to evaluate how relevant the items in the recommended list are. Some of the very often used measures for such problems are typical information retrieval evaluation metrics such as precision, recall, and the F1-measure.

Precision defines the fraction of received predictions that are relevant. It expresses how many of the given recommendations are relevant.

$$\text{precision} = \frac{|\{\text{relevant\_predictions}\} \cap \{\text{received\_predictions}\}|}{|\{\text{received\_predictions}\}|} \tag{4.21}$$

Recall represents the fraction of relevant predictions that are successfully received and explains how many relevant recommendations of all expected recommendations are received.

$$\text{recall} = \frac{|\{\text{relevant\_predictions}\} \cap \{\text{received\_predictions}\}|}{|\{\text{relevant\_predictions}\}|} \tag{4.22}$$

The *F1 measure* combines precision and recall and describes the harmonic mean between them:

$$\text{F1} = 2 * \frac{precision * recall}{precision + recall} \tag{4.23}$$

Rank Accuracy Metrics

In many applications, instead of showing a small list of top-N recommended items, a very long list of items is divided into multiple pages. Users can then browse through these pages. Thereby, *rank accuracy evaluation metrics* are introduced. These metrics evaluate the order or ranking of the items based on the preferences of the respective user. Examples of rank accuracy metrics are the *normalized distance-based performance measure* and the *half-life utility metric*. For more details, see J. L. Herlocker et al., 2004.

4.5.2. Online Evaluation

The underlying goal of testing an recommender system online is to monitor and investigate the reaction and the change in user behavior whenever users receive recommendations generated by *different* recommender algorithms (Shani and Gunawardana, 2009). In contrast to offline evaluation, the online method records and statistically examines live interactions from real users. The online evaluation is directly integrated into an active environment of different recommender systems equipped with distinct recommendation algorithms. Typically, the online test randomly splits the set of active users of an online application into multiple user groups, each receiving recommendations from a different recommender system (i.e. different recommender algorithm). Apart from the different suggestions, the online method usually presents the exact same user interface of the application to all user groups. Hence, users do not notice that they participate in a "hidden" online experiment. This online testing approach is also named *A/B testing* or *split testing* (Kohavi and Longbotham, 2015). According to Kohavi and Longbotham, 2015, A/B Testing is a data mining technique that allows establishing a causal relationship with high probability. In a controlled experiment, users are randomly split between different variants[7] in a persistent manner. Thereby, "A" refers to the control version and "B" to the variation, whereby both provide the exact same user experience but a different list of recommendations. It is important to be note that A/B testing is not limited to two versions A and B. There can also be further versions C, D, E, etc. In practice, version A is typically used to define a baseline against which all other versions are compared. The online interactions of the users are then instrumented and key evaluation metrics such as *click-through rate* or the *conversion rate* are computed. In terms of recommender systems, system designers of these systems can utilize this evaluation technique to form a hypothesis of the form "If a specific change in the recommendation algorithm is introduced or a recommendation algorithm is replaced by another, will it improve the key evaluation metrics?" and evaluate it online with real users (Kohavi and Longbotham, 2015). The results of the online evaluation are then used to conclude which recommender system performs best.

[7] In this specific use case different variants mean different recommender algorithms.

4.5.3. User Studies

Similar to online evaluation, user studies evaluate the behavior and feedback of real users for various recommender versions (Ricci et al., 2010; Shani and Gunawardana, 2009). Besides the fact they can be run offline, these tests are specifically designed to be applied in more constrained and controlled environments. In contrast to online evaluation, the number of users is often much smaller and each user is given a bunch of tasks in order to assess each recommender system. User studies allow researchers to select a specific target group of users. In addition to that, users can be asked further quantitative and qualitative questions related to their tasks (Ricci et al., 2010; Shani and Gunawardana, 2009). For example, users may be asked how they liked the user experience or whether the suggestions were relevant to them or not. Moreover, in some cases the physical behavior of the user (e.g., tracking the movement of the user's eyes via webcam) can be captured as well. Obviously, it is no secret that user tests, unlike online and offline evaluation approaches, possess the potential to reveal allegedly latent factors.

4.5.4. Comparison of Evaluation Methods

Besides its efficiency and simplicity, offline evaluation methods have some serious shortcomings (Shani and Gunawardana, 2009; Wit, 2008). For example, offline testing is unable to fully cover user satisfaction as well as the recommender system's influence on the behavior of the user. Furthermore, the offline evaluation only allows the designers of a recommender system to evaluate predictions for items which have been rated or purchased by users in the past. In practice, a large amount of unrated/unpurchased items exists which can not be accounted for by this evaluation approach. Moreover, Beel and Langer, 2015 and Garcin, Faltings, Donatsch, et al., 2014 show that results of offline evaluations and online evaluations or user studies do not correlate and are sometimes contradictory. Instead of offline evaluation, more sophisticated approaches such as online evaluation or user studies have to be applied. For example, an online test lets thousands or even millions of users participate simultaneously in the online experiment. Since online and user studies do not have to simulate user behavior, they can evaluate direct feedback from real users and are therefore able to measure user satisfaction.

Furthermore, the evaluation also includes the recommender system's influence on the user behavior of the user. All these things can not be achieved in an offline setting.

In practice, however, running online experiments could be risky sometimes (Shani and Gunawardana, 2009). For instance, in case a group of users receives inappropriate recommendations produced by an algorithm under test, some of them may stop using the application forever. Therefore Shani and Gunawardana, 2009, propose to first perform extensive offline tests in order to find a small set of promising candidates, then conduct a user study with some test users and finally run the online experiment.

To sum up, this chapter gave an overview of the broad research area of Recommender Systems. It described the recommendation process and discussed the difference between user-specific and non-personalized recommendations as well as the disparity between explicit and implicit input data. Finally, several recommendation approaches and evaluation methods that define the foundation for the practical part of this Master's thesis, were explained. Now that all essential background knowledge and relevant methods have been presented, this sentence marks the end of the theoretical part of this thesis.

5. Catrobat Project

The focus in the subsequent chapters now lies on the practical relevance of the Catrobat project. In this chapter the Catrobat project is introduced and an overview of the organisation's core projects is given.

5.1. About the Organisation and the Project

The Catrobat organisation[1] was founded in 2010 with the aim of promoting the development of free educational apps for children (12 year olds and above), teenagers, and other novices. The main idea of the Catrobat project consists of improving young people's computational thinking skills by sharing their self-created apps. Catrobat is the name of a free and open source software (FOSS) project and also the official name of the project's visual programming language.

The Catrobat project has already won a number of awards, including the Austrian National Innovation Award 2013[2] in the category of Multimedia and the Regional Award Europe 2016[3] at the Reimagine Education Awards from the Wharton School of the University of Pennsylvania. Moreover, the Catrobat project got honored with the Austrian "Internet for Refugees" award[4] in 2016 for the ongoing development of a Right-to-Left language version of its Pocket Code app (see Section 5.3). This app version supports Right-to-Left languages such as Arabic or Farsi, and particularly focuses on refugees and children in crisis or development areas.

[1] http://www.catrobat.org

[2] http://www.ffg.at/content/sieger-des-staatspreises-multimedia-und-e-business-2013

[3] http://www.reimagine-education.com/awards/reimagine-education-2016-honours-list/

[4] http://www.tugraz.at/en/tu-graz/services/news-stories/tu-graz-news/singleview/article/preis-internet-for-refugees-fuer-programmier-app-der-tu-graz

Catrobat currently participates in the "No One Left Behind"[5] research project funded by the Horizon 2020 program of the European Union. Google is also supporting the Catrobat project by featuring it on Google Play for Education. It has accepted it six times for the Google Summer of Code program[6] so far.

5.2. Catrobat: The Programming Language

The visual programming language of the Catrobat project is especially designed for mobile platforms, such as smartphones, tablets, and mobile web browsers. It has been inspired by Scratch[7], a programming environment created by the Lifelong Kindergarten Group at the MIT Media Lab.

Similar to Scratch, the programming elements of the Catrobat language are graphical building blocks, called "bricks". The language also supports event-driven programming with multiple objects. There are two different types of bricks: action bricks and head bricks. Action bricks describe a basic instruction (e.g., move an object 10 steps forward) which is performed when the brick is executed during runtime. These bricks are arranged in head bricks which represent scripts and receive events and look slightly different from action bricks (see Figure 5.1). In other words, each action brick is assigned to a dedicated head brick. By using many head bricks for the same event in an object, users are able to create very powerful interactive programs like games or animations. Figure 5.1 visually demonstrates two head bricks and their corresponding action bricks as part of an object of a Catrobat program.

In order to create such multimedia programs the Pocket Code programming environment is used and asset files (i.e. images & sound files) can be created or imported to the program. The entire Catrobat program is compressed to a single ZIP-archive containing the serialized code in an XML file and its asset files. All programs created with Pocket Code are licensed under the GNU Affero General Public License (AGPL)[8] and can be freely reused, extended, or modified by other users.

[5]No One Left Behind: `http://no1leftbehind.eu`
[6]Google Summer of Code: `http://developers.google.com/open-source/gsoc`
[7]Scratch: `http://scratch.mit.edu`
[8]AGPL: `http://www.gnu.org/licenses/agpl-3.0.en.html`

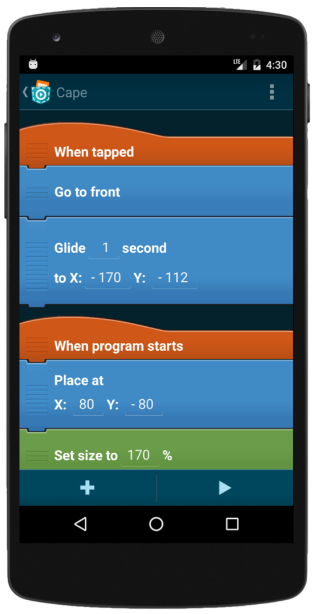


Figure 5.1.: Visualization of two head bricks (shown as orange hat-shaped bricks) and their corresponding action bricks (see blue and green colored bricks) as part of an object named "Cape"

5.3. Pocket Code App

The Catrobat programming environment is part of the mobile app "Pocket Code". It consists of a programming language interpreter for the Catrobat language and a *visual integrated development environment* (visual IDE). The IDE is responsible for automatically transforming the underlying code parsed from the XML file into visual brick elements and vice versa. It allows users to modify or create new programs by simply adding and putting brick elements together via drag and drop.

In Figure 5.2 objects of a Catrobat program are visually displayed through Pocket Code's IDE.

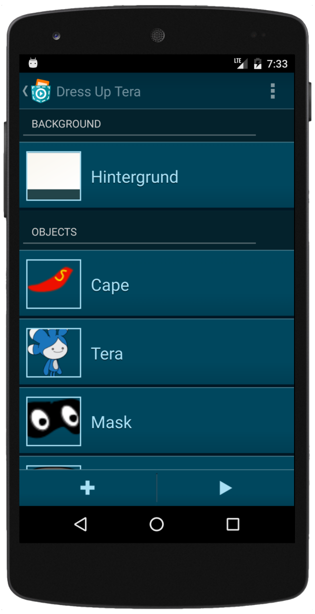


Figure 5.2.: Pocket Code's IDE listing the objects of a Catrobat program

The Pocket Code app is available on Google's Play Store for Android [9] and is expected to be released soon on the Apple App Store for iOS [10].

5.4. Catrobat's Community Website

Part of Pocket Code is also a mobile version of the community website[11], which is seamlessly integrated as a webview into the app. The web application of the community website is developed as an independent project and named *Catroweb*. It provides an online platform for users to download and upload programs, share them with other users, search for programs, and express feedback on them (e.g. post a comment, rate the program, report the program

[9]Pocket Code app on Google Play Store: https://catrob.at/pc
[10]Apple iOS App Store: http://www.apple.com/appstore
[11]Catrobat's Community Website: http://share.catrob.at

as inappropriate, etc.). The web application is based on the Symfony Web Framework[12] and implemented in PHP, a web programming language. The data is stored in a relational MySQL database[13] and it uses Doctrine[14] as an object-relational mapping tool.

Figure 5.3 shows a screenshot of the mobile version of the community website revealing its latest programs (see "newest programs"). The programs are presented as small, clickable images.

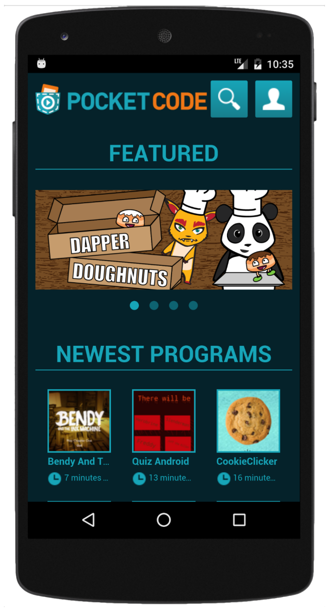


Figure 5.3.: Mobile version of the Pocket Code's community website

[12]Symfony Web Framework: http://symfony.com
[13]MySQL: http://www.mysql.com
[14]Doctrine: http://www.doctrine-project.org

5.5. Scratch to Pocket Code Converter

Apart from Pocket Code for Android/iOS and the community website, Catrobat has many other smaller subprojects that are either part of Pocket Code (e.g. several projects for controlling robots via Catrobat bricks) or are partial implementations of Pocket Code for other platforms (for instance, Catrobat's HTML5 project). Although explaining all of them would go beyond the aims of this thesis, the Scratch Converter[15,16] subproject should be mentioned here due to its relevance to the project's goals. Since the Catrobat programming language is strongly influenced by Scratch, converting Scratch programs into Catrobat programs is possible without any major restrictions. At the time of writing this thesis, Scratch is a Flash-based[17] visual programming environment which is supposed to be used in desktop web browsers. This stands in contrast to Catrobat, which is tailored to mobile devices. However, both programming languages provide similar functionality and features. Moreover, at the time of writing this thesis, Scratch has over 22 million programs (or also labelled as Scratch projects) with over 18 million registered users[18] and hence provides a rich set of high-quality media programs. Therefore, there is great interest in bridging the gap between Scratch and Catrobat. One approach to achieve this is the Scratch Converter project, which deals with the aforementioned challenges. The converter project refers to a conversion tool which is directly integrated into Pocket Code. In addition to that, users can also directly search for and convert Scratch programs within Pocket Code.

Figure 5.4 illustrates the conversion process which takes place in several successive steps. First, the code of the Scratch program is compiled into Catrobat code. Second, the code is then serialized to a new XML document. Third, some necessary metadata is added to the XML document as well and finally, the file is persisted to disk. Such metadata includes, for example, extra information about the original Scratch program, relevant remix data, and much more. The remix data is especially important here. It is needed to keep track of the remix relationship between the converted Catrobat and original Scratch program within the Catrobat ecosystem.

[15]Scratch Converter: http://scratch2.catrob.at
[16]Github Project of Scratch Converter: http://github.com/Catrobat/ScratchToCatrobat
[17]Adobe Flash Player: http://www.adobe.com/software/flash/about
[18]Scratch Statistics: http://scratch.mit.edu/statistics

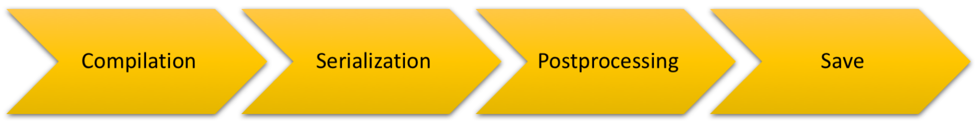


Figure 5.4.: Conversion flow of Scratch Converter

It is important to be aware of that the remix data in the XML is also used for another purpose, which is to link a modified version of a Catrobat program with its original version. The remix information in the XML defines the basis of the remix functionality in the next Chapter 6.

To summarize, this chapter gave an overview of the Catrobat organisation and the Catrobat programming language. Moreover, the most important subprojects which are relevant to the project of this thesis, including the Pocket Code app, community website, and Scratch converter, were introduced and briefly discussed. The next chapter, Chapter 6, describes the implementation part of this thesis.

6. Implementation and Evaluation

This chapter describes the implementation and evaluation of different collaborative recommendation approaches within the practical part of this thesis. A basic recommender system was built and directly integrated into Catrobat's collaborative web system, the community website (see Section 5.4). In addition, a variety of features to collect data from users was added to the web system. This collaborative information serves as input for the recommender system. Moreover, basic functionality to present a list of recommended Catrobat programs to the user, as well as to graphically visualize connections between remix-related Catrobat programs, was implemented. These visualization tools are intended to assist the users in the finding process. Finally, the used A/B tests are explained which were conducted to evaluate the performance of the different recommendation approaches. The aim of the evaluation was to answer the research questions which are defined in Chapter 1.

6.1. Implementation of Remixing

This section gives an overview of the implementation of the remixing feature in Catrobat. The Pocket Code app makes it very easy for its users to download programs from Catrobat's community website and to remix them. All remixing data is automatically stored in the XML file of the downloaded program and remains there when the program is modified. This guarantees no remix information between remix-related programs can get lost. The main idea behind remixing is described in Section 3.4. With the implementation of the remixing feature on the server side, Catrobat's community website is able to track the collaborative remixing activity of all users and to collect important data. The remixing data is then used in Section 6.3 as an input for a collaborative recommendation approach.

Before the beginning of this Master's thesis, essential parts of the remix implementation on the client-side and server-side already existed, including support for *Remixing without Merging* (see Section 6.1.2) and *Merged Remixing* (see Section 6.1.2). All information was completely stored in the XML files, meaning no database model existed. Furthermore, the merge implementation accidentally combined wrong or invalid remix relations. One main goal of this thesis was to address and fix this merge issue, and to provide support on the server-side for new features such as *Support for Program Updates* (Section 6.1.2), *Remixing of Scratch Programs* (Section 6.1.2) and *Remix Graph Visualization* (Section 6.1.3).

6.1.1. The Remix Graph

Before the implementation can be discussed in more detail, a structure to describe the remix-relationship must be introduced and the process of remixing should be explained as well. In the context of this thesis, mathematical graphs are used to model the remix-relationship between Catrobat programs. Such graphs are named *remix graphs*. Thereby, a node in a remix graph represents a Catrobat program (or a Scratch program, see Section 6.1.2) and is connected via directed edges with its remixed Catrobat programs, its child programs. A *directed edge* is an arrow which points from the parent program to the child program (i.e., remixed program). Directed edges are used to describe immediate relationships between two nodes. Furthermore, more distant relationships between two nodes can be explained by using directed paths in a graph. A *directed path* defines a sequence of successively connected edges which are directed in the same direction. The shortest possible directed path is a directed edge and describes the aforementioned parent to child relationship in a remix graph. Longer directed paths are used to express more distant relationships such as grandparent to grandchild, great-grandparent to great-grandchild, etc.

Figure 6.1 shows an example of a remix graph. The nodes (i.e., programs) are arranged in chronological order from top to bottom. Each program to which an arrow points (i.e., incoming link or incoming directed edge) is a child program (i.e., remix) of another program (i.e., its parent program). In this graph, there are two programs, “Five Nights” and “JonnetPelaa”, which have no incoming link and hence are not remixed programs. Such programs are named *root programs* or *base programs*. The reason why they are shown is that they have been remixed, meaning they are parents of another program. In addition, directed paths can be used to express the remix relationship between two immediately/directly related programs. For example, the path between the root program "Five Nights" and the program "Logos" describes a great-grandfather to great-grandchild relationship.

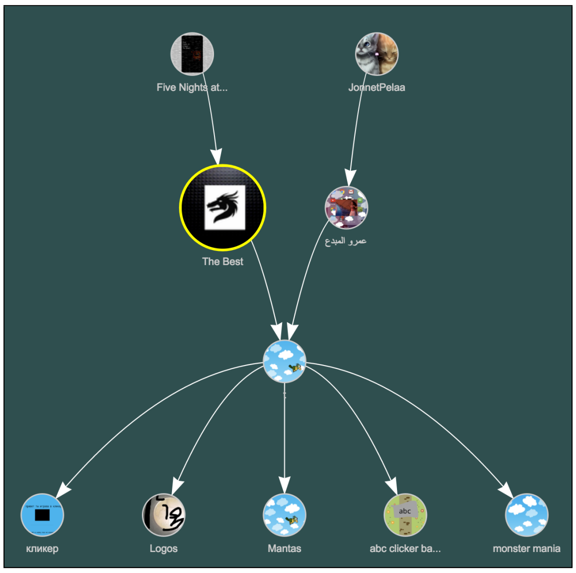


Figure 6.1.: Example of a remix graph. The yellow cycle highlights the current program.

The structure of the remix graph depends on the order in which the programs were remixed. In the following, the process of remixing Catrobat programs and different ways of remixing are presented.

6.1.2. The Process of Remixing in Catrobat

This section describes the entire use case of remixing in Catrobat. Thereby, the communication takes place between the app (client-side) and the community website (server-side). As previously mentioned, the remixing information is stored in the XML file of an uploaded program. The header of each program's XML file contains many fields. Only two of them are relevant to remixing: these are the two fields `url` and `remixOf`.

Figure 6.2 visualizes the process of remixing a single or multiple programs in a sequence diagram. The case of remixing more than one program is called *merging* or *merged remixing*. Merging is described in Section 6.1.2 and appears as an optional step in the sequence diagram. The process is best explained as follows:

i. **Download:**
The user downloads a program from the community website within the Pocket Code app. The downloaded program represents a *copy* of the original program. The `url` field of the downloaded program contains the URL of the *original* program and the `remixOf` field contains the URL of the original's parent program.

ii. **Merging (optional):**
The user merges a second program into the downloaded program by using the Pocket Code app (see Section 6.1.2). Thus, the merged program is now a remix of two parent programs. Now, the app appends the URL of the second program to the existing URL in the `url` field of the XML. This step can be repeated many times in order to remix more than two parent programs.

iii. **Remixing:**
The downloaded (or merged) program is opened and modified by the user via the Pocket Code app. It is important to be aware that the values of both XML fields are not updated by the app.

iv. **Upload:**
The user uploads the modified program via the Pocket Code app. The server of the community website copies the URLs of all parent programs to the `remixOf` field in order to preserve them. In addition, the server generates a "Catrobat-wide" *unique ID* for the uploaded program and assigns a unique URL based on the generated ID to the `url` field of the *uploaded* program's XML header.

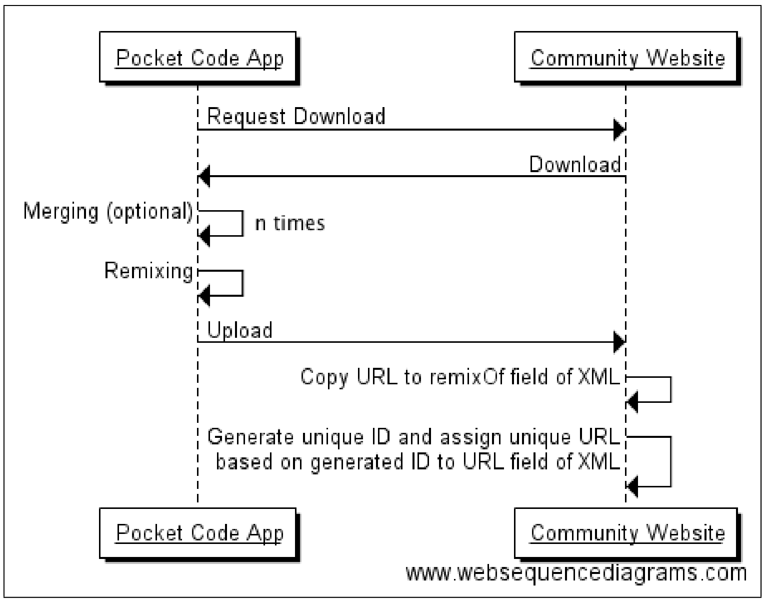

Pocket Code App
Community Website
Request Download
Download
Merging (optional)
n times
Remixing
Upload
Copy URL to remixOf field of XML
Generate unique ID and assign unique URL based on generated ID to URL field of XML
Pocket Code App
Community Website
www.websequencediagrams.com

Figure 6.2.: Sequence diagram describing the process of remixing
Generated by: `http://www.websequencediagrams.com`

This organized process ensures that the XML header of the uploaded program contains the URL of all parent programs in the `remixOf` field as well as the URL for itself in the `url` field after upload. If another user downloads the uploaded program and remixes it some time later, the Pocket Code app of this user can reliably assume that the remixing information in the XML is already up to date. To phrase it differently, the app never needs to update the two XML fields, as this is already handled entirely on the server.

Remixing without Merging

Remixing without merging refers to remixing a *single* Catrobat program. Thereby, a child program is based on exactly one parent program. Such a remixed child program can be uploaded and remixed again by other users. Subsequently, the same can happen repeatedly to its child programs again. This remixing process continues as long as other users create other remixes based on these new remixed remixes and so on. This way, a hierarchical tree structure consisting of remixed programs is created. In general, a tree structure is a special form of an acyclic graph, where each child node must have exactly one parent node. Consequently, a tree must have exactly *one* root program. In the context of remixing, this tree is referred to as the *remix tree*.

According to the remix flow explained in Section 6.1.2, remixing without merging covers all steps of the sequence diagram except merging (step ii.). Listing 6.1 shows the XML header of an unmerged, remixed Catrobat program after upload. The `remixOf` and `url` fields (marked in red color) contain the URL of the parent program (ID is 817) and the URL of the remixed child program (ID is 2576).

```
<program>
  <header>
    <applicationName>Pocket Code</applicationName>
    <applicationVersion>0.9.14</applicationVersion>
    <catrobatLanguageVersion>0.92</catrobatLanguageVersion>
    <description>Three in one win. However, ...</description>
    <platform>Android</platform>
    <programName>Tic-Tac-Toe Master</programName>
    <remixOf>http://pocketcode.org/details/817</remixOf>
    <screenHeight>960</screenHeight>
    <screenMode>STRETCH</screenMode>
    <screenWidth>540</screenWidth>
    <url>http://pocketcode.org/details/2576</url>
    <userHandle>fusuy</userHandle>
    <!-- ... -->
  </header>
  <!-- ... -->
</program>
```

Listing 6.1: XML Header of a remixed program (without merging)

Figure 6.3 illustrates an example of a remix tree. At the top of the tree one can find the original root program, which is not a remix of any other program. The directed edges are shown as arrows pointing from the parent program to the remixed program. All other programs represent remixes which have only one incoming link, meaning they are solely based on one parent program.

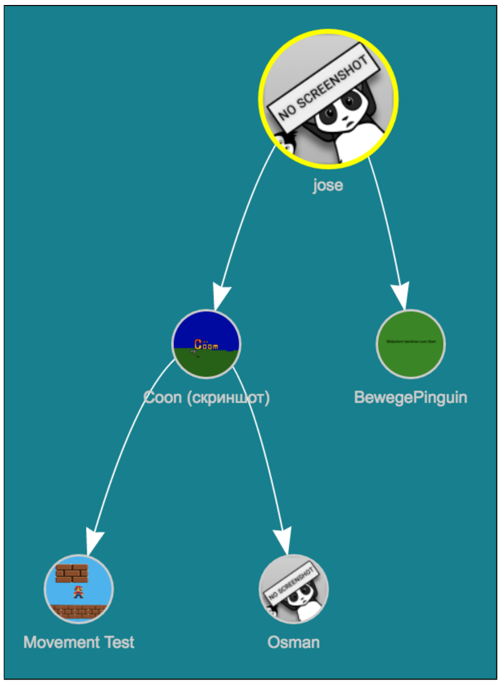


Figure 6.3.: Example of a remix tree, a special form of a directed acyclic remix graph

Merged Remixing

Merged remixing, also referred to as *merging*, allows users to remix multiple programs by merging them together. It is intended to support different users in group work by simply merging two programs together into one Catrobat program. In contrast to *remixing without merging*, where remixes are based only on a single parent program, merged programs can combine resources from many different programs. To phrase it differently, merged programs have multiple parents (at least two). The main idea of merging consists in further improving the collaborative e-learning process by enabling users to combine multiple ideas at once.

According to the remixing flow discussed in Section 6.1.2, merged remixing covers all steps of the sequence diagram (also including step ii.). Listing 6.1

shows the XML header of a merged Catrobat program after upload. The `remixOf` and `url` fields (marked in red color) contain the URL of both parent programs (IDs are 817 and 211) and the URL of the merged child program (ID is 201112).

```
<program>
  <header>
    <applicationName>Pocket Code</applicationName>
    <applicationVersion>0.9.14</applicationVersion>
    <catrobatLanguageVersion>0.92</catrobatLanguageVersion>
    <description>Three in one win. However, ...</description>
    <platform>Android</platform>
    <programName>Merged test program</programName>
    <remixOf>
      Tic-Tac-Toe [http://pocketcode.org/details/817],
      Test program [http://pocketcode.org/details/211]
    </remixOf>
    <screenHeight>960</screenHeight>
    <screenMode>STRETCH</screenMode>
    <screenWidth>540</screenWidth>
    <url>http://pocketcode.org/details/201112</url>
    <userHandle>john_doe</userHandle>
    <!-- ... -->
  </header>
  <!-- ... -->
</program>
```

Listing 6.2: XML Header of a merged program

Once a program is merged and uploaded to the server, the merge-relations between the programs cannot be modeled by a remix tree any more. As a consequence, the remix tree has to be automatically converted into a directed graph.

In case the merged program is uploaded for the first time, a *directed acyclic graph* is created. Basically, a directed acyclic graph is a limited type of a graph which contains no cycles. In the context of the remix graph, this acyclic limitation means that no remixed program can ever become a parent of *any* of its ancestor programs (including parent, grandparents, great-grandparents, etc.).

As a side note, the combination of both features, *merged remixing* and *remixing without merging* (see Section 6.1.2), transforms the entire collaborative Catrobat environment (including the Pocket Code app and Catrobat's community website) into a powerful *version control system*. These two features empower Catrobat's users to collaboratively work together on large and complex programs such as games, animations, etc. Compared to Git[1] which is a version control system for software development, in Catrobat, each regular remix (i.e., remix without merging) defines a contribution which can be viewed as an equivalent to a single commit in Git. Likewise, a merged remix in Catrobat can be understood as a merge commit in Git.

Support for Program Updates

Sometimes a creator of a program wants to make changes to his/her program or fix errors in the program after she or he has uploaded the program. Catrobat's community website supports this use case and allows users to re-upload their programs. However, in this use case, it appears to be very tricky to model the modified remix-connections correctly by a remix graph. There are three different scenarios that had to be implemented:

Case 1: Unchanged Acyclic Remix Graph

Unless a user does not merge another program into the already uploaded program and re-uploads it, the parents of the modified program will remain the same. Thus, the directed acyclic remix graph remains untouched after upload.

Case 2: Extended Acyclic Remix Graph

If the user merges the updated program with one of its ancestor programs (e.g., parent, grandparent, great-grandparent, etc.) or any other (yet) unrelated program, then the program receives a new parent. Consequently, the directed acyclic remix graph has to be extended with a new incoming link.

[1]Git Version Control System: `http://git-scm.com`

Case 3: Remix Graph with Cycles

However, in case the author merges the uploaded program with one of its descendant programs (e.g., child program, grandchild, great-grandchild, etc.) and uploads it again, the situation becomes more complex. Now, the uploaded program has received a new parent which is one of its descendant programs. This means, in addition to the existing directed path from the uploaded program to the descendant program, a new link and hence a new path between the descendant program and the uploaded program has been added to the graph. More precisely, before upload, the uploaded program and the descendant program were already connected once (only forwardly) and after upload, they are connected twice (forwardly and backwardly). In other words, a cycle was added to the previously directed acyclic remix graph. As a consequence, the remix graph is not acyclic any more. That means that the graph now contains a forward path from the uploaded program to its descendant program as well as a backward path from the descendant to the uploaded program. It is important to note that this is the only possible known-scenario when such a cycle can arise.

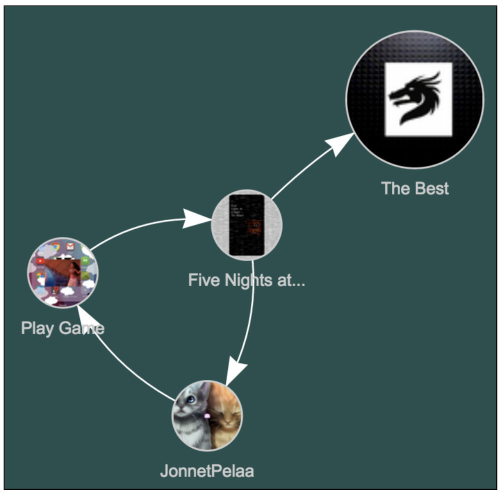


Figure 6.4.: Example of a remix graph with cycles

Figure 6.4 visualizes such an example of a cyclic remix graph. In this example, the program "Five Nights" has been merged with its grandchild "Play Game" and then re-uploaded. After the merge happened, the grandchild program "Play Game" suddenly became a parent of *its* grandparent program "Five Nights".

Remixing of Scratch Programs

As described in Section 5.5, the Scratch converter is a tool which is directly integrated into Pocket Code. It allows users to convert Scratch programs into Catrobat programs. Such converted programs are like normal Catrobat programs and can be uploaded to and shared by users on the community website. As a consequence, the community website has to ensure that valuable information about the original Scratch program (e.g., name of the Scratch user, attached copyright information, acknowledgements, etc.) is included in the information shown on the detail page of every converted program. Additionally, in terms of remixing, a converted Scratch program can be seen as a remix of the original Scratch program. Therefore, a converted program is a copy of the original Scratch program that can be edited by using the Pocket Code app.

Consequently, including the Scratch program in the remix graph of a converted program was an important task of the remix graph implementation. Every time a user converts a Scratch program, the converter tool automatically adds the hyperlink of the Scratch program to the XML file of the converted program. This hyperlink uniquely identifies the Scratch program and refers to the detail page of the original Scratch program. The task thereby was to extract this reference from the XML file and to support more complex merge use cases, such as the merging of two converted Scratch programs or merging a converted Scratch program into a normal Catrobat program.

Listing 6.3 lists the XML header of a converted Scratch program after upload. The `remixOf` and `url` fields (marked in red color) contain the URL of the original Scratch program (Scratch-ID is 17828107) and the URL of the converted program (Catrobat-ID is 1808).

```
<program>
 <header>
   <applicationName>Pocket Code</applicationName>
   <applicationVersion>0.9.9</applicationVersion>
   <catrobatLanguageVersion>0.92</catrobatLanguageVersion>
   <description>Made with ScratchToCatrobat ...</description>
   <platform>Android</platform>
   <programName>DJ Scratch Cat remix</programName>
   <remixOf>http://scratch.mit.edu/projects/17828107</remixOf>
   <screenHeight>360</screenHeight>
   <screenMode>MAXIMIZE</screenMode>
   <screenWidth>480</screenWidth>
   <url>http://pocketcode.org/details/1808</url>
   <userHandle>chwt</userHandle>
   <!-- ... -->
 </header>
 <!-- ... -->
</program>
```

Listing 6.3: XML Header of a converted Scratch program

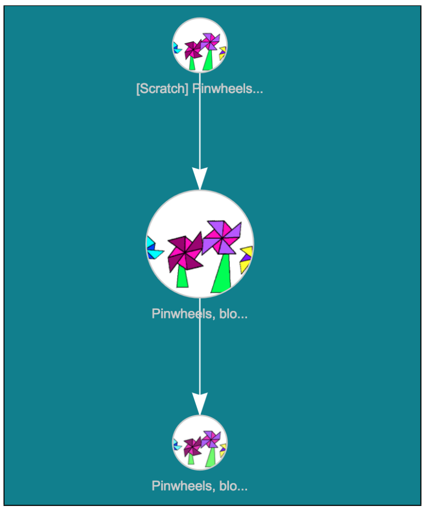


Figure 6.5.: Example of a remix graph including a converted program and its original Scratch program (prefixed with “[Scratch]”)

Figure 6.5 depicts a simple remix graph which includes the original Scratch program (see root program prefixed with "[Scratch]"). In this example, the program in the middle and the bottommost program are Catrobat programs. The program in the middle is the converted Scratch program; this is shown as a remix of the original Scratch program (i.e., direct child) at the top. The bottommost program is a regular remix of the Catrobat program in the middle.

Due to the nature of the remixing behavior in Catrobat, Scratch programs always appear as root programs in a remix graph, meaning they can only have outgoing links but no incoming links. Furthermore, a converted Scratch program is always represented as the direct (Catrobat) child of its corresponding original (Scratch) program. These fundamental facts simplify the design and implementation of the database model which is described in Section 6.1.3.

Remixing is also a key feature in Scratch and likewise, Scratch also allows its users to remix their Scratch programs (Monroy-Hernández, 2007). However, at the time of writing this thesis, unlike Catrobat, Scratch does not support merged remixing of their programs; therefore, inter-Scratch remix relations can always be modeled as *remix trees*. These *Scratch remix trees* are not included in Catrobat's remix graphs. However, the hyperlink referring to the original Scratch program is shown on the detail page of every converted program on Catrobat's community website. Thus, Catrobat users can easily click on this hyperlink and then have direct access to the link referring to the Scratch remix tree web page.

6.1.3. Implementation of Remix Graph Model and Visualization

In order to visualize large remix graphs very quickly, a database model for fast data accesses had to be designed and implemented. Before the visualization of the graph can be discussed in more detail, the database model has to be explained first.

Database Model of Remix Graph

As the XML files of the respective Catrobat programs already keep track of the relevant remix relations between them (including potential Scratch programs, see Section 6.1.2), it is sufficient to just extract the parent remix URLs from each program's XML file and convert this data into a more suitable data model. This data model should be a graph representation of all remix graphs. In addition, to support program updates (see Section 6.1.2), it is also necessary to sync the updated remix data in the XML file with this data graph model.

There are many possible ways to model such graph data, e.g., *graph databases* (Vicknair et al., 2010). However, the web application of Catrobat's community website (see Section 5.4) already stores all of its data in a relational MySQL database and uses Doctrine as an object-relational mapping tool. Furthermore, introducing a new technology to the web application would increase the costs of maintenance and testability. Therefore, the author of this thesis decided to create an self-developed solution for the remix graph model which perfectly fits into the existing relational database solution.

As already discussed throughout this chapter, it is required to support two types of graphs in order to cover all possible remixing scenarios in Catrobat. These are *directed acyclic graphs* and *directed graphs with cycles*. The easiest way to model a graph with a relational database would be to use a single many-to-many relationship table. However, fetching a complete graph in MySQL, which does not support recursive SQL queries, requires using a sequence of successive SQL queries. This solution would be very slow for large graphs and would require using MySQL specific stored procedures.

Thus, the implemented solution follows a more efficient and abstract approach which uses redundant information. Like the whole implementation of the community website's web application, this approach is based on Doctrine and hence works with other relational database systems as well. This way, the web application is kept independent of the underlying MySQL database system. Basically, the implemented approach views all graphs as they were directed acyclic graphs, including those which actually contain cycles. Thereby, the implementation uses *closure tables* to model these directed acyclic graphs and a *separate* table to store all cycles of those graphs which are not acyclic. This way, each cyclic remix graph is divided into its *acyclic part* and its *cyclic part*.

As described by Karwin, 2010, a closure table is a simple and elegant way of storing data hierarchies in a database table. It can be used to store all paths (i.e., not only those with a direct parent-child relationship) of any directed acyclic graph. In the context of the implementation of the database model, a path consists of three fields: an ancestor program field, a descendant program field, and a depth field. The depth field describes the distance between the ancestor program and the descendant program. For example, the distance between a parent program and its child remix is 1, the distance between a grandparent and its grandchild is 2, etc. Moreover, a closure table normally contains self-referencing relations. A self-referencing relation is a path where the ancestor is equal to the descendant, meaning the distance (or depth) is 0. This implementation also makes use of self-referencing relations in order to simplify the database queries (see Section 6.1.3). The same argument applies to all those *redudant* paths where $depth > 1$.

This implemented solution introduces the following three relational database tables to model all remix relations:

- *program_remix_relation* defines the closure table
- *program_remix_backward_relation* was used to store the cycles
- *scratch_program_remix_relation* contains all Scratch-Catrobat relations

Figure 6.6 shows an extract of the database model. To give a better overview, the *program* table was added to the diagram. It contains the records of all uploaded Catrobat programs and already existed before the project of this thesis. It should be noted that a new boolean field `remix_root` was added to the *program* table. This field is necessary to distinguish between root programs and remixed programs (see Section 6.1.1). Another implementation detail is that this field is also set to `true` for those converted programs which only have (one or many) Scratch programs but no Catrobat programs as parents. This simplifies the visualization algorithm described in Section 6.1.3.

The *program_remix_relation* closure table has three primary keys `ancestor_id`, `descendant_id`, and `depth`. This triple uniquely identifies the relationship between an ancestor program (`ancestor_id` is a foreign key referring to id field of the *program* table) and a descendant program (`descendant_id` is also a foreign key referring to id field of the *program* table). As already mentioned before, the `depth` field is needed to describe the distance between the ancestor

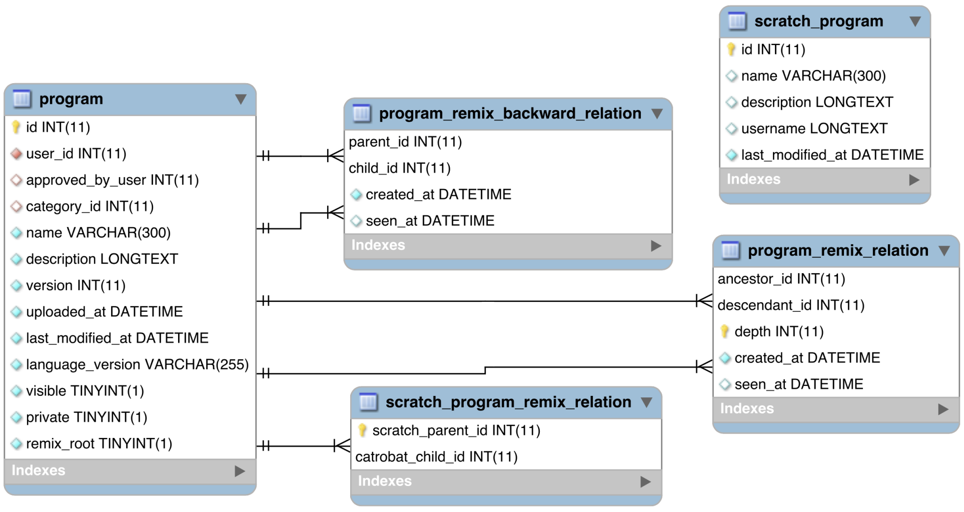


Figure 6.6.: Relations between database tables of the remix graph model
Created with MySQL Workbench: `http://www.mysql.com/products/workbench/`

program and the descendant program. For example, directed edges which describe a parent-child relation have a depth of 1, relations between grandparents and grandchilds are of depth 2, etc. It should be mentioned that a single descendant program may occasionally have multiple paths with different lengths to the same ancestor program. In this case, multiple records with the same `ancestor_id` and `descendant_id` but with different depth values have to be added to avoid loosing any relevant path information of the directed acyclic graph.

The *program_remix_backward_relation* table contains only the backward path of a cycle, which is always only a directed edge (i.e., depth is 1). Therefore, this table does not need a depth field. The backward edge is identified by two primary keys: `parent_id` and `child_id`. Both are foreign keys, each referring to the ID field of the *program* table. The forward path is part of the directed acyclic graph, meaning the relations of the forward path are stored in the *program_remix_relation* closure table.

The *scratch_program_remix_relation* contains all remix relations between the converted programs and their corresponding original Scratch program. Similar to the *program_remix_backward_relation* table, it only stores direct edges (depth is 1) and acts as an extension to the closure table. This separation decouples Scratch programs from Catrobat programs in the database and simplifies the model but requires additional queries in the remix graph fetching code (see next Section 6.1.3). The table contains two primary keys: `scratch_parent_id` and `catrobat_child_id`. While the `scratch_parent_id` field is only an integer field which contains the ID of the original Scratch program, the `catrobat_child_id` field is a foreign key which refers to the ID field of the converted program in the *program* table.

As already mentioned before, the implementation is based on Doctrine. Instead of designing the database model for the MySQL system, Doctrine provides a convenient way to model system-independent database models by defining entities. Basically, an entity is a PHP class which describes a database table by using annotations (declared with the ORM prefix). The Doctrine system provides tools which then check for all entity classes and automatically map them to the corresponding database table. To give an impression of how such an entity is implemented, a shortened version of the closure table entity's PHP code is shown in Listing 6.4.

```
<?php
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 * @ORM\Table(name="program_remix_relation")
 * @ORM\Entity(repositoryClass="Entity\ProgramRemixRepository")
 */
class ProgramRemixRelation
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer", nullable=false)
     */
    protected $ancestor_id;

    /**
     * @ORM\ManyToOne(targetEntity="Entity\Program",
         inversedBy="catrobat_remix_descendant_relations",
         fetch="LAZY")
     * @ORM\JoinColumn(name="ancestor_id", referencedColumnName="id")
     * @var \Catrobat\AppBundle\Entity\Program
     */
    protected $ancestor;

    /**
     * @ORM\Id
     * @ORM\Column(type="integer", nullable=false)
     */
    protected $descendant_id;

    /**
     * @ORM\ManyToOne(targetEntity="Entity\Program",
         inversedBy="catrobat_remix_ancestor_relations",
         fetch="LAZY")
     * @ORM\JoinColumn(name="descendant_id",
         referencedColumnName="id")
     * @var \Catrobat\AppBundle\Entity\Program
     */
    protected $descendant;

    /**
     * @ORM\Id
     * @ORM\Column(type="integer", nullable=false,
         options={"default" = 0})
     */
    protected $depth = 0;

    /**
     * @param \Catrobat\AppBundle\Entity\Program $ancestor
     * @param \Catrobat\AppBundle\Entity\Program $descendant
     * @param int $depth
     */
    public function __construct(Program $ancestor, Program
        $descendant, $depth)
    {
        $this->setAncestor($ancestor);
        $this->setDescendant($descendant);
        $this->setDepth($depth);
        // [...]
    }

    /**
     * @param \Catrobat\AppBundle\Entity\Program $ancestor
     */
    public function setAncestor(Program $ancestor)
    {
        $this->ancestor = $ancestor;
        $this->ancestor_id = $ancestor->getId();
    }

    /**
     * @param \Catrobat\AppBundle\Entity\Program $descendant
     */
    public function setDescendant(Program $descendant)
    {
        $this->descendant = $descendant;
        $this->descendant_id = $descendant->getId();
    }

    /**
     * @param int $depth
     */
    public function setDepth($depth)
    {
        $this->depth = (int) $depth;
    }
}
```

Listing 6.4: PHP Code of the closure table's entity class

Data Retrieval and Visualization

This section discusses the visualization of remix graphs. Basically, remixing can be understood as a concept which relies on basing a remixed program on a single or multiple parent programs. Consequently, the remixed program automatically inherits the complete remix history of these parent programs (including parts of their content) as well. Thus, all remix-related programs (including parent to child, grandparent to grandchild, etc.) are very likely to contain similar content.

Remix graphs are mathematical structures which describe all relations between remix-similar programs and hence can be viewed as clusters of content-similar programs. Each cluster (i.e., remix graph) contains programs which are similar to each other but also dissimilar to most programs from other clusters. That means programs of remix graphs contain relevant information which can be used for user recommendations. In other words, users who are interested in a program might also be interested in its remixed programs and vice versa. From a recommender systems perspective, this corresponds to a *content-based* recommendation approach (see Section 4.4.1). Hence, it would be quite desirable for Catrobat's community website to visualize these remix graphs and present their remix-related programs to the user. Actually, this approach was implemented as part of the project of this thesis.

The implementation is divided into two parts: a server part and a web client part. The server provides a REST API for the web client and is responsible for fetching the relevant graph data from the database. A code snippet of the implemented REST API method is presented in Appendix B. The web client sends a HTTP request to the REST API and is responsible for visualizing the delivered data.

Server Part As already mentioned in Section 6.1.3 the database model divides each remix graph into its acyclic forward paths and, if existing, the backward paths of its cycles. The acyclic part is stored in the *program_remix_relation* closure table and the backward paths in the *program_remix_backward_relation* table. In addition, the *scratch_program_remix_relation* table contains direct edges to Scratch parents. The records of these three tables can be retrieved either via MySQL-specific SQL queries or via platform-independent Doctrine queries.

In order to keep the implementation platform-independent, the latter variant was chosen. The aforementioned closure table contains not only direct edges but also redundant paths, including all forward paths (i.e., $depth > 1$) and a self-referencing path for every Catrobat program (i.e., $depth = 0$).

Considering all this redundancy, the algorithm for fetching a complete cyclic remix graph can be kept very clear and simple. After fetching the complete remix graph, the data is returned. Thereby, the Doctrine queries of the algorithm take full advantage of the forward and self-referencing paths. For reasons of clarity and comprehensibility, a (slightly) simplified version of the algorithm is described. In contrast to the implemented version, this simplified version does not cover very flat graphs. Given any program of the remix graph as start program, the simplified algorithm works as follows:

i. **Retrieve its root programs:**
Use the ID of the given start program and retrieve all of its Catrobat ancestor programs by only taking the acyclic part of the graph from the *program_remix_relation* table into account. Some (at least one) of these ancestors must be root programs. Root programs can be found by checking which of these ancestor programs are marked as `true` in the `remix_root` boolean field of the program table (see Section 6.1.3). Both steps can be combined in one single query (costs: 1 Doctrine query). Even in the case that the start program is already a root program, this will still work since the query would only return the self-referencing query which contains the given program as a root program.

ii. **Fetch all Catrobat programs of the complete graph:**
Find all Catrobat programs of the graph. These are nothing less than the descendant programs of all retrieved root programs. Thus, these descendants can be fetched with a single Doctrine query by simply passing the root program IDs to the query (costs: 1 Doctrine query). Again, the self-referencing relations are quite useful here as well, as the root programs are automatically included in the query result, meaning the result contains every path and every Catrobat program of the full graph.

iii. **Get direct forward edges:**
In order to visualize the remix graph, the directed forward edges between all Catrobat programs have to be determined. This requires another query (costs: 1 Doctrine query).

iv. **Fetch direct backward edges:**
Given all the IDs of the previously fetched Catrobat programs, the backward edges of each program can be retrieved from the *program_remix_backward_relation* table via a single query (costs: 1 Doctrine query). In case the graph is acyclic, the result will be empty. It is critical to note that these direct backward edges can never contain any new Catrobat programs as a backward edge can only exist if a forward path exists as well.

v. **Fetch Scratch programs:**
Some programs in the graph might be converted programs which are a remix-child of a Scratch program. Given all the IDs of all Catrobat programs, any directed edge between a converted program and an original Scratch program can be fetched from the *scratch_program_remix_relation* table via a single query (costs: 1 Doctrine query). In case no Catrobat program is a converted Scratch program, the result will be empty.

vi. **Compose and return the data:**
Finally, all retrieved programs and paths are composed and returned to the client (no queries needed).

Overall this simplified algorithm *constantly* requires only five database queries in order to fetch the complete graph regardless of whether the graph is very small (e.g., only one node) or very big (hundreds or thousands of nodes). Listing 6.5 shows a shortened version of the simplified algorithm implemented in PHP. Additional comments were added to the code which mark each of the above described steps.

```php
<?php
// [...]
public function getFullRemixGraph($program_id)
{
  $catrobat_ids = [$program_id];
  $prev_descendant_ids = $catrobat_ids;

  // Step (i.) - Retrieve root programs
  $root_ids = $remix_repository->getRootProgramIds($catrobat_ids);

  // Step (ii.) - Fetch all Catrobat programs of complete graph
  $catrobat_ids = $remix_repository->getDescendantIds($root_ids);

  $diff_new = array_diff($catrobat_ids, $prev_descendant_ids);
  $diff_previous = array_diff($prev_descendant_ids, $catrobat_ids);
  $diff = array_merge($diff_new, $diff_previous);
  sort($catrobat_ids);

  // Step (iii.) - Get direct forward edges
  $catrobat_forward_edge_relations = $this
    ->program_remix_repository
    ->getDirectEdgeRelationsBetweenProgramIds($catrobat_ids);

  $catrobat_forward_edge_data = array_map(function ($r) {
    return [
      'ancestor_id' => $r->getAncestorId(),
      'descendant_id' => $r->getDescendantId(),
      'depth' => $r->getDepth()
    ];
  }, $catrobat_forward_edge_relations);

  // Step (iv.) - Fetch direct backward edges
  $catrobat_backward_edge_relations = $this
    ->program_remix_backward_repository
    ->getDirectEdgeRelations($catrobat_ids, $catrobat_ids);

  $catrobat_backward_edge_data = array_map(function ($r) {
    return [
      'ancestor_id' => $r->getParentId(),
      'descendant_id' => $r->getChildId()
    ];
  }, $catrobat_backward_edge_relations);

  // Step (v.) - Fetch Scratch programs
  $scratch_edge_relations = $scratch_remix_repository
    ->getDirectEdgeRelationsOfProgramIds($catrobat_ids);

  $scratch_node_ids = array_values(array_unique(array_map(
    function ($r) {
      return $r->getScratchParentId();
    }, $scratch_edge_relations
  )));

  sort($scratch_node_ids);

  $scratch_edge_data = array_map(function ($r) {
    return [
      'ancestor_id' => $r->getScratchParentId(),
      'descendant_id' => $r->getCatrobatChildId()
    ];
  }, $scratch_edge_relations);

  // Step (vi.) - Compose and return the data
  return [
    'catrobatNodes' => $catrobat_ids,
    'scratchNodes' => $scratch_node_ids,
    'catrobatForwardEdgeRelations' => $catrobat_forward_edge_data,
    'catrobatBackwardEdgeRelations' => $catrobat_backward_edge_data,
    'scratchEdgeRelations' => $scratch_edge_data
  ];
}
```

Listing 6.5: Implementation for fetching the complete remix graph

Client Part The web client was implemented in JavaScript. It uses the popular jQuery [2] library to fetch the graph data from the server's REST API via AJAX requests and the vis.js [3] library to visualize the graph. In addition, jQuery contextMenu[4], a jQuery plugin for showing context menus, was used. To limit the scope of this thesis only the implementation of the visualization part is briefly discussed in the following sections.

The visualization is completely taken over by the very powerful vis.js library.

[2]jQuery library: http://jquery.com
[3]vis.js library: http://visjs.org
[4]jQuery contextMenu: http://swisnl.github.io/jQuery-contextMenu/index.html

Vis.js provides a rich set of powerful features and configuration settings. The configuration options can be used to define the style, order, and arrangement of the nodes and edges in a graph. Furthermore, the library provides control buttons and already supports scrolling and zooming. In addition, event listener methods can be implemented in order to adapt the behavior of a graph. These event listener methods receive events which are basically user interactions detected by the library. For the project of this thesis, some event listener methods had to be implemented. The library had to be configured appropriately and the retrieved data of the remix graph had to be prepared and converted into a suitable format for the library.

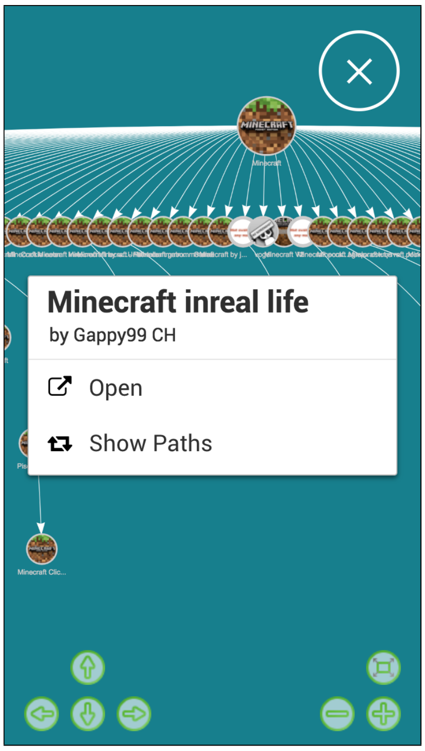


Figure 6.7.: Zoomed-in view of a remix graph

Figure 6.7 demonstrates a zoomed-in view of the visualization after the user clicked on a program-node of the graph. At the bottom of the figure, the library shows control buttons for navigation (arrow buttons) and for zooming (plus, minus, and reset zoom buttons). The figure also shows a popped up context menu that is presented to the user after the program-node is clicked. It contains detailed information about the clicked program and provides options to select which action should be performed next.

6.2. Implementation of Like Rating System

Another task of the practical part of this thesis was to implement a *like rating system* for Catrobat's community website. The goal of the rating system is to collect user preferences which will later serve as an input source for an alternative collaborative filtering approach in Section 6.3. Basically, a like rating system is a popularity measuring system that offers only a single option to the user, i.e. to *like* an item. In contrast to a like/dislike system which also allows users to dislike an item, a like rating system collects only positive feedback. Such like-only rating systems are quite common on social networks, where dislike ratings could have a negative impact on the overall user satisfaction. A good example of a like-only rating system can be found on Facebook[5]. In addition to a normal like button which is shown as a "thumbs up" icon, Facebook supplements its thumbs up button with a variety of different emoji buttons.

The implementation of the like rating system in Catrobat's community website is mainly based on Facebook's idea and follows a similar approach. The like buttons are shown on the detail page of every uploaded Catrobat program. Thereby, the implemented system lets its users give feedback for a Catrobat program by using *one* of four different emoticon buttons, including "thumbs up", "laugh", "love", and "wow".

Figure 6.8 gives an impression of how the rating system is presented to the user. The four different buttons are shown below the screenshot of the Catrobat program. In this example, the logged-in user has liked the program "Minecraft" by clicking on the "thumbs up" button (see white-colored button). The total number of likes (i.e. the sum of all "thumb up", "laugh", "love", and "wow" clicks) is shown on the right-hand side of the figure (in this case, it

[5]Facebook: `http://www.facebook.com`

is 52). Below, the number of people is shown who also clicked on the same button as the logged-in user (in this case, it is 32 and refers to the number of people who gave thumbs up on the Minecraft program).

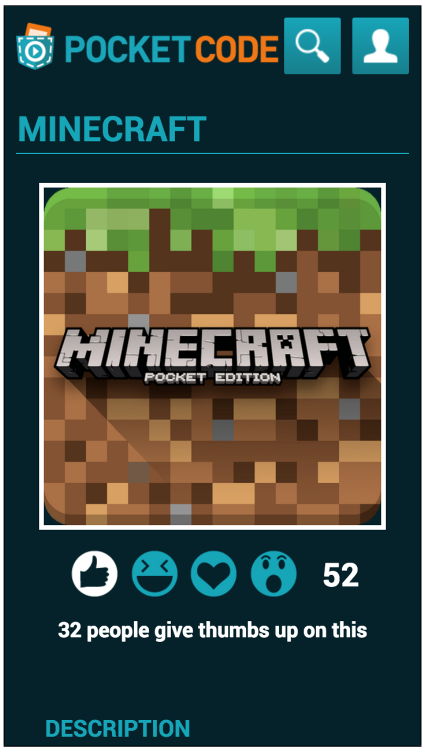


Figure 6.8.: Detail page of a Catrobat program showing the 4 different like buttons

In order to track users' likes, a new Doctrine entity (PHP class) was added to the web application of the community website. This entity class introduces a new database table called *program_like* which models a many-to-many relationship between the *program* table and Symfony's user table, labeled as *fos_user* table. For reasons of brevity, the code of the entity class is not presented and only the *program_like* table is briefly described.

Figure 6.9 shows the relations between these database tables. The `program_like` table has two primary keys called `program_id` and `user_id`. Both primary keys are also foreign keys; the `program_id` field refers to the `id` field of the *program* table and the `user_id` field refers to the `fos_user` table.

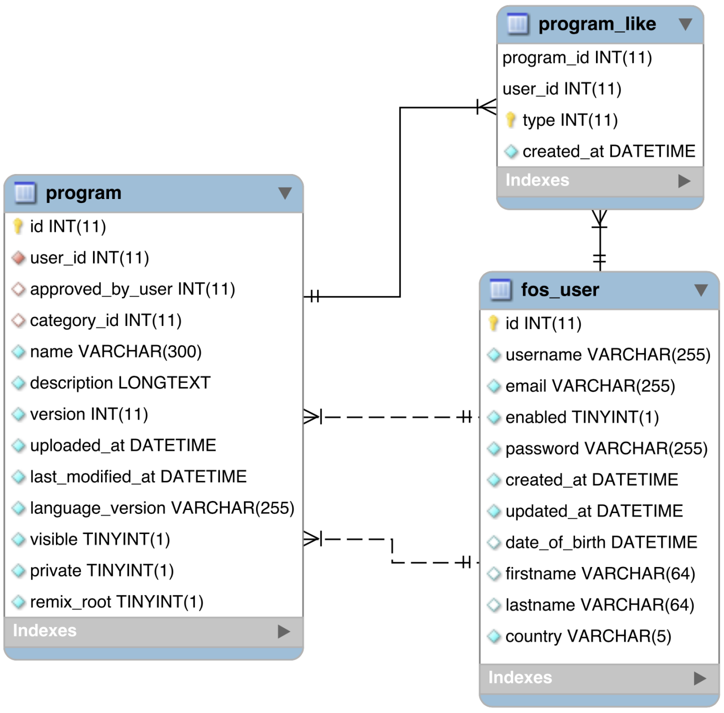


Figure 6.9.: Database relations between *program_like*, *program* and *fos_user* table
Created with MySQL Workbench: `http://www.mysql.com/products/workbench/`

6.3. Integration of Recommender System

Now that all functionality for gathering collective user data has been explained, the implementation of the two different recommendation approaches can be discussed. Both approaches are based on *user-based* collaborative filtering, which has been described in Section 4.4.2. The first approach is based on the active user's remixes (introduced in Section 6.3.3) and the second is based on the active user's likes (introduced in Section 6.3.3).

The reasons for using a user-based collaborative approach was mainly based on the following aspects:

- **Collaborative Data:**
  Catrobat's community website is a groupware system primarily driven by collaborative user data. Hence, a collaborative filtering system would be a suitable approach.

- **Serendipitous Programs:**
  According to Schafer, Frankowski, et al., 2007, a collaborative filtering approach would be able to discover more "serendipitous" items, i.e. Catrobat programs.

- **Number of Users vs. Number of Programs:**
  The difference between the number of users and the number of Catrobat programs on the community website is very small. Hence, a user-based approach would be more appropriate than using an item-based technique.

- **Small Amount of Descriptive Content:**
  Due to the nature of content-based approaches, they would either require existing keywords/tags or *descriptive* content information. Usually, Catrobat programs do not contain much descriptive data (mainly, program title, username, and short description).

The implemented approaches recommend yet unseen Catrobat programs to the active user which have been liked by other taste-similar users. These recommendations are shown on the home page of the community website (see A/B Tests in Section 6.4).

6.3.1. Precomputation of User Similarities

Taste-similar users are those users who share a similar remix taste (i.e., first approach; see Section 6.3.3) or similar like taste (i.e., second approach; see also Section 6.3.3) with the active user. Thereby, a binary user vector is analyzed where each vector element refers to a Catrobat program and indicates whether a program has been remixed/liked (i.e., 1) or not (i.e., 0). This type of binary input data can be treated as implicit data (see Section 4.2) regardless of the used approach. In order to find such taste-similar users, the similarity between the user vectors of each user pair has to be computed first. User similarities can be computed by using an appropriate metric formula. A variety of different metrics to measure similarities between two users has been discussed in Section 4.4.2. Most metrics are intended to be used primarily for non-binary data. However, the implemented approaches have to deal with binary datasets and hence, a more suitable measure for binary data is needed. One such metric is the Jaccard distance (defined in Formula 4.17), which has been used by both implemented user-based recommendation approaches. In case of the first approach, the Jaccard distance is calculated based on the common remixed programs of two users for each single user pair. In contrast to the first approach, the second approach uses the Jaccard metric to compute the similarity based on the common liked programs of two users for each single user pair.

An illustrative example on how the Jaccard distance computation works for the second approach has been given in Figure 4.5 of Section 4.4.2. The example under discussion is the one depicted on the left side of the figure and described as "user-based collaborative filtering".

At the time of implementing the recommender system, there were more than 30,000 registered users on the Catrobat community website and hence about 450 million similarities of potential user pairs that would have to be calculated (i.e., $\binom{30000}{2} \approx 450$ million). However, in practice, only users who have remixed (first approach) or liked (second approach) at least one Catrobat program can be similar and hence only these users are relevant. This also drastically reduces the number of user pairs. Even though the computation of all relevant user pairs might be somewhat manageable, it would still take a great length of time to complete. This makes the recommender system ineligible for generating instant recommendations every time a user sees the home page of the community website. Hence, a cronjob script had to be implemented in order to precompute the similarities and store the results for later use. For reasons of performance, the similarity computation was directly implemented in Java. The Java application is automatically executed by the script and the script is supposed to be run in the background periodically in order to keep the similarities up to date. To give an impression of the

Java implementation a short code snippet was extracted and is shown in Appendix C.

After the Java program is finished, the generated SQL file is imported into the MySQL database of the community website. For this purpose, two new Doctrine entity classes were added to the web application which define two database tables, *user_remix_similarity_relation* and *user_like_similarity_relation*. Figure 6.10 visualizes the used database model. Aside from the different table names, both tables are identically equal. Each of them has two primary keys, `first_user_id` and `second_user_id`, which uniquely identify a user pair and are both foreign keys referring to the `id` field of the `fos_user` table. In addition, the `similarity` field contains the decimal value of the calculated Jaccard distance of the respective user pair and the `created_at` field stores the timestamp when the distance was last updated (i.e, time of last run of cronjob).

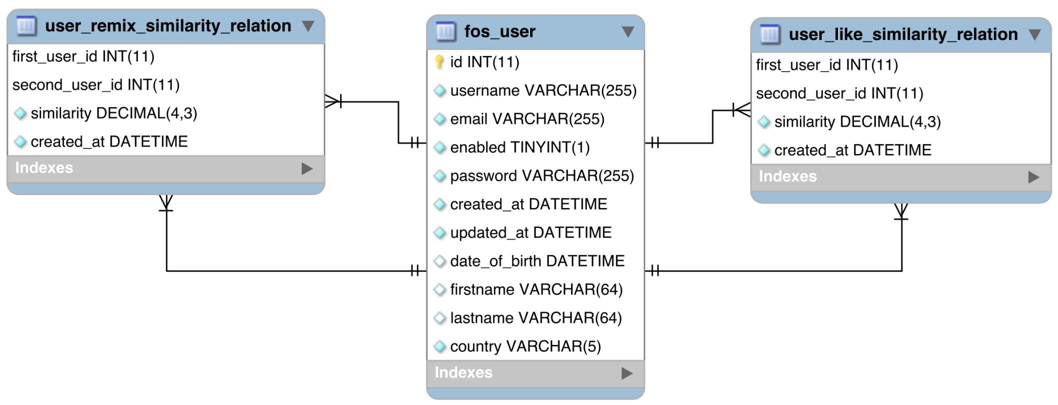


Figure 6.10.: Database relations between *user_remix_similarity_relation*, *user_like_similarity_relation* and *fos_user* table
Created with MySQL Workbench: `http://www.mysql.com/products/workbench/`

6.3.2. Framework of the Recommender System

Basically, the implemented recommender system itself can be viewed as a small framework which allows a dynamic switch between different recommendation algorithms. Both implemented recommendation approaches were directly integrated into this framework. Unlike the precomputation of similarity values, the recommender system and both recommendation approaches were realized in PHP. For this task, a REST API was implemented for the Server and the JavaScript code of the community website's home page had to be properly adapted and extended. Both the implementation details of the server-side REST API as well as the client-side JavaScript code are not relevant for this thesis and are hence not discussed. However, a brief overview of the communication flow is given.

The API provides unique URLs for each recommendation approach, which can be called via AJAX requests by the JavaScript client to invoke the algorithm of the corresponding approach. A reference to the implemented REST API method is given in Appendix B. The respective algorithm then generates a list of recommended Catrobat programs based on the used approach and returns the results to the recommender. Then, the recommender serializes this list to JSON data and sends it to the client. Finally, the recommended list of Catrobat programs is presented to the active user by the JavaScript client (see figures in Section 6.4).

As already mentioned before, the framework is able to dynamically switch between different recommendation algorithms. In the following sections, the algorithms of the two recommendation approaches are discussed in more detail.

6.3.3. Implementation of Recommendation Approaches

User-based Collaborative Filtering Approach based on Remixing

The first implemented approach represents a user-based collaborative filtering algorithm. This approach generates a list of Catrobat programs for the active user which have been remixed by other remix-similar users. The approach is based on a collaborative filtering approach which has been previously proposed for Scratch by Fessakis and Dimitracopoulou, 2009.

Listing 6.6 presents the PHP code of the implemented algorithm which is called by the recommender system. The code was slightly reformatted and comments were added in order to increase the readability. Given the pre-computed remix similarity values from Section 6.3.1 for all relevant user pairs, users who are similar to the active user can be easily determined by the recommendation algorithm in time. This can be achieved by fetching the user-pairs with the highest similarities from the *user_remix_similarity_relation* table where `first_user_id` or `second_user_id` is equal to the active user's ID (see Figure 6.10). Each of these obtained user-pairs contains the ID of another remix-similar user. These remix-similar user IDs are then used in order to find a list of such Catrobat programs which have been remixed by these similar users but have not been remixed by the active user yet. Then weights for each list's program are determined based on how similar the remix-similar user is to the active user. In cases where more than one similar user have remixed the same program, the similarity value of each user is added to the weight of that program. This weighting scheme is based on Formula 4.2 for non-binary user vectors. In contrast to the Formula 4.2, the formula which is actually used ignores the denominator. Finally, the list is sorted based on these weights and returned to the recommender system.

```
<?php
// [...]

public function recommendProgramsOfRemixSimilarUsers($user, $flavor)
{
  // Find remix-similar users and add their user ID
  // and similarity to the $similar_user_mapping array
  $user_similarity_relations = $this
      ->user_remix_similarity_relation_repository
      ->getRelationsOfSimilarUsers($user);

  $similar_user_mapping = [];

  foreach ($user_similarity_relations as $r) {
    $id_of_similar_user = ($r->getFirstUserId() != $user->getId())
                        ? $r->getFirstUserId()
                        : $r->getSecondUserId();
    $similar_user_mapping[$id_of_similar_user] = $r->getSimilarity();
  }

  // Fetch all programs remixed by the active user and add their
  // program IDs to the $excluded_ids_of_remixed_programs array
  $parent_relations_of_all_remixed_programs_of_user = $this
      ->program_remix_repository
      ->getDirectParentRelationDataOfUser($user->getId());

  $ids_of_similar_users = array_keys($similar_user_mapping);
  $excluded_ids_of_remixed_programs = array_unique(array_map(
    function ($data) {
      return $data['ancestor_id'];
    }, $parent_relations_of_all_remixed_programs_of_user)
  );

  // Fetch all programs remixed by similar users and exclude all
  // programs that appear in $excluded_ids_of_remixed_programs
  $relations_of_differing_parents = $this
    ->program_remix_repository
    ->getDirectParentRelationsOfUsersRemixes(
        $ids_of_similar_users,
        $user->getId(),
        $excluded_ids_of_remixed_programs,
        $flavor);
  // Weight the fetched programs based on how similar
  // the similar user is to the active user
  $weights = [];
  $programs_remixed_by_others = [];
  foreach ($relations_of_differing_parents as $parent_relation) {
    $key = $parent_relation->getAncestorId();
    assert(!in_array($key, $excluded_ids_of_remixed_programs));

    if (!array_key_exists($key, $weights)) {
        $weights[$key] = 0.0;
        $program = $parent_relation->getAncestor();
        $programs_remixed_by_others[$key] = $program;
    }

    $id_of_similar_user = $parent_relation
        ->getDescendant()
        ->getUser()
        ->getId();

    $weights[$key] += $similar_user_mapping[$id_of_similar_user];
  }

  // Sort the weights
  arsort($weights);

  // Sort and return the list of recommended programs
  return array_map(
    function ($program_id) use ($programs_remixed_by_others) {
      return $programs_remixed_by_others[$program_id];
    }, array_keys($weights)
  );
}
```

Listing 6.6: Implementation of user-based collaborative filtering based on remixing

User-based Collaborative Filtering Approach based on User Likes

The second implemented approach represents a user-based collaborative filtering algorithm based on common likes between the active user and other users. Listing 6.7 shows the PHP code of the implemented algorithm called by the recommender system. Similarly to the first approach, the precomputed remix similarity values from Section 6.3.1 for all relevant user pairs are used to quickly find like-minded users. Thereby, the user-pairs with the highest similarities are fetched from the *user_like_similarity_relation* table where `first_user_id` or `second_user_id` is equal to the active user's ID (see Figure 6.10). Each of these obtained user-pairs contains the ID of another like-similar user. Based on the liked programs of these similar users, a program list is created which includes Catrobat programs that have not been liked by the active user yet. Subsequently, weights for each program are computed based on how similar the like-similar user is to the active user. In cases where more than one similar user have liked the same program, the total weight of that program includes the similarity value of each of these users. To phrase it differently, the described weighting scheme uses Formula 4.2 but ignores the denominator. Finally, before the list can be returned to the recommender system, it has to be sorted based on these weights.

```php
<?php
// [...]

public function recommendProgramsOfLikeSimilarUsers($user, $flavor)
{
  // Fetch all programs liked by the active user and add their
  // program IDs to the $excluded_ids_of_liked_programs array
  $all_likes_of_user = $this
      ->program_like_repository
      ->findBy(['user_id' => $user->getId()]);

  $excluded_ids_of_liked_programs = array_unique(array_map(
    function ($like) {
      return $like->getProgramId();
    }, $all_likes_of_user)
  );

  // Find like-similar users and add their user IDs
  // to the $ids_of_similar_users array
  $user_similarity_relations = $this
      ->user_like_similarity_relation_repository
      ->getRelationsOfSimilarUsers($user);

  $similar_user_mapping = [];

  foreach ($user_similarity_relations as $r) {
    $id_of_similar_user = ($r->getFirstUserId() != $user->getId())
                        ? $r->getFirstUserId()
                        : $r->getSecondUserId();
    $similar_user_mapping[$id_of_similar_user] = $r->getSimilarity();
  }

  $ids_of_similar_users = array_keys($similar_user_mapping);

  // Fetch all programs liked by similar users and exclude all
  // programs that appear in $excluded_ids_of_remixed_programs
  $differing_likes = $this->program_like_repository->getLikesOfUsers(
      $ids_of_similar_users,
      $user->getId(),
      $excluded_ids_of_liked_programs,
      $flavor
  );
  // Weight the fetched programs based on how similar
  // the similar user is to the active user
  $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, $weights)) {
      $weights[$key] = 0.0;
      $recommended_program = $differing_like->getProgram();
      $programs_liked_by_others[$key] = $recommended_program;
    }

    $id_of_similar_user = $differing_like->getUserId();
    $weights[$key] += $similar_user_mapping[$id_of_similar_user];
  }

  // Sort the weights
  arsort($weights);

  // Sort and return the list of recommended programs
  return array_map(
    function ($program_id) use ($programs_liked_by_others) {
      return $programs_liked_by_others[$program_id];
    }, array_keys($weights)
  );
}
```

Listing 6.7: Implementation of user-based collaborative filtering based on likes. The PHP code was slightly reformatted and comments were added in order to increase readability.

6.3.4. Common Challenges

The implemented system aims to tackle some of the most common challenges that have been discussed in Section 4.4.3.

- **Scalability and Limited Computational Power:**
  The computation of the user similarities has been decoupled from the web application of Catrobat's community website and was implemented in Java. The computation can be performed in advance and periodically as a cronjob task (see Section 6.3.1). Moreover, the Java implementation already takes advantage of efficient data structures and can be still be adapted for parallel computation.

- **Cold Start Problem:**
  A very common problem for recommender systems is the cold start problem. Due to the nature of the two implemented recommendation approaches, both work only when the active user is logged-in on the community website. This is because remixing data and like ratings only exist for logged-in users. This user scenario describes the new user problem. The same also applies to newly registered users who have not liked or remixed any programs before. As already mentioned in Section 6.3.2, the recommender system is able to dynamically switch between different algorithms. In order to tackle the "new user" cold start problem, the system recommends the most remixed or most like programs to its guest or newly registered users. This is a very simple and more general recommendation approach.

- **Shilling attacks:**
  By using positive-only ratings, shilling attacks are very unlikely to happen in Catrobat. However, the like rating system has been limited to allow only logged-in users to like a Catrobat program. Further, multiple ratings per program are not possible, meaning each program can only be liked once by a user.

- **Similar and Duplicate Items:**
  The Pocket Code app plans to restrict its users from uploading remixes that are identical equal to the downloaded parent program. This should

reduce the amount of duplicate programs. However, there is still plenty of room left for improvement. More efforts would also be needed to further reduce the amount of similar programs on the community website.

6.4. Evaluation of Recommendation Approaches

Basically, there are three evaluation methods for recommender systems: offline evaluation, online evaluation, and user studies (see Section 4.5.2). In order to evaluate the research questions, three A/B tests have been conducted (see Section 4.5.2). The reason for using A/B testing over other evaluation methods was mainly based on the arguments explained in Section 4.5.4. The online experiment was conducted from March 7th, 2017 to May 7th, 2017. Each of the A/B tests used consisted of different versions: version A, version B, and version C. As a result, the version shown to each user was solely based on the preconfigured language of the user's browser.

6.4.1. Test Scenario I

The first test scenario was designed to answer the first research question:

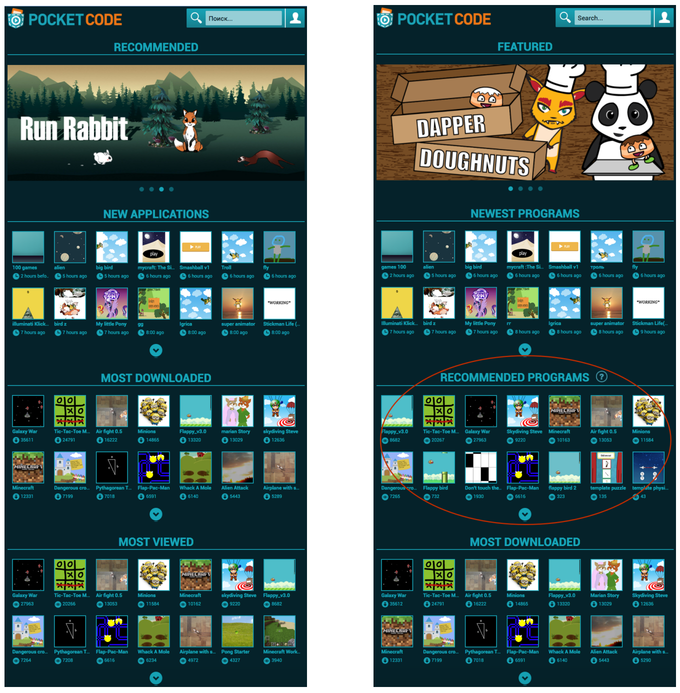
> *"Do recommendations of Catrobat programs have a positive impact on the overall download activity?"*

In this scenario, an A/B test with three different versions was implemented: version A, version B, and version C. Basically, the goal of this test was to monitor the download activity of version A (no recommendations), of version B (recommendations based on likes), and of version C (recommendations based on remixes). The controlled variable of this test was a new section (named as "Recommended programs") which was added to the community website's home page (see Figure 6.11). The new section was filled with recommended Catrobat programs and is described as the recommender system's framework in Section 6.3.2. In this test, version A acts as the control version, refers to the original version of the community website's home page, and hides the section. Version B and version C are both variation versions and show the section but include a list of *different* recommendations in the section. Version B uses the second recommendation approach (see Section 6.3.3) to include a list of Catrobat programs liked by like-minded users. Finally, the first recommendation approach which was discussed in Section 6.3.3), recommends Catrobat programs in version C which have been remixed by remix-similar users. The user's allocation to the different versions was based on the browser language of the respective user; the used allocation setting is summarized in Table 6.1.

	Version A	**Version B**	**Version C**
Recommendations	no	yes, based on likes	yes, based on remixes
Browser Language	Russian	German	any, except Russian, French, and German

Table 6.1.: Matrix of the first A/B test's allocation setting

As has already been discussed in Section 6.3.4, it is important to note that user-based recommendations in version B and C can only be made to logged-in users who have like-similar (version B) or remix-similar users (version C). This requires having at least one Catrobat program being remixed or liked by the active user. In addition, this program must also be remixed/liked by another user who has remixed/liked other programs (i.e., programs that can be recommended) as well. In all other cases during this online experiment, users received general recommendations (most liked programs in version B and most remixed programs in version C; see *Cold Start Problem* in Section 6.3.4).



(a) Home page without recommendations

(b) Home page with recommendations

Figure 6.11.: Home page of Catrobat's community website hiding (a) and showing (b) the section of recommended programs

6.4.2. Test Scenario II

This scenario tackles the second research question:

> *"Can a user-based recommendation approach based on collaborative filtering produce recommendations of higher quality compared to those generated by naive recommendation approaches? Which of the two implemented user-based collaborative filtering approaches performs best: (a) the first approach, which is based on remixing data, or (b) the second approach, which is based on like ratings?"*

In other words, the goal of the second test scenario was to show which of the two recommendation approaches performs best and to measure the quality of the recommendations. Although this can be achieved by using the same A/B test as described in test scenario I (see Section 6.4.1), it would be reasonable to define a baseline and to conduct a second A/B test. In this A/B test, the baseline was a naive recommendation approach which recommends random Catrobat programs. The A/B test was based on the A/B test of test scenario I, however, version A of the previous test had to be replaced by the baseline version. In order to avoid influencing the results of other tests, the baseline version was only shown to French-speaking users. The used allocation setting used is shown in Table 6.2.

	Version A	**Version B**	**Version C**
Recommendations	yes, random programs (baseline)	yes, based on likes	yes, based on remixes
Browser Language	French	German	any, except Russian, French, and German

Table 6.2.: Matrix of the second A/B test's allocation setting

6.4.3. Test Scenario III

The last test scenario intended to answer the third research question:

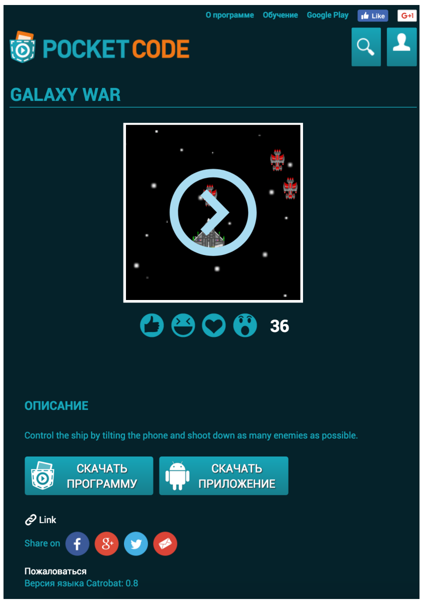
> *"Given the set of Catrobat programs remixed by all users, can these remixed programs be used to generate high quality content-based recommendations for other users in order to increase the overall remixing activity?"*

To phrase it differently, the goal was to figure out whether or not the recommendation of remix-related Catrobat programs made by the remix graph visualization tool represents an appropriate recommendation alternative to promote remixing in Catrobat. These suggested remixed programs can be viewed as content-based recommendations (see Section 6.1.3). The implemented A/B test used two versions: version A and version B. In this test, a "Show Remix Graph" button on the program detail page acted as a control variable or to phrase it differently, the button was shown (version A) or hidden (version B) on the detail page of every Catrobat program. Every time a user clicked on the "Show Remix Graph" button, the implemented JavaScript function of the graph visualization tool (see *Client Part* in Section 6.1.3) was triggered and it opened a view of the current program's remix graph in fullscreen which includes all remix-related Catrobat and Scratch programs of the current program. Even though, Scratch programs were also shown in the remix graph, the focus of this A/B test was only on Catrobat remixes.

Figure 6.12 shows the program detail page with and without the button. The used allocation setting is summarized in Table 6.3.

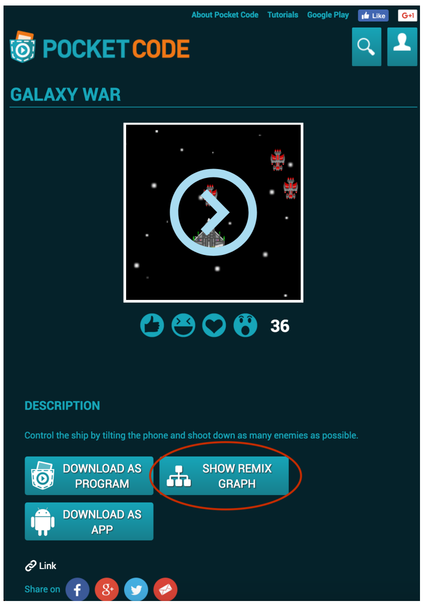
	Version A	**Version B**
Recommendations	no	yes, remix graph
Browser Language	Russian	any, except Russian, French and German

Table 6.3.: Matrix of the third A/B test's allocation setting



Control the ship by tilting the phone and shoot down as many enemies as possible.

(a) Program detail page without "Show Remix Graph" button (version A)



Control the ship by tilting the phone and shoot down as many enemies as possible.

(b) Program detail page with "Show Remix Graph" button (version B)

Figure 6.12.: Program detail page of Catrobat's community website hiding (a) and showing (b) the "Show Remix Graph" button

To summarize, this chapter explained important implementation details of the practical part of this Master's thesis. It discussed all relevant parts of the implemented source code and the designed database models which were necessary to collect collaborative input data for the recommender system. Subsequently, the implementation of two recommendation approaches, the first one based on like ratings and the second on remix data, was described in more detail. Finally, the last section of the chapter presented the conducted online tests which were needed to solve the research questions and to evaluate the performance of both approaches.

7. Results

Given the research goals and the used evaluation protocol specified in Chapter 6, the following sections of this chapter are intended to present the final evaluation results of this project. To better understand the achieved evaluation results, general statistics about Catrobat's community platform (see Section 5.4) and the used online experiments are given below. Subsequently, all evaluation results of the conducted A/B tests are presented. The results are discussed in more detail at the end of this chapter.

7.1. General Statistics of the Community Website

This section explains key characteristics of Catrobat's community website and provides the foundation for the rest of the chapter. Key performance indicators of Catrobat's online community are listed in the following section.

7.1.1. Key Performance Indicators of Catrobat

Table 7.1 summarizes the characteristic data which was directly retrieved from MySQL database of the community website at the time of writing this thesis. At that time, the community website had over $30,000$ registered users and over $28,000$ uploaded Catrobat programs. As described in Chapter 6, user preferences were collected via like ratings and remix data was extracted from uploaded Catrobat programs. Thus far, a total of 1047 users (about 3.4% of all users) have liked 1996 programs in total and 3663 users (about 11.9%) have remixed 6866 Catrobat programs. Moreover, 27 users have converted and uploaded a Scratch program (i.e., remix of a Scratch program).

Number of users	30713
Number of programs	28439
Number of downloads	298037
Number of Catrobat remixes	6866
Number of Scratch remixes	49
Number of likes	1996
Number of users engaged in remixing	3663
Number of users engaged in like ratings	1047

Table 7.1.: Key performance indicators of Catrobat's online community (last update: 2017-05-07)

7.1.2. Distribution of Catrobat Programs

Figure 7.1 illustrates the distribution of all uploaded Catrobat programs based on remixes. About 75.7% of all Catrobat programs are not based on any other program, 24.1% of all programs are Catrobat remixes, and 0.2% represent converted Scratch programs.

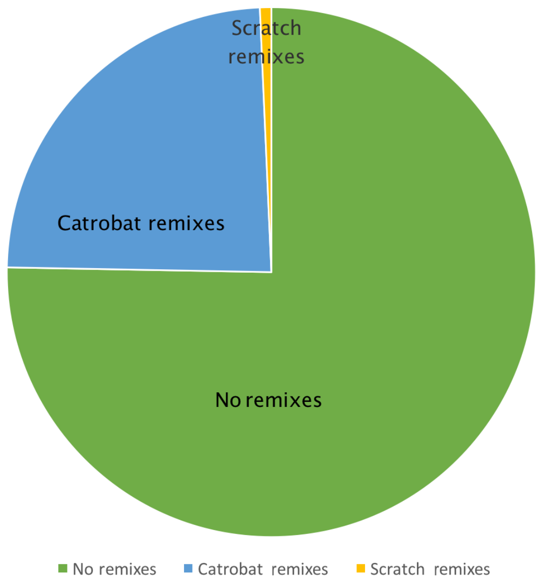


Figure 7.1.: Distribution of Catrobat programs based on remixes

7.2. Evaluation Results

This section presents the results of all test scenarios described in Section 6.4. All online experiments were run over a two month period, starting on March 7th, 2017, and ending on May 7th, 2017.

7.2.1. Results of Test Scenario I

In order to evaluate the impact of recommendations on the overall download activity, an A/B test was conducted. The A/B test consists of three different versions (version A, B, and C) and is described in Section 6.4.1. At the end of the test period, the following measures were obtained for each version:

- Number of visits of the community website's home page
- Number of conversions

The number of conversions was defined as the number of downloads. This number includes all downloads of Catrobat programs. Both of these measures were separated by the language of the user's browser corresponding to the respective version (A, B, or C) of the test and stored for each version. The number of visits were automatically tracked via Google Analytics[1]. Further, the number of downloads was kept up to date by the web application of the community website. Figure 7.2 depicts three diagrams; each diagram shows the number of downloads/conversions of the corresponding version on the y-axis and the date on the x-axis. The first graphic represents the number of downloads of version A, the second graphic shows the number of downloads of version B, and the third graphic represents the number of downloads of version C. In addition, all graphs include a dotted line which best fits all individual data points. This linear line reflects the long-term movement of the downloads in the given time series and is hence called the trend-line. The calculation of the trend-line is based on linear regression. The slope of version A's trend-line is slightly negative to stagnant throughout the full test period. In sharp contrast, the trend-lines of version B and C indicate a noticeable increase of the number of downloads/conversions throughout the test period.

[1]Google Analytics: http://analytics.google.com

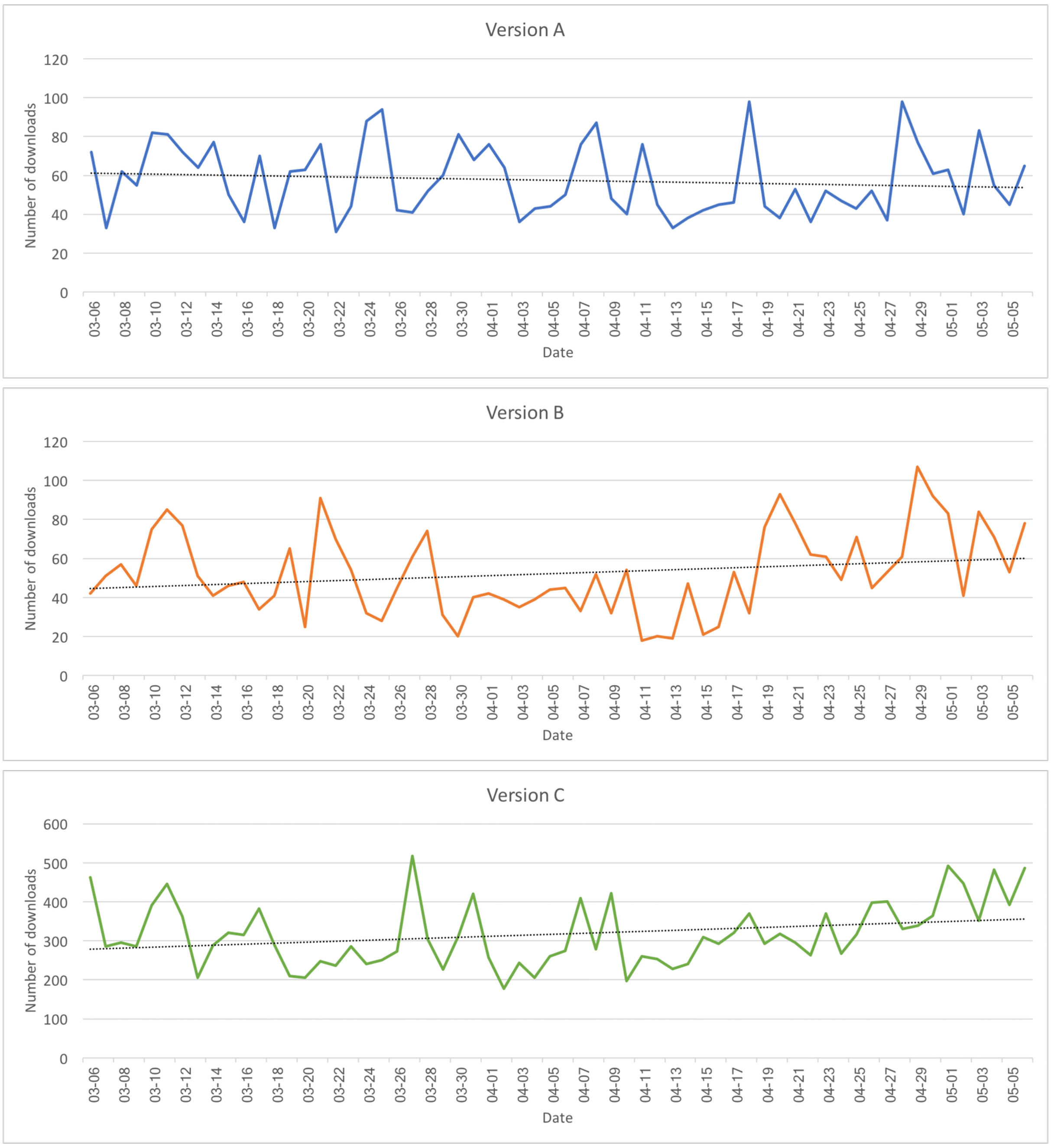


Figure 7.2.: Number of downloads over time for version A, B, and C of test scenario I

Moreover, a comparison of the different conversion rates of each version over the complete test period is given in Table 7.2. Here, the conversion rate is defined as the number of conversions (i.e., downloads) divided by the number of visits. It is important to note that this definition can be made since nearly every user has to visit the home page first before she or he can find and download a Catrobat program. Visiting the home page first is the only possible way for a user to download the Catrobat program, especially when downloading a program directly within the Pocket Code app. Since each version has a different number of visits, the number of conversions can not be directly compared. Instead the conversion rate can be used to compare versions against one another in order to find the best solution.

Measure	**Version A**	**Version B**	**Version C**
Number of visits	9454	8213	32732
Number of conversions	3565	3238	19672
Conversion rate (in %)	37.71	39.43	60.10

Table 7.2.: User activity of each version in test scenario I

Figure 7.3 illustrates the comparison of the conversion rates listed in Table 7.2. The conversion rates achieved by version B and C are higher than the conversion rate of version A. Furthermore, there is a remarkable difference between the conversion rates of version C and version B.

In order to ensure that the differing conversion rates of the evaluated A/B test are representative and can be used for making reliable conclusions, the statistical significance of the A/B test has to be calculated as well. This was done by calculating the confidence level of a chi-squared test, which was based on the number of conversions. The obtained results are presented in Table 7.3. For reasons of clarity, the versions are viewed as three separate version-pairs, or to phrase it differently, the pairs are viewed as three individual A/B tests consisting of only a control version and a variation version. The table shows a positive uplift of the conversion rate as well as high confidence for version B and C when compared to version A (also see Figure 7.3). This also holds for version C when version B is used as control version.

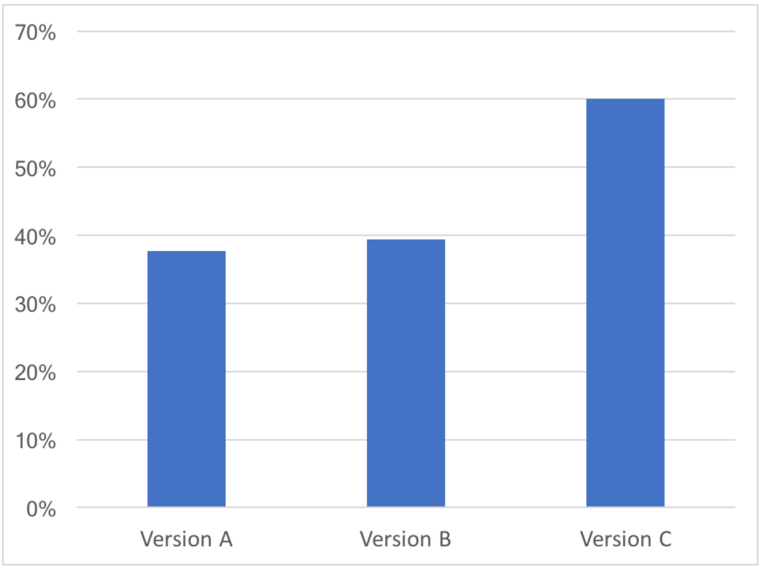


Figure 7.3.: Comparison of conversion rates of version A, B, and C (test scenario I)

Control version	Variation version	Uplift (in %)	Confidence (in %)
Version A	Version B	4.55	98.06
Version A	Version C	59.37	> 99.99
Version B	Version C	52.44	> 99.99

Table 7.3.: Comparison of uplift and confidence level between each version of test scenario I

7.2.2. Results of Test Scenario II

This section shows the results of the second test scenario. It measures the click-through rate and the conversion rate of all three versions. As already explained in Section 6.4.2, the test compares the two implemented recommender approaches (i.e., version B and C are same as in Test scenario I), but introduces a random recommender approach which defines a baseline and is represented by version A. At the end of the test period, the following measures were obtained:

- Number of visits of the community website's home page
- Number of clicks on recommended programs on the home page
- Number of conversions

In this test scenario, the number of conversions was defined as the number of downloads of recommended programs. This number includes only downloads of programs which have been recommended on the home page. This requires two steps from the users; first they have to click on the recommended program on the home page and then they must click on the download button of this recommended program. Otherwise the download will not be counted as a "recommended download".

Each of the aforementioned measures was separated by the language of the user's browser corresponding to the respective version (A, B, or C) of the test and stored for each version. The number of visits was automatically tracked via Google Analytics[2]. The number of clicks was collected via JavaScript and sent to the server via AJAX requests. In order to keep track of the connection between a click on a recommended program and the download of this recommended program, the user context had to be tracked. This was attained by appending a parameter to the URL such that the server could easily figure out whether the user had discovered the program via a recommendation or not.

Table 7.4 summarizes the retrieved information of each version. The click-trough rate and conversion rate of each version were calculated and added to the table as well. They are also visualized in Figure 7.4.

Measure	**Version A**	**Version B**	**Version C**
Number of visits	952	8213	32732
Number of clicks on recommended programs	86	1447	5106
Number of conversions	37	579	2101
Click-through rate (in %)	9.03	17.62	15.60
Conversion rate (in %)	3.89	7.05	6.42

Table 7.4.: User activity of each version in test scenario II

[2]Google Analytics: `http://analytics.google.com`

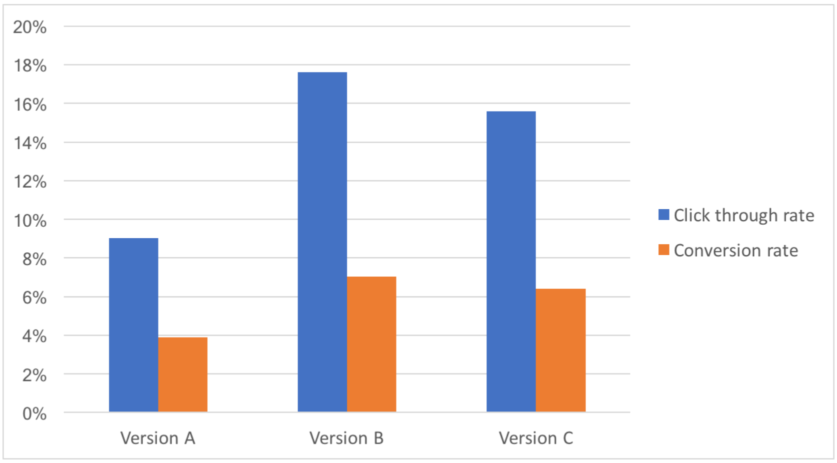


Figure 7.4.: Comparison of the click-through rate and conversion rate of version A, B, and C (test scenario II)

Since the conversion rate and click-through rate would not provide a reliable basis for making meaningful conclusions, the statistical significance of the A/B test had to be proved as well. Thereby, the statistical confidence levels were computed by two two-sided chi-squared test which were (a) based on the number of clicks and (b) based on the number of conversions. For the sake of clarity and simplicity, the versions are viewed as three separate version-pairs, or to phrase it differently, each of these pairs is viewed as an individual A/B test consisting of only a control version and a variation version. Table 7.5 lists the result of the uplift and the confidence level between the two versions of each version-pair for the case (a). These results are based on the number of clicks.

Control version	**Variation version**	**Uplift (in %)**	**Confidence (in %)**
Version A	Version B	95.03	> 99.99
Version A	Version C	72.68	99.99
Version C	Version B	12.94	99.99

Table 7.5.: Comparison of uplift and confidence level based on the number of clicks for case (a) between each version of test scenario II

In addition, the results of case (b) were determined and are shown in Table 7.6. Again, the uplift and confidence level, based on the number of conversions between the two versions of each version-pair, are given. Version B and C achieve a positive uplift with a high confidence when version A is used as control version. Moreover, the same holds true for version C when compared to version B.

Control version	**Variation version**	**Uplift (in %)**	**Confidence (in %)**
Version A	Version B	81.38	99.97
Version A	Version C	65.15	99.84
Version C	Version B	9.83	96.12

Table 7.6.: Comparison of uplift and confidence level based on the number of conversions for case (b) between each version of test scenario II

7.2.3. Results of Test Scenario III

The A/B test of the third test scenario used two versions, i.e. versions A and B (see Section 6.4.3). Before the remixing activity is compared, it is advisable to have a look at the usage data of the remix graph visualization tool first. The following data has been measured:

- Number of visits of remix graphs
- Number of clicks on recommended programs
- Number of conversions

Since the “Show remix graph” button was hidden in version A, this data is only available for version B. The number of visits of remix graphs refers to the number of clicks on the “Show Remix Graph” button (see Section 6.4.3), which is shown on the detail page of every Catrobat program in version B. This number has been tracked via JavaScript in the same way as in test scenario II. The measured number of clicks is exact since a click on the aforementioned button is the only possible way for a user to get to see a program’s remix graph. Furthermore, the number of clicks on recommended programs refers to the number of clicks on the open button of the context-menu of any remixed program in a remix graph (see Figure 6.7 in Section 6.1.3). All remixed programs are interpreted as recommended programs due to their content-similarity to the original program (see Section 6.4.3). Finally, in this test scenario, the number of conversions is defined as the number of downloads of such recommended programs found in the remix graph of another program. Again, all clicks were captured and the downloads of recommended programs were tracked via JavaScript in the exact same manner as the number of visits.

Table 7.7 shows the results retrieved from the MySQL database. In addition, the click-through rate and the conversion rate was calculated. Even though the click-through rate is quite high, the number of conversions is remarkably low.

Measure	**Version B**
Number of visits	1751
Number of clicks on recommended programs	354
Number of conversions	25
Click-through rate (in %)	20.22
Conversion rate (in %)	1.43

Table 7.7.: User activity of version B in test scenario III

However, for a detailed investigation, the overall impact on the remixing activity of this new visualization feature should be measured. Since a positive increase in the remixing activity always implies a growth in the number of uploads, a first approach would be to take a look at the temporal development of the number of uploads. Thus, the number of uploads was examined over time for both versions and the temporal movement of this data is shown in Figure 7.5 for the complete test period. Each diagram presents the number of uploads of the corresponding version on the y-axis and the date on the x-axis. The blue line shows the number of uploads of version A and the orange line represents version B. A trend-line was computed for both diagrams by using linear regression and is visualized as a linear dotted line in the diagrams. The upper diagram of version A shows a slightly decreasing trend-line throughout the test period, whereas the trend-line of version B in the lower diagram indicates a noticeable increase of the number of uploads throughout the test period.

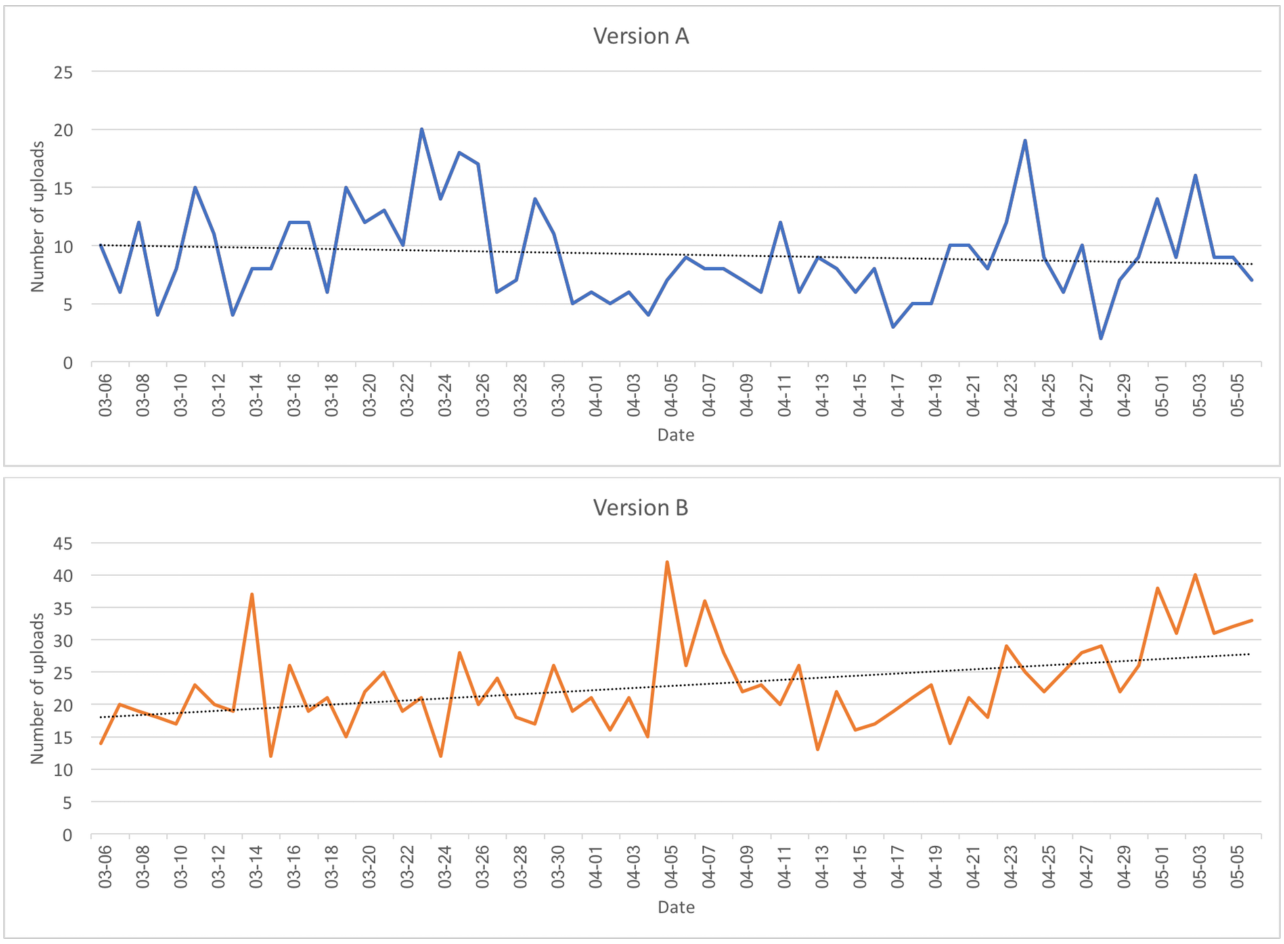


Figure 7.5.: Number of uploads over time for version A and B of test scenario III

For a more detailed and reliable analysis of the change in remixing activity, the number of remixed uploads has to be taken into account; the statistical significance must be proven as well. Since the "Show remix graph" button was hidden in version A and hence the visualization was not available to users of version A, comparable measures had to be defined, which were:

- Number of uploaded Catrobat programs
- Number of remixes of these uploaded programs

The census of these two numbers did not require any additional implementation since both can be easily inferred from the data stored in the MySQL database. Table 7.8 presents the results for the duration of the full test period. As the number of uploaded programs differs between both versions, the remix rate was used for comparison purposes. The remix rate is the quotient between the number of uploaded remixes and the number of uploaded programs. As a result, version B achieved a higher remix rate than version A.

Measure	Version A	Version B
Number of uploaded programs	572	1422
Number of uploaded remixes	124	411
Remix rate (in %)	21.68%	28.86%

Table 7.8.: Uploaded programs and remix-programs of each version in test scenario III

However, the difference in both remix rates could have been the result of a fluke in the experiment; therefore, the statistical significance of this A/B test needed to be proven. Hence, the statistical confidence levels were computed by a two two-sided chi-squared test, which was based on the number of remixes. Table 7.9 shows the result of the uplift and the confidence level between the variation version B and control version A. Version B achieved a noticeable uplift of 33.32% with a confidence level of 99.90%.

Control version	Variation version	Uplift (in %)	Confidence (in %)
Version A	Version B	33.32	99.90

Table 7.9.: Uplift and confidence level based on the number of remixes between version B and control version A of test scenario III

7.3. Discussion of Results

The three A/B tests were conducted over a two month long test period. Before the tests were evaluated, key characteristics of Catrobat's online community were examined in more detail at the end of the test period. According to the data shown in Table 7.1 and the diagram depicted in Figure 7.1, remixing appears to be a popular feature in Catrobat. At the time of writing this theses, about one out of four uploaded Catrobat programs represented a remix of another Catrobat program and about 11.9% of all users have remixed at least one Catrobat program. Although these numbers are quite high, it is important to take into account that this feature is already in action for many years. However, the number of converted Scratch programs is quite low and only contributes 0.2% to the total number of uploaded programs. This is mainly due to the reason that Catrobat's Scratch converter subproject (see Section 5.5) is still in the beta phase and hence has not been promoted by the Catrobat organization. In contrast to remixing, the like rating system (see Section 6.2) was implemented as part of the practical part of this thesis. It was released a few weeks before the start of the test period. Nevertheless, it achieved a remarkably high user adoption rate (compared to remixing) and broke the barrier of 1,000 different users, who have given a like on at least one Catrobat program, within that short period of time (see Table 7.1). Thus, the like feature appears to be very appealing to Catrobat's user community.

7.3.1. Discussion of Test Scenario I

The aim of the first test scenario was to prove the hypothesis which was defined at the beginning of this thesis:

> *"Do recommendations of Catrobat programs have a positive impact on the overall download activity?"*

The trend-lines in Figure 7.2 of Section 7.2.1 show a positive trend for user-based recommendations based on like ratings (version B) and user-based recommendations based on remixing (version C). In contrast, the slope of the trend-line of the unmodified home page (control version A) which does not show any recommendations at all, even indicates a negative trend. Since a viewable increase of the download activity only occurred in both versions which recommended programs, the recommendations might be the underlying reason for this increase. However, this assumption is not sufficient to answer the first research question. Hence, the number of visits has to be taken into account as well as tests for statistical significance are required. Due to the nature of this A/B test, the number of clicks on recommended programs as well as the number of downloads of recommended programs can not be used to compare all versions. This is mainly because, version A does not show any recommendations, meaning both values would always be zero and hence no reasonable comparison can be made. Instead, all downloads of a version were included in the calculations for the respective version (i.e., *overall* download activity of the respective version). The results of the conversion rates which are listed in Table 6.1 and visualized Figure 7.3, can expressly underline the previous findings and also provide a more detailed look at the performance of all three versions. Table 7.3 presents the uplift and confidence level of each version pair. Again, the obtained results confirm the previous observations. It is remarkable, that all confidence levels of all version pairs are higher than 95%. This consequently means that the uplifts are statistically significant.

Based on these confidence levels and the uplift results, the following conclusions can be made:

- The home page of version C which included user-based recommendations based on remixes was able to *significantly* outperform both competitor versions, i.e. the original home page (version A) as well as the home page of version B which generated user-based recommendations based on like preferences (very high significance: both confidence levels are $> 99.99\%$).

- Likewise, version B still performed *significantly* better than version A (significant result with 98.06% confidence).

As a consequence, the hypothesis is most probably true for both implemented recommendation approaches. In addition, the investigations of this test scenario also show that the recommendations produced by the first recommendation approach in version C have a higher impact on the *overall* download activity than the recommendations generated by the second recommendation approach (see Section 6.3.3). However, this does not say anything about the performance of version B and C. Consequently, a different A/B testing approach is needed to examine the quality of the two recommendation approaches. Actually, test scenario II represents such a testing approach.

7.3.2. Discussion of Test Scenario II

The second test scenario tried to tackle the second research question of this thesis which was defined as follows:

> *"Can a user-based recommendation approach based on collaborative filtering produce recommendations of higher quality compared to those generated by naive recommendation approaches? Which of the two implemented user-based collaborative filtering approaches performs best: (a) the first approach, which is based on remixing data, or (b) the second approach, which is based on like ratings?"*

In contrast to the first test scenario, all three versions produced recommendations in the second test scenario. Therefore, the number of clicks on recommended programs and the number of downloads of recommended programs can be measured for all versions (A, B, and C). Moreover, the click-through rate based on the number of those clicks and the conversion rate based on the number of such downloads can be computed and directly compared among each other. This allows more accurate conclusions about the quality of the implemented recommendation approaches, since the click-through rate and conversion rate only take the user activities in terms of recommended programs into consideration.

Given the comparison in Figure 7.4 and the results in Table 7.5 and 7.6, the click-through rate and the conversion rate of both implemented recommendation approaches (i.e. version B and C) performs *significantly* better than the naive recommendation approach which was based on random recommendations. The uplift of the click-through rate and conversion rate between version B and A as well as between version C and A are quite high and also have a very high confidence level. In addition, there is also an uplift of the click-through rate and conversion rate between version C and B. However, this uplift is small but is still significant.

Based on these results, one can conclude that:

- The quality of the recommendations produced by both user-based collaborative filtering approaches outweighs the baseline of the naive recommendation approach (high statistical significance of uplift based on "recommended" clicks as well as "recommended" downloads/conversions; all over 99%).
- The second recommendation approach which is based on like ratings generates and suggests more valuable recommendations to the user than the first recommendation approach which is based on remixing data.

The bottom line is that a user-based recommendation approach based on collaborative filtering can produce recommendations of higher quality compared to those random suggestions generated by a naive recommendation approach. Furthermore, the second recommendation approach performs significantly better in terms of "recommended" clicks and "recommended" downloads than the first recommendation approach. This stands in contrast to the research finding of the first test scenario, where the first approach reached better results in terms of *overall* download activity than the second approach. Considering all theses findings, it would be particularly advisable for Catrobat to migrate the existing recommender system to a hybrid solution, which suggests a combination of the Catrobat programs recommended by both approaches. Such a hybrid system would be able to combine the strengths and minimize the weaknesses of both approaches (see Section 4.4.5). Besides that, with an increasing amount of collaborative remix and like data and a growing user community, the quality and performance of both recommendation approaches are expected to automatically improve as time goes by.

7.3.3. Discussion of Test Scenario III

The last test scenario deals with the third research question of this thesis:

> *"Given the set of Catrobat programs remixed by all users, can these remixed programs be used to generate high quality content-based recommendations for other users in order to increase the overall remixing activity?"*

Unlike the previous two test scenarios, the third test scenario used an A/B test which consisted of only two versions (version A and B) and was active on the detail page of every Catrobat program. It is based on the A/B test of the first test scenario and uses the same version A and its version C as version B. The results of version B in Table 7.7 show an acceptable click-through rate of 20.22%, meaning that the remixed programs were clicked quite often. However, version B could only achieve a very low conversion rate of 1.43%. This means that the remixed programs presented by the visualization tool might not represent an appropriate approach for content-based recommendations. Hence, the third research question has to be rejected/negated for the content-based approach on the detail page.

Since all tests were ran at the same time, this test is also deeply related to the impact of the first test which generated user-based recommendations on the home page. Therefore, the activity of the uploads was studied in both versions. By looking at the trend-lines in Figure 7.5, one can spot a latent trend of the upload activity throughout the test period. The trend-line in version A indicates a negative trend whereas the trend-line of version B shows a positive trend. In addition, version B achieved a higher remix rate and a decent uplift of 33.32% (see Table 7.8 and Table 7.9) which are very significant (confidence level: 99.9%).

Consequently, one can conclude that remixed programs can be used as recommendations in order to increase the overall remixing activity. However, the content-based recommendations generated by the visualization tool are not a accurate enough to achieve an acceptable number of downloads/conversions. Instead, the increase of the remixing activity in version B most probably happened because of the recommendations on the home page. Nevertheless, the quite high click-through rate of the content-based recommendations indicate a certain user demand for this feature. Therefore, it can still be suggested to Catrobat to continue providing this feature to the users but to not view it as a standalone recommender system.

In conclusion, this chapter presented general statistics about Catrobat's community website and the results of the evaluated A/B tests. Furthermore, answers were given to all research questions. The investigations were explained in more detail and suggestions based on the scientific outcome were made to Catrobat.

8. Conclusion

This thesis concludes with a recapitulation of the most essential findings and provides suggestions for improvement which are given to Catrobat. The thesis was structured into a theoretical part and a practical part which was built upon the theoretical part. The chapters of the theoretical part conveyed important background knowledge about collaborative software and recommender systems. In the practical part of this thesis, the integration of a basic recommender system into Catrobat's collaborative e-learning system, the community website (see Section 5.4), was discussed. Two user-based collaborative filtering approaches, which have been implemented as part of the practical part of this thesis, were studied. In addition, a content-based recommender approach, which had been designed to visualize the remix graph of a Catrobat program, was investigated as well.

8.1. Outcome

All obtained results, which were presented in Chapter 7, show that both user-based collaborative filtering approaches represent a great benefit to the users of the community website. Both of these user-based approaches are easy to understand and readily comprehensible since they are built upon simple mathematical foundations. It is important to mention that the remixing feature was available for a considerably longer period of time when compared to the like rating system. This is because the like rating system has been released only a few weeks before the start of the test period. Consequently, more remixing data was available than like rating data. Therefore, the first approach which was based on remix ratings, could use more data in order to generate predictions when compared to the second approach which was based on like rating data. Nevertheless, the second approach performed significantly better in terms of number of clicks and number of downloads. Although the like rating system was released only a few weeks before the test period, the new like rating feature showed an immense user adoption rate. Since that time, more than one thousand users have already liked at least one Catrobat program. In addition, the collaborative approach, which was based on like ratings, also performed significantly better in terms of click-through rate and conversion rate than the other approach which was based on remixing data.

According to the results in Chapter 7, the content-based recommender of the remix graph achieved a satisfactory click-through rate but a very low conversion rate in terms of downloads. The click-through rate and the high number of visits shows that the visualization tool of the remix graph is primarily used to find and navigate through remix-related Catrobat programs. The combination of a high click-through rate and a low conversion rate means that a large number of users appears to be interested in discovering remix-related programs but many of these users could probably not find what they were actually looking for and hence did not download the program after seeing the detail page of the clicked program. Nevertheless, in retrospect, the visualization tool had a significant positive impact on the overall remixing activity.

At the time of writing this thesis, about one of four uploaded programs are remixes which is a quite acceptable remix rate. Moreover, less than one percent of all uploaded programs on the community website have been Scratch remixes. This is largely due to the status of the Scratch converter, which is still in a beta phase and has not yet been actively promoted.

8.2. Future Work

Based on the outcome of the evaluation results, it is suggested that Catrobat adopts both implemented recommendation approaches for ongoing use. Although the approaches are already performing quite satisfactorily, further development of the recommender system and the adaption to more advanced recommendation algorithms is advisable. Moreover, further statistical investigations would be advantageous for Catrobat in order to maintain a well-developed, progressive application. In particular, research in the area of recommender systems shows that hybrid recommender solutions have proven to perform very well in practice. As a result, it is proposed that Catrobat proceeds with the migration to a hybrid recommender system which combines the suggestions of both implemented collaborative filtering approaches. It must be stressed, that the quality of a collaborative filtering approach depends heavily on the amount and quality of collaborative data. Even though Catrobat already has a very active community with many users, the performance of both implemented approaches will greatly improve with an increasing amount of collaborative data and a growing number of users. In particular, the high user adoption rate of the new like feature reveals great potential of the collaborative like rating data to act as a more appropriate input source for future collaborative filtering approaches. However, as the number of Catrobat programs and the number of users grows, so does the factor of sparsity of a rating matrix. Therefore, a model-based collaborative filtering algorithm would be a more appropriate process in terms of performing dimensionality reduction and discovering latent factors within the data (see Model-based Collaborative Filtering in Section 4.4.2). Although research about model-based collaborative filtering systems primarily focuses on non-binary data, there still exist some approaches which can be applied on binary data. For example, Koren, Bell, and Volinsky, 2009 propose a collaborative filtering algorithm which is based on a matrix factorization technique and tailored for implicit feedback. The approach is also optimized to be efficiently used in parallel computing environments. An implementation of this parallel algorithm can be found, for example, in Apache Spark's Machine Learning Library[1]. In general, matrix factorization approaches achieved great success during the Netflix Prize competition[2]. Thus, moving from the existing memory-based technique to such a matrix factorization approach in order to improve and optimize the performance and prediction accuracy of the community website's recommender system would be another possible option for Catrobat.

Concerning the results of the content-based recommender system, there is also plenty of room left for improvements. An alternative approach, especially for very large remix graphs (with hundreds of programs), would be to view each remix graph as a web graph and to apply, for example, Google's Page Rank algorithm on the graph. The Page Rank algorithm and web graph have been

[1]Apache Spark MLlib: `http://spark.apache.org/mllib`
[2]Netflix Prize: `http://www.netflixprize.com`

briefly described in Section 3.1.2 (also see Figure 3.3). As with the original web graph, the remix graph also consists of nodes which refer to pages (i.e., the detail page of the respective Catrobat program) and of edges which describe the connection between two program detail pages. Those Catrobat programs (i.e., nodes) that received the highest weights by the algorithm can then be easily recommended to the user. However, in order to encourage more users to take advantage of the remixing feature and to raise the conversion rate, a higher remix rate is indispensable. Monroy-Hernández, 2012 showed that the engagement of users in remixing particularly depends on the presence or absence of self-written acknowledgement notes or *manual* attributions given to the user who created the source program which has been remixed. In the context of Catrobat, this would consequently mean that an increase in the engagement in remixing and hence a higher remix rate could probably be achieved by prompting users to enter a short acknowledgement text within the Pocket Code app whenever they attempt to upload a remixed Catrobat program. It is important to note that this makes sense only for those remixed programs that originated from other users. In conclusion, it could be suggested to highlight this text on the community website and to automatically promote these programs. Furthermore, another possible option for Catrobat could be to adapt the recommender system in such a way that more preference is automatically given to such programs.

Concerning the number of converted Scratch programs, this number reveals a great need for improvement within the system. Since the Scratch platform hosts millions of Scratch programs, it offers huge potential for Catrobat too. Therefore, active promotion of the Scratch converter is recommended to Catrobat as soon as the Scratch converter leaves its beta state.

Finally, it should be acknowledged that there are numerous ways to improve the quality of the recommendations and the data collection process. Furthermore, enhancements towards dealing with problematic situations, such as the cold start problem, could also positively affect the overall performance of a recommender system. However, considering and discussing all of these cases would have been beyond the scope of this thesis and hence they were not covered in this thesis.

Appendix

Appendix A.

List of Abbreviations

Abbreviation	**Explanation**
ACM	Association for Computing Machinery
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
App	Application
BSD	Berkeley Software Distribution
CTR	Click-Through Rate
GNU	Name of a collection of software and the GNU Operating System
FOSS	Free and Open Source Software
MIT	Massachusetts Institute of Technology
CSS	Cascading Style Sheets
CSCW	Computer-supported Cooperative Work
E-Learning	Electronic Learning
E-Commerce	Electronic Commerce
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
ID	Identifying Number
IDE	Integrated Development Environment
ORM	Object-relational Mapping
REST	Representational State Transfer
SQL	Structured Query Language
PHP	Hypertext Preprocessor
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformation
URL	Uniform Resource Locator
ZIP archive	Compressed file archive

Appendix B.

Web API Reference

Finally, a brief overview over the two implemented server-side REST API methods is given below. The overview includes the HTTP method, the address URL, a short description, the corresponding PHP Class, method name, method parameters and a code snippet of the implemented API method.

API of Recommender System

HTTP Request:
`GET /api/projects/recsys_general_programs.json`

PHP Class, Method and Parameters:
`ProgramController::listRecsysGeneralProgramsAction()`

Description:
Shows user-based recommendations generated by the recommender system. The recommendations are either based on the remixes of the active user, based on the like ratings of the active user or random suggestions. The decision which these three recommendation approaches is chosen depends on the browser language settings of the active user (see A/B tests in Section 6.4.1, Section 6.4.2, and Section 6.4.3).

Code:

```php
<?php
// [...]
public function listRecsysGeneralProgramsAction(Request $request)
{
    $limit = intval($request->query->get('limit', 20));
    $offset = intval($request->query->get('offset', 0));

    $program_manager = $this->get('programmanager');
    $flavor = $request->getSession()->get('flavor');

    $locale = strtolower($request->getLocale());
    $programs_count = 0;
    $programs = [];
    $is_user_specific_recommendation = false;

    if (substr($locale, 0, 2) == 'de') {
        $user = $this->getUser();
        if ($user != null) {
            $recommender_manager = $this->get('recommendermanager');
            $all_programs = $recommender_manager
                ->recommendProgramsOfLikeSimilarUsers($user,
                    $flavor);
            $programs_count = count($all_programs);
            $programs = array_slice($all_programs, $offset, $limit);
        }

        if (($user == null) || ($programs_count == 0)) {
            $programs_count = $program_manager
                ->getTotalLikedProgramsCount($flavor);

            $programs = $program_manager
                ->getMostLikedPrograms($flavor, $limit, $offset);
        } else {
            $is_user_specific_recommendation = true;
        }
    } else if (substr($locale, 0, 2) == 'fr') {
        $programs_count = count($program_manager
            ->getTotalPrograms($flavor));

        $programs = $program_manager
            ->getRandomPrograms($flavor, $limit, $offset);

    } else {
        $user = $this->getUser();
        if ($user != null) {
            $recommender_manager = $this->get('recommendermanager');
            $all_programs = $recommender_manager
                ->recommendProgramsOfRemixSimilarUsers($user,
                    $flavor);
            $programs_count = count($all_programs);
            $programs = array_slice($all_programs, $offset, $limit);
        }

        if (($user == null) || ($programs_count == 0)) {
            $programs_count = $program_manager
                ->getTotalRemixedProgramsCount($flavor);

            $programs = $program_manager
                ->getMostRemixedPrograms($flavor, $limit, $offset);

        } else {
            $is_user_specific_recommendation = true;
        }
    }
    return new ProgramListResponse($programs, $programs_count, true,
        $is_user_specific_recommendation);
}
```

Listing B.1: PHP Code of the Recommender System's API method

Data Retrieval API of Remix Graph

HTTP Request:
`GET /api/program/remixgraph/{id}`

PHP Class, Method and Parameters:
`RecommenderController::programRemixGraphAction($id)`

Description:
Returns the list that includes all nodes and edges of the complete remix graph of a program. The `{id}` *parameter specifies the ID of the desired program and is automatically passed as* `$id` *parameter to the* `programRemixGraphAction` *method by the Symfony Framework.*

Code:

```php
<?php
// [...]
public function programRemixGraphAction(Request $request, $id)
{
    $remix_graph_data = $this->get('remixmanager')
                             ->getFullRemixGraph($id);

    $screenshot_repository = $this->get('screenshotrepository');
    $catrobat_program_thumbnails = [];
    foreach ($remix_graph_data['catrobatNodes'] as $node_id) {
        if (!array_key_exists($node_id,
            $remix_graph_data['catrobatNodesData'])) {
            $catrobat_program_thumbnails[$node_id] =
                '/images/default/not_available.png';
            continue;
        }
        $catrobat_program_thumbnails[$node_id] = '/' .
            $screenshot_repository->getThumbnailWebPath($node_id);
    }
    // [...]
    return new JsonResponse([
        'id' => $id,
        'remixGraph' => $remix_graph_data,
        'catrobatProgramThumbnails' => $catrobat_program_thumbnails,
    ]);
}
```

Listing B.2: PHP Code of the Data Retrieval API method

Appendix C.

Code Snippet of User Similarity Computation

The code snipped in Listing C.1 describes the computation of user similarities based on either likes or remixes. Although some variables in the code contain the word "like", the same code snippet can be used without any modifications in order to calculate the similarities based on remixes.

The shown code demonstrates the efficient implementation by using appropriate data structures from Java's class library such as HashSets and HashMaps. These data structures are used for determining the intersection (see variable `numOfSameProgramsLikedByBoth`) and union (see variable `numOfAllProgramsLikedByAnyOfBoth`) from both user sets. At the end of each loop iteration, the distance result of the respective user pair is packed into a MySQL-specific SQL statement and stored into a SQL file. It is important to mention, since Doctrine can not be used by Java, this small part of the implementation is not platform-independent.

```
[...]

public static void computeSimilarities(...) {
  int userCounter = 0;
  int totalNumOfRemixedUsers = userLikeRelations.size();
  Set<String> alreadyAddedRelations = new HashSet<>();
  writer.write("use " + sqlDbName + ";\n");
  Map.Entry<Integer, Set<Integer>> entry;

  for (entry : userLikeRelations.entrySet()) {
    Integer firstUserId = entry.getKey();
    Set<Integer> idsOfProgramsLikedByFirstUser = entry.getValue();
    ++userCounter;
    Map.Entry<Integer, Set<Integer>> secondEntry;

    for (secondEntry : userLikeRelations.entrySet()) {
      Integer secondUserId = secondEntry.getKey();
      Set<Integer> idsOfProgramsLikedBySecondUser =
          secondEntry.getValue();

      String key = firstUserId + "_" + secondUserId;
      String reverseKey = secondUserId + "_" + firstUserId;

      if ((firstUserId.intValue() == secondUserId.intValue()) ||
          alreadyAddedRelations.contains(key)
          || alreadyAddedRelations.contains(reverseKey)
      ) {
        continue;
      }

      alreadyAddedRelations.add(key);

      Set<Integer> idsOfSameProgramsLikedByBoth = new
          HashSet<Integer>(idsOfProgramsLikedByFirstUser);
      idsOfSameProgramsLikedByBoth.retainAll(
          idsOfProgramsLikedBySecondUser);

      Set<Integer> idsOfAllProgramsLikedByAnyOfBoth = new
          HashSet<Integer>(idsOfProgramsLikedByFirstUser);
      idsOfAllProgramsLikedByAnyOfBoth.addAll(
          idsOfProgramsLikedBySecondUser);

      int numOfSameProgramsLikedByBoth =
          idsOfSameProgramsLikedByBoth.size();
      int numOfAllProgramsLikedByAnyOfBoth =
          idsOfAllProgramsLikedByAnyOfBoth.size();

      if (numOfSameProgramsLikedByBoth == 0) {
        continue;
      }

      float jaccardSim = ((float) numOfSameProgramsLikedByBoth)
                       / ((float) numOfAllProgramsLikedByAnyOfBoth);

      if (jaccardSim < 0.01) {
        continue;
      }

      writer.write("REPLACE INTO " + sqlTableName
          + " (first_user_id, second_user_id, similarity,"
          + " created_at) VALUES (" + firstUserId + ", "
          + secondUserId + ", " + jaccardSim + ", NOW());\n");
    }
  }
  writer.write("\n");
}
```

Listing C.1: Java implementation of user similarity computation

Bibliography

Adomavicius, Gediminas and Alexander Tuzhilin (2005). “Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions.” In: *IEEE Trans. on Knowl. and Data Eng.* 17.6, pp. 734–749. ISSN: 1041-4347. DOI: 10.1109/TKDE.2005.99. URL: https://doi.org/10.1109/TKDE.2005.99 (cit. on p. 49).

Adomavicius, Gediminas and Alexander Tuzhilin (2008). “Context-aware Recommender Systems.” In: *Proceedings of the 2008 ACM Conference on Recommender Systems*. RecSys ’08. Lausanne, Switzerland: ACM, pp. 335–336. ISBN: 978-1-60558-093-7. DOI: 10.1145/1454008.1454068. URL: http://doi.acm.org/10.1145/1454008.1454068 (cit. on p. 34).

Aggarwal, Charu C. (2016). *Recommender Systems: The Textbook*. 1st. Springer Publishing Company, Incorporated. ISBN: 3319296574, 9783319296579 (cit. on pp. 50, 51).

Anderson, Chris (2006). *The Long Tail: Why the Future of Business Is Selling Less of More*. 1st ed. New York: Hyperion, p. 238 (cit. on p. 28).

Army ants' "living" bridges span collective intelligence, "swarm" robotics (2015). URL: https://blogs.princeton.edu/research/2015/11/24/army-ants-living-bridges-span-collective-intelligence-swarm-robotics-pnas/ (visited on 02/22/2017) (cit. on p. 8).

Article about "PageRank Algorithm" (2017). URL: https://en.wikipedia.org/w/index.php?title=PageRank&oldid=774470009 (visited on 04/10/2017) (cit. on p. 13).

Article of the Boeing 787 Dreamliner (2017). URL: https://en.wikipedia.org/wiki/Boeing_787_Dreamliner (visited on 04/08/2017) (cit. on p. 11).

Baecker, R.M. (1995). *Readings in Human-computer Interaction: Toward the Year 2000*. Interactive Technologies Series. Morgan Kaufmann Publishers. ISBN: 9781558602465. URL: https://books.google.at/books?id=gjm6FpMUTXgC (cit. on p. 15).

Beel, Joeran and Stefan Langer (2015). "A Comparison of Offline Evaluations, Online Evaluations, and User Studies in the Context of Research-Paper Recommender Systems." In: *Research and Advanced Technology for Digital Libraries: 19th International Conference on Theory and Practice of Digital Libraries, TPDL 2015, Poznań, Poland, September 14-18, 2015, Proceedings*. Ed. by Sarantos Kapidakis, Cezary Mazurek, and Marcin Werla. Cham: Springer International Publishing, pp. 153–168. ISBN: 978-3-319-24592-8. DOI: 10.1007/978-3-319-24592-8_12. URL: http://dx.doi.org/10.1007/978-3-319-24592-8_12 (cit. on p. 56).

Borghoff, Uwe M. and J. H. Schlichter (2000). *Computer-Supported Cooperative Work: Introduction to Distributed Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 3540669841 (cit. on p. 17).

Boyd, Danah M. and Nicole B. Ellison (2007). "Social Network Sites: Definition, History, and Scholarship." In: *Journal of Computer-Mediated Communication* 13.1, pp. 210–230. ISSN: 1083-6101. DOI: 10.1111/j.1083-6101.2007.00393.x. URL: http://dx.doi.org/10.1111/j.1083-6101.2007.00393.x (cit. on p. 22).

Brand, Matthew (2002). "Incremental Singular Value Decomposition of Uncertain Data with Missing Values." In: *Proceedings of the 7th European Conference on Computer Vision-Part I*. ECCV '02. London, UK, UK: Springer-Verlag, pp. 707–720. ISBN: 3-540-43745-2. URL: http://dl.acm.org/citation.cfm?id=645315.649157 (cit. on p. 47).

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 pp. 46, 47).

Brynjolfsson, Erik, Yu (Jeffrey) Hu, and Duncan Simester (2011). "Goodbye Pareto Principle, Hello Long Tail: The Effect of Search Costs on the Concentration of Product Sales." In: *Manage. Sci.* 57.8, pp. 1373–1386. ISSN: 0025-1909. DOI: 10.1287/mnsc.1110.1371. URL: http://dx.doi.org/10.1287/mnsc.1110.1371 (cit. on p. 28).

Buder, Jürgen and Christina Schwind (2012). "Learning with Personalized Recommender Systems: A Psychological View." In: *Comput. Hum. Behav.* 28.1, pp. 207–216. ISSN: 0747-5632. DOI: 10.1016/j.chb.2011.09.002. URL: http://dx.doi.org/10.1016/j.chb.2011.09.002 (cit. on p. 32).

Burke, Robin (2000). “Knowledge-Based Recommender Systems.” In: *ENCYCLOPEDIA OF LIBRARY AND INFORMATION SYSTEMS*. Marcel Dekker, p. 2000 (cit. on p. 34).

Burke, Robin (2002). “Hybrid Recommender Systems: Survey and Experiments.” In: *User Modeling and User-Adapted Interaction* 12.4, pp. 331–370. ISSN: 0924-1868. DOI: 10.1023/A:1021240730564. URL: http://dx.doi.org/10.1023/A:1021240730564 (cit. on pp. 51, 52).

Burke, Robin (2007). “The Adaptive Web.” In: ed. by Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. Berlin, Heidelberg: Springer-Verlag. Chap. Hybrid Web Recommender Systems, pp. 377–408. ISBN: 978-3-540-72078-2. URL: http://dl.acm.org/citation.cfm?id=1768197.1768211 (cit. on p. 27).

Campos, Pedro G., Fernando Diez, and Ivan Cantador (2014). “Time-aware Recommender Systems: A Comprehensive Survey and Analysis of Existing Evaluation Protocols.” In: *User Modeling and User-Adapted Interaction* 24.1-2, pp. 67–119. ISSN: 0924-1868. DOI: 10.1007/s11257-012-9136-x. URL: http://dx.doi.org/10.1007/s11257-012-9136-x (cit. on p. 34).

Candillier, Laurent, Frank Meyer, and Françoise Fessant (2008). “Designing Specific Weighted Similarity Measures to Improve Collaborative Filtering Systems.” In: *Proceedings of the 8th Industrial Conference on Advances in Data Mining: Medical Applications, E-Commerce, Marketing, and Theoretical Aspects*. ICDM ’08. Leipzig, Germany: Springer-Verlag, pp. 242–255. ISBN: 978-3-540-70717-2. DOI: 10.1007/978-3-540-70720-2_19. URL: http://dx.doi.org/10.1007/978-3-540-70720-2_19 (cit. on pp. 45, 46).

Carstensen, Peter H. and Kjeld Schmidt (1999). “Computer Supported Cooperative Work: New Challenges to Systems Design.” In: *In K. Itoh (Ed.), Handbook of Human Factors*, pp. 619–636 (cit. on p. 14).

Claypool, Mark et al. (1999). *Combining Content-Based and Collaborative Filters in an Online Newspaper* (cit. on p. 50).

Cunningham, Padraig et al. (2001). “WEBSELL: Intelligent Sales Assistants for the World Wide Web.” In: *KI* 15.1, pp. 28–32 (cit. on p. 52).

Dalsgaard, Christian (2006). “Social software: E-learning beyond learning management systems.” In: *European Journal of Open, Distance and E-Learning (EURODL)*. URL: http://www.eurodl.org/materials/contrib/2006/Christian_Dalsgaard.htm (cit. on pp. 23, 24).

Dasgupta, Sayamindu et al. (2016). “Remixing As a Pathway to Computational Thinking.” In: *Proceedings of the 19th ACM Conference on Computer-Supported*

Cooperative Work & Social Computing. CSCW '16. San Francisco, California, USA: ACM, pp. 1438–1449. ISBN: 978-1-4503-3592-8. DOI: 10.1145/2818048.2819984. URL: http://doi.acm.org/10.1145/2818048.2819984 (cit. on p. 25).

Deerwester, Scott et al. (1990). "Indexing by latent semantic analysis." In: *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE* 41.6, pp. 391–407 (cit. on p. 50).

Du, Zhao et al. (2013). "Interactive and Collaborative E-Learning Platform with Integrated Social Software and Learning Management System." In: *Proceedings of the 2012 International Conference on Information Technology and Software Engineering: Software Engineering & Digital Media Technology*. Ed. by Wei Lu et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 11–18. ISBN: 978-3-642-34531-9. DOI: 10.1007/978-3-642-34531-9_2. URL: http://dx.doi.org/10.1007/978-3-642-34531-9_2 (cit. on p. 24).

Durkheim, EC (1912). *The Elementary Forms of the Religious Life, trans. Karen Fields* (cit. on p. 7).

Ekstrand, Michael D., John T. Riedl, and Joseph A. Konstan (2011). "Collaborative Filtering Recommender Systems." In: *Found. Trends Hum.-Comput. Interact.* 4.2, pp. 81–173. ISSN: 1551-3955. DOI: 10.1561/1100000009. URL: http://dx.doi.org/10.1561/1100000009 (cit. on p. 36).

Ellis, Clarence A., Simon J. Gibbs, and Gail Rein (1991). "Groupware: Some Issues and Experiences." In: *Commun. ACM* 34.1, pp. 39–58. ISSN: 0001-0782. DOI: 10.1145/99977.99987. URL: http://doi.acm.org/10.1145/99977.99987 (cit. on pp. 14, 15, 17).

Facts about Google and Competition (2011). URL: https://web.archive.org/web/20111104131332/https://www.google.com/competition/howgooglesearchworks.html (visited on 11/04/2011) (cit. on p. 12).

Felfernig, Alexander et al. (2014). "Basic Approaches in Recommendation Systems." In: *Recommendation Systems in Software Engineering*. Ed. by Martin P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 15–37. ISBN: 978-3-642-45135-5. DOI: 10.1007/978-3-642-45135-5_2. URL: http://dx.doi.org/10.1007/978-3-642-45135-5_2 (cit. on p. 34).

Fessakis, Georgios and Angelique Dimitracopoulou (2009). "Proposing "Collaborative Filtering" to Foster Collaboration in ScratchR Community." In: *Proceedings of the 9th International Conference on Computer Supported Collaborative Learning - Volume 2*. CSCL'09. Rhodes, Greece: International Society of the Learning Sciences, pp. 168–170. ISBN: 978-1-4092-8598-4.

URL: http://dl.acm.org/citation.cfm?id=1599503.1599560 (cit. on p. 98).

Fuchs, Christian (2011). "Web 2.0, Prosumption, and Surveillance." In: *Surveillance & Society* 8.3. FWF (Austrian Science Fund) Project "Social Networking Sites in the Surveillance Society", project number P22445-G17, pp. 288–309 (cit. on p. 20).

Gadea, C. et al. (2011). "A Collaborative Cloud-Based Multimedia Sharing Platform for Social Networking Environments." In: *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–6. DOI: 10.1109/ICCCN.2011.6006079 (cit. on p. 22).

Garcin, Florent, Boi Faltings, Olivier Donatsch, et al. (2014). "Offline and Online Evaluation of News Recommender Systems at Swissinfo.Ch." In: *Proceedings of the 8th ACM Conference on Recommender Systems*. RecSys '14. Foster City, Silicon Valley, California, USA: ACM, pp. 169–176. ISBN: 978-1-4503-2668-1. DOI: 10.1145/2645710.2645745. URL: http://doi.acm.org/10.1145/2645710.2645745 (cit. on p. 56).

Garcin, Florent, Boi Faltings, Radu Jurca, et al. (2009). "Rating Aggregation in Collaborative Filtering Systems." In: *Proceedings of the Third ACM Conference on Recommender Systems*. RecSys '09. New York, New York, USA: ACM, pp. 349–352. ISBN: 978-1-60558-435-5. DOI: 10.1145/1639714.1639785. URL: http://doi.acm.org/10.1145/1639714.1639785 (cit. on pp. 39, 44, 46).

Garrett, Jesse James (2005). *Ajax: A New Approach to Web Applications*. URL: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/ (visited on 02/28/2017) (cit. on p. 5).

Goldberg, David et al. (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. 36).

Grudin, Jonathan (1994). "Computer-Supported Cooperative Work: History and Focus." In: *Computer* 27.5, pp. 19–26. ISSN: 0018-9162. DOI: 10.1109/2.291294. URL: http://dx.doi.org/10.1109/2.291294 (cit. on p. 14).

Herlocker, Jon, Joseph A. Konstan, and John Riedl (2002). "An Empirical Analysis of Design Choices in Neighborhood-Based Collaborative Filtering Algorithms." In: *Inf. Retr.* 5.4, pp. 287–310. ISSN: 1386-4564. DOI: 10.1023/A:1020443909834. URL: http://dx.doi.org/10.1023/A:1020443909834 (cit. on p. 40).

Herlocker, Jonathan L. et al. (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. 43, 53, 54).

Hofmann, Thomas (2004). “Latent Semantic Models for Collaborative Filtering.” In: *ACM Trans. Inf. Syst.* 22.1, pp. 89–115. ISSN: 1046-8188. DOI: 10.1145/963770.963774. URL: http://doi.acm.org/10.1145/963770.963774 (cit. on p. 47).

Hu, Yifan, Yehuda Koren, and Chris Volinsky (2008). “Collaborative Filtering for Implicit Feedback Datasets.” In: *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*. ICDM ’08. Washington, DC, USA: IEEE Computer Society, pp. 263–272. ISBN: 978-0-7695-3502-9. DOI: 10.1109/ICDM.2008.22. URL: http://dx.doi.org/10.1109/ICDM.2008.22 (cit. on pp. 32, 47).

Isinkaye, FO, YO Folajimi, and BA Ojokoh (2015). “Recommendation systems: Principles, methods and evaluation.” In: *Egyptian Informatics Journal* 16.3, pp. 261–273 (cit. on pp. 29, 30, 32).

Jaindl, Stefan (2016). “Social Media Software Integration for the Symfony Web Framework and Android and iOS Versions of the Catrobat Project.” MA thesis. Graz University of Technology (cit. on pp. 11, 22).

Johansen, Robert (1988). *GroupWare: Computer Support for Business Teams*. New York, NY, USA: The Free Press. ISBN: 0029164915 (cit. on pp. 15, 16).

Johnson, Christopher C. (2014). “Logistic Matrix Factorization for Implicit Feedback Data.” In: (cit. on p. 47).

Karwin, Bill (2010). *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. 1st. Pragmatic Bookshelf. ISBN: 1934356557, 9781934356555 (cit. on p. 81).

Kohavi, Ron and Roger Longbotham (2015). “Online Controlled Experiments and A/B Tests.” In: *Encyclopedia of Machine Learning and Data Mining* (cit. on p. 55).

Konstan, Joseph A. et al. (1997). “GroupLens: Applying Collaborative Filtering to Usenet News.” In: *Commun. ACM* 40.3, pp. 77–87. ISSN: 0001-0782. DOI: 10.1145/245108.245126. URL: http://doi.acm.org/10.1145/245108.245126 (cit. on p. 36).

Koren, Yehuda (2010). “Factor in the Neighbors: Scalable and Accurate Collaborative Filtering.” In: *ACM Trans. Knowl. Discov. Data* 4.1, 1:1–1:24. ISSN:

1556-4681. DOI: 10.1145/1644873.1644874. URL: http://doi.acm.org/10.1145/1644873.1644874 (cit. on p. 47).

Koren, Yehuda, Robert Bell, and Chris Volinsky (2009). "Matrix Factorization Techniques for Recommender Systems." In: *Computer* 42.8, pp. 30–37. ISSN: 0018-9162. DOI: 10.1109/MC.2009.263. URL: http://dx.doi.org/10.1109/MC.2009.263 (cit. on p. 133).

Laal, Marjan and Seyed Mohammad Ghodsi (2012). "Benefits of collaborative learning." In: *Procedia - Social and Behavioral Sciences* 31, pp. 486–490. ISSN: 1877-0428. DOI: http://dx.doi.org/10.1016/j.sbspro.2011.12.091. URL: http://www.sciencedirect.com/science/article/pii/S1877042811030205 (cit. on p. 23).

Lam, Shyong K. and John Riedl (2004). "Shilling Recommender Systems for Fun and Profit." In: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. New York, NY, USA: ACM, pp. 393–402. ISBN: 1-58113-844-X. DOI: 10.1145/988672.988726. URL: http://doi.acm.org/10.1145/988672.988726 (cit. on p. 49).

Levy, Pierre (1997). *Collective Intelligence: Mankind's Emerging World in Cyberspace*. Cambridge, MA, USA: Perseus Books. ISBN: 0306456354 (cit. on p. 8).

Linden, Greg, Brent Smith, and Jeremy York (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. 40).

Liu, Haifeng et al. (2014). "A New User Similarity Model to Improve the Accuracy of Collaborative Filtering." In: *Know.-Based Syst.* 56, pp. 156–166. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2013.11.006. URL: http://dx.doi.org/10.1016/j.knosys.2013.11.006 (cit. on p. 45).

Mahmood, Tariq and Francesco Ricci (2009). "Improving Recommender Systems with Adaptive Conversational Strategies." In: *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*. HT '09. Torino, Italy: ACM, pp. 73–82. ISBN: 978-1-60558-486-7. DOI: 10.1145/1557914.1557930. URL: http://doi.acm.org/10.1145/1557914.1557930 (cit. on p. 27).

Marusteri, Marius et al. (2015). "Challenges in the Design and Development of a ?Third Generation? E-Learning/Educational Platform." In: *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global, pp. 1369–1379 (cit. on p. 24).

Meteren, Robin van and Maarten van Someren (2000). “Using Content-Based Filtering for Recommendation.” In: (cit. on p. 34).

Miyahara, Koji and Michael J. Pazzani (2000). “Collaborative Filtering with the Simple Bayesian Classifier.” In: *Proceedings of the 6th Pacific Rim International Conference on Artificial Intelligence*. PRICAI’00. Melbourne, Australia: Springer-Verlag, pp. 679–689. ISBN: 3-540-67925-1. URL: http://dl.acm.org/citation.cfm?id=1764967.1765055 (cit. on p. 47).

Mnih, Andriy and Yee Whye Teh (2011). “Learning Item Trees for Probabilistic Modelling of Implicit Feedback.” In: *CoRR* abs/1109.5894 (cit. on p. 47).

Monroy-Hernández, Andrés (2007). “ScratchR: Sharing User-generated Programmable Media.” In: *Proceedings of the 6th International Conference on Interaction Design and Children*. IDC ’07. Aalborg, Denmark: ACM, pp. 167–168. ISBN: 978-1-59593-747-6. DOI: 10.1145/1297277.1297315. URL: http://doi.acm.org/10.1145/1297277.1297315 (cit. on pp. 6, 23, 79).

Monroy-Hernández, Andrés (2012). “Designing for Remixing: Supporting an Online Community of Amateur Creators.” PhD dissertation. Massachusetts Institute of Technology (cit. on p. 134).

Nielsen Norman Group (2017). URL: http://www.nngroup.com/articles/conversion-rates/ (visited on 02/28/2017) (cit. on p. 5).

Oard, Douglas and Jinmook Kim (1998). “Implicit Feedback for Recommender Systems.” In: *in Proceedings of the AAAI Workshop on Recommender Systems*, pp. 81–83 (cit. on p. 32).

OnPageWiki (2017). URL: https://de.onpage.org/wiki/Click-Through-Rate (visited on 02/28/2017) (cit. on p. 5).

Page, Lawrence et al. (1999). *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab. URL: http://ilpubs.stanford.edu:8090/422/ (cit. on pp. 12, 13).

Pazzani, Michael J. and Daniel Billsus (1997). “Learning and Revising User Profiles: The Identification ofInteresting Web Sites.” In: *Mach. Learn.* 27.3, pp. 313–331. ISSN: 0885-6125. DOI: 10.1023/A:1007369909943. URL: http://dx.doi.org/10.1023/A:1007369909943 (cit. on pp. 34, 35).

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: http://dx.doi.org/10.1007/978-3-540-72079-9_10 (cit. on pp. 34, 35).

Reid, Chris R. et al. (2015). "Army ants dynamically adjust living bridges in response to a cost?benefit trade-off." In: *Proceedings of the National Academy of Sciences* 112.49, pp. 15113–15118. DOI: 10.1073/pnas.1512241112. eprint: http://www.pnas.org/content/112/49/15113.full.pdf. URL: http://www.pnas.org/content/112/49/15113.abstract (cit. on p. 7).

Rendle, Steffen et al. (2009). "BPR: Bayesian Personalized Ranking from Implicit Feedback." In: *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*. UAI '09. Montreal, Quebec, Canada: AUAI Press, pp. 452–461. ISBN: 978-0-9749039-5-8. URL: http://dl.acm.org/citation.cfm?id=1795114.1795167 (cit. on p. 47).

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. 36).

Resnick, Paul and Hal R. Varian (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 pp. 27, 28, 36, 49).

Ricci, Francesco et al. (2010). *Recommender Systems Handbook*. 1st. New York, NY, USA: Springer-Verlag New York, Inc. ISBN: 0387858199, 9780387858197 (cit. on pp. 27, 28, 34, 56).

Richardson, W. (2006). *Blogs, Wikis, Podcasts, and Other Powerful Web Tools for Classrooms*. Annales. Nouv. sér. Sciences, medecine. SAGE Publications. ISBN: 9781412927673. URL: https://books.google.at/books?id=6PFjF9BQe2AC (cit. on p. 21).

Richman, Louis S. and Julianne Slovak (1987). "Software Catches the Team Spirit." In: *Fortune Magazine*, pp. 128–136 (cit. on p. 14).

Salakhutdinov, Ruslan, Andriy Mnih, and Geoffrey Hinton (2007). "Restricted Boltzmann Machines for Collaborative Filtering." In: *Proceedings of the 24th International Conference on Machine Learning*. ICML '07. Corvalis, Oregon, USA: ACM, pp. 791–798. ISBN: 978-1-59593-793-3. DOI: 10.1145/1273496.1273596. URL: http://doi.acm.org/10.1145/1273496.1273596 (cit. on p. 47).

Sarin, Sunil K. and Irene Greif (1984). "Software for Interactive On-line Conferences." In: *Proceedings of the Second ACM-SIGOA Conference on Office Information Systems*. COCS '84. New York, NY, USA: ACM, pp. 46–58. ISBN: 0-89791-140-7. DOI: 10.1145/800023.808333. URL: http://doi.acm.org/10.1145/800023.808333 (cit. on p. 14).

Sarwar, Badrul et al. (2002). "Incremental SVD-Based Algorithms for Highly Scalable Recommender Systems." In: *5th International Conference on Computer and Information Technology (ICCIT), 2002*. University of Minnesota, Minneapolis, MN 55455, USA (cit. on p. 48).

Schafer, J. Ben, Dan Frankowski, et al. (2007). "The Adaptive Web." In: ed. by Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. Berlin, Heidelberg: Springer-Verlag. Chap. Collaborative Filtering Recommender Systems, pp. 291–324. ISBN: 978-3-540-72078-2. URL: http://dl.acm.org/citation.cfm?id=1768197.1768208 (cit. on pp. 38, 51, 95).

Schafer, J. Ben, Joseph Konstan, and John Riedl (1999). "Recommender Systems in e-Commerce." In: *Proceedings of the 1st ACM Conference on Electronic Commerce*. EC '99. Denver, Colorado, USA: ACM, pp. 158–166. ISBN: 1-58113-176-3. DOI: 10.1145/336992.337035. URL: http://doi.acm.org/10.1145/336992.337035 (cit. on p. 33).

Segaran, Toby (2007). *Programming Collective Intelligence*. O'Reilly Media (cit. on p. 11).

Shani, Guy and Asela Gunawardana (2009). *Evaluating Recommender Systems*. Tech. rep. URL: https://www.microsoft.com/en-us/research/publication/evaluating-recommender-systems/ (cit. on pp. 52, 55–57).

Su, Xiaoyuan and Taghi M. Khoshgoftaar (2009). "A Survey of Collaborative Filtering Techniques." In: *Adv. in Artif. Intell.* 2009, 4:2–4:2. ISSN: 1687-7470. DOI: 10.1155/2009/421425. URL: http://dx.doi.org/10.1155/2009/421425 (cit. on p. 47).

Tapscott, Don and Anthony D. Williams (2006). *Wikinomics: How Mass Collaboration Changes Everything*. Portfolio Hardcover. ISBN: 1591841380 (cit. on pp. 9, 10).

Techopedia (2017a). *Technology Dictionary*. URL: http://www.techopedia.com/definition/32512/overfitting (visited on 05/08/2017) (cit. on p. 6).

Techopedia (2017b). *Technology Dictionary*. URL: http://www.techopedia.com/definition/1312/representational-state-transfer-rest (visited on 05/08/2017) (cit. on p. 6).

Terms, Tech (2017a). *The Tech Terms Computer Dictionary*. URL: http://techterms.com/definition/algorithm (visited on 05/08/2017) (cit. on p. 5).

Terms, Tech (2017b). *The Tech Terms Computer Dictionary*. URL: http://techterms.com/definition/hyperlink (visited on 05/08/2017) (cit. on p. 6).

Terms, Tech (2017c). *The Tech Terms Computer Dictionary*. URL: https://techterms.com/definition/web20 (visited on 05/08/2017) (cit. on p. 6).

Teufel, Stephanie et al. (1995). *Computerunterstützung für die Gruppenarbeit*. Bonn: , Addison-Wesley. ISBN: 3-89319-878-4 (cit. on pp. 17, 18).

The Google Search Engine (2017). URL: http://www.google.com (visited on 02/22/2017) (cit. on p. 12).

Toffler, A. (1980). *The Third Wave*. Morrow. ISBN: 9780688035976. URL: https://books.google.at/books?id=ViRmAAAAIAAJ (cit. on p. 20).

Tsuji, Keita et al. (2012). "Use of Library Loan Records for Book Recommendation." In: *Proceedings of the 2012 IIAI International Conference on Advanced Applied Informatics*. IIAI-AAI '12. Washington, DC, USA: IEEE Computer Society, pp. 30–35. ISBN: 978-0-7695-4826-5. DOI: 10.1109/IIAI-AAI.2012.16. URL: http://dx.doi.org/10.1109/IIAI-AAI.2012.16 (cit. on p. 34).

Verstrepen, Koen (2015). "Collaborative Filtering with Binary, Positive-only Data." PhD thesis. University of Antwerpen (cit. on p. 47).

Vicknair, Chad et al. (2010). "A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective." In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM SE '10. Oxford, Mississippi: ACM, 42:1–42:6. ISBN: 978-1-4503-0064-3. DOI: 10.1145/1900008.1900067. URL: http://doi.acm.org/10.1145/1900008.1900067 (cit. on p. 80).

Wang, Yuanyuan, Stephen Chi-Fai Chan, and Grace Ngai (2012). "Applicability of Demographic Recommender System to Tourist Attractions: A Case Study on Trip Advisor." In: *Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology - Volume 03*. WI-IAT '12. Washington, DC, USA: IEEE Computer Society, pp. 97–101. ISBN: 978-0-7695-4880-7. DOI: 10.1109/WI-IAT.2012.133. URL: http://dx.doi.org/10.1109/WI-IAT.2012.133 (cit. on p. 34).

Wells, Herbert George (1938). *World Brain*. ISBN: 1568493827 (cit. on p. 7).

Wheeler, William Morton (1911). "The ant-colony as an organism." In: *Journal of Morphology* 22.2, pp. 307–325 (cit. on p. 7).

Wit, Joost de (2008). *Evaluating recommender systems : an evaluation framework to predict user satisfaction for recommender systems in an electronic programme guide context*. URL: http://essay.utwente.nl/59711/ (cit. on pp. 53, 56).

Yao, Guanwen and Lifeng Cai (2015). "User-Based and Item-Based Collaborative Filtering Recommendation Algorithms Design." In: (cit. on pp. 37, 38, 40).

Yin, Hongzhi et al. (2012). "Challenging the Long Tail Recommendation." In: *Proc. VLDB Endow.* 5.9, pp. 896–907. ISSN: 2150-8097. DOI: 10.14778/2311906.2311916. URL: http://dx.doi.org/10.14778/2311906.2311916 (cit. on p. 28).

Yu, Kai et al. (2004). "Probabilistic Memory-Based Collaborative Filtering." In: *IEEE Trans. on Knowl. and Data Eng.* 16.1, pp. 56–69. ISSN: 1041-4347. DOI: 10.1109/TKDE.2004.1264822. URL: http://dx.doi.org/10.1109/TKDE.2004.1264822 (cit. on p. 49).

Zhao, Zhi-Dan and Ming-sheng Shang (2010). "User-Based Collaborative-Filtering Recommendation Algorithms on Hadoop." In: *Proceedings of the 2010 Third International Conference on Knowledge Discovery and Data Mining*. WKDD '10. Washington, DC, USA: IEEE Computer Society, pp. 478–481. ISBN: 978-0-7695-3923-2. DOI: 10.1109/WKDD.2010.54. URL: http://dx.doi.org/10.1109/WKDD.2010.54 (cit. on pp. 38, 40).