Thesis

RACE:GP – a Generic Approach to Automatically Creating and Evaluating Hybrid Recommender Systems

by

A.J. van Ramshorst

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on July 16, 2021 at 14:00

Student number:	4226631	
Project duration:	August 1, 2020 – July 16, 2021	
Thesis committee:	Dr. N. Yorke-Smith,	TU Delft, supervisor
	Dr. J. Yang,	TU Delft, co-supervisor
	Prof. dr. P.A.N. Bosman	TU Delft, thesis committee

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

In recent years, recommender systems have become a fundamental part of our online experience. Users rely on such systems in situations with many potential choices, such as watching a movie on a streaming service, reading a blog post, or listening to a song. Traditionally, these systems use techniques such as collaborative filtering and content-based recommendation. Both approaches have disadvantages, so to reduce those, recent research combines various techniques in different ways to create hybrid recommender systems. Creating a well-performing hybrid recommender system generally requires extensive knowledge of recommender systems, the domain on which one wants to provide recommendations, and trial and error. Automating this process makes recommender systems accessible for organizations that lack the resources to build these systems themselves. However, there is a lack of research regarding automating this process. This study aims to provide an initial exploration into this area by proposing RACE:GP, an end-to-end approach that automatically produces accurate non-trivial hybrid recommender systems with only a dataset and a definition of 'relevance'. RACE:GP automatically creates a programming language from a dataset in which any valid program is a recommender system on that dataset. By defining the relevant interaction, it can automatically evaluate the accuracy of these programs. It uses a search strategy based on genetic programming to find the best performing recommender systems in the language. To test our hypothesis, RACE:GP is used to produce recommender systems on three well-known datasets in recommender systems literature, and the results are compared to baselines based on collaborative filtering. Additionally, to verify its adaptability, we analyzed the produced recommender systems given different recommendation scenarios. The results showed that RACE:GP is able to produce recommender systems that outperform our chosen baselines by a significant margin. Furthermore, analysis of the produced recommender systems on different recommendation scenarios within a dataset shows that it can find systems that are especially accurate in situations with different densities or recommending specific interactions in datasets that contain different interactions. These results suggest that RACE:GP is a viable and generally applicable approach that makes creating hybrid recommender systems accessible for anyone with a dataset.

Preface

Completing this thesis marks the end of my time as a student. The report that lies before you has been written to fulfill the graduation requirements of the Computer Science master at the Delft University of Technology. It is the result of 11 challenging months of research, programming, running experiments, and writing.

I would like to express my gratitude to my two supervisors. First of all, Neil, who supported me when we first met to discuss my thesis over a year ago and I proposed my ideas regarding automating the creation of recommender systems, a subject I have been interested in since the first time I built such a system in a second-year bachelors course. With your feedback, suggestions, and questions I was able to shape this idea into the report that lies in front of you. Also, for bringing me into contact with Jie, who, after a couple of fruitful conversations about recommender systems, agreed to be my second supervisor. Our conversations and the feedback you provided helped me immensely to stay on track and focus on the right things. Even though last year was a strange year, with a global pandemic that made meeting in person difficult and unfortunately removed many of the social aspects normally associated with writing a thesis, you have both made it a great experience.

I would also like to thank my parents for believing in me and supporting me in the decisions I have made, not just during this thesis, but for my last 9 years as a student. My path to graduation has not always been straightforward, but I am glad that (as far as I know) you agreed with the choices I have made.

Finally, getting to this point would not have been possible without the friends I have made during my time as a student. A special thanks to Club Memento, Huize Shawshank, and Maurits for taking this journey with me, as well as my childhood friends, with whom I am afraid I am stuck for life.

A.J. van Ramshorst
The Hague, July 2021

Contents

1 Introduction

An increasing proportion of the limited amount of time we have in our lives is spent online. If we also have to spend most of that time sifting through the enormous amount of available content, any chance of ever being productive again would be practically non-existent. Fortunately, in conjunction with the growth of the internet, recommender systems have been developed. These systems assist users by filtering and ranking the enormous amounts of videos, films, songs, blogs, products, comments, and memes so that we are not required to view, listen to, read, and buy them all. Since every user has their own preferences, recommending these items is not trivial: these recommender systems must provide personalized recommendations for each user.

The concept of recommendation (or filtering, as it was initially called) has existed since before the world wide web, with the first well-documented approach using 'collective human knowledge' being Tapestry in 1992 [13]. In the years after that, research into recommender systems soared, gaining especially significant traction in 2006, when Netflix offered the 'Netflix prize' to the first researchers that could improve the performance of Netflix's 'Cinematch' algorithm by at least 10% [7]. Over 20,000 teams worldwide participated, collectively submitting over 13,000 recommender systems in the first two years. Three years later, the 10% improvement border was crossed by a matrix-factorization-based algorithm called BellKor, awarding the creators the prize of 1 million dollars [28].

Since then, research in this area has far from halted and has moved into many new directions [33, 44]. Active areas of research are, for example, based around scalability [11, 49, 50], explainability [1, 3, 5], and applying novel techniques such as deep learning [58] and neural networks [22] to (parts of) recommender systems.

The field of recommendation is clearly extensively studied over the last couple of decades, and recommender systems are already widely applied online. However, creating these systems and applying them in practice still requires considerable knowledge of recommender systems, the domain it is applied on, and how to effectively evaluate the performance of these systems. Additionally, trial and error is often required to find systems that perform. In short, effectively creating and utilizing recommender systems in practice requires a lot of time and money. While there has been some research trying to create a standard framework for creating recommender systems [26, 46], it is mainly focused on standardizing recommender systems for academic purposes and not on simplifying or even automating the process of creating well-performing recommender systems for specific situations. Automating the creation of recommender systems for any situation would make it feasible for anyone with data to produce meaningful recommendations for the users in their system. This thesis addresses that gap by proposing an approach based on genetic programming that automatically creates novel and diverse hybrid recommender systems, given *any* dataset as input.

1.1. Problem description

Recommender systems are generally based on the concept of collaborative filtering [8]. This is a technique based on the idea that if people had a similar preference in the past, they will have a similar preference in the future. For example, in a movie recommendation system, if Alice and Bob both liked 'The Shawshank Redemption', 'Pulp Fiction' and 'Back to the Future 2', and Bob also liked 'The Dark

Knight', it is likely that Alice will like 'The Dark Knight' as well. Another approach to recommendation is content-based recommendation, which uses properties of items to recommend items. In the same example with movies, it could recommend Bob 'Back to the Future 3', since it has mostly the same actors and director as its predecessor, which Bob also liked. Both approaches have drawbacks: collaborative filtering never recommends items that are not yet rated by anyone, and content-based filtering does not produce very diverse recommendations. Additionally, two movies having similar properties does not necessarily translate to liking both. For example, while 'Terminator 3' is similar on paper to its critically acclaimed predecessors, it is generally regarded as inferior. Both content-based and collaborative filtering approaches require users to rate several movies before relevant recommendations can be made. When a new user enters the system, recommendation strategies such as 'popular' can be used, which recommends items that are liked by other users not necessarily similar to you. The obvious disadvantage is that it does not address a user's particular taste.

Hybrid recommender systems combine different recommendation techniques to reduce or eliminate the weaknesses of individual techniques. Within this thesis, we make a distinction between trivial and non-trivial hybrid recommender systems. We define trivial hybrid recommender systems as systems that recommend items based on a weighted linear aggregation of the recommendations of multiple recommender systems. Non-trivial hybrid recommender systems combine two or more recommendation techniques in novel ways, such as using content-based filtering on the output of a collaborative filtering system. Research shows that these systems can be very successful in many different situations and domains [9].

Unfortunately, designing non-trivial hybrid recommender systems is not trivial. In addition to being aware of recommendation techniques and the possibilities regarding combining them, domain-specific knowledge is required. Even with this information, creating these systems generally requires a lot of trial and error to get right, and even then they still only work for a single application. For every new application or situation, this process has to be started from scratch. This process is costly and time-consuming, making it infeasible for smaller organizations to incorporate recommendations in their domain effectively. Automating the design and evaluation of recommender systems would enable these parties to effectively incorporate recommendations into their application without investing the resources generally associated with effective hybrid recommender systems.

Before we can verify whether our approach can automate this process, we need to define what 'automating the creation of hybrid recommender systems' means. Since our main goal is to make implementing recommender systems accessible for everyone, our approach must be both generally applicable and usable by anyone. For example, a webshop (1) designer that wants to suggest *exactly 3* (2) relevant products a user might want to *buy during checkout* (3), must be able to create an accurate recommender system with just these two decisions and a connection with the database (4). With this in mind, we can translate our goal into the following four general requirements that an approach to successfully achieve this goal must address:

1. It should make it trivial to create a non-trivial hybrid recommender system for *any* dataset without any domain knowledge or knowledge of hybrid recommender systems in general.

2. It should automatically create recommender systems that perform specifically well on certain recommending scenarios, such as when a system can only recommend a single item or when all potentially relevant items must be recommended.

3. It should be able to automatically design recommender systems that accurately recommend items based on a specific 'relevant' interaction type within a dataset, such as *buying an item* versus *reviewing an item*.

4. It should create a recommender system that is as accurate as possible given the data density of the actual dataset used to recommend items.

While the research regarding recommender systems is endless, not much has been done regarding generalizing the concept, and even less about automating the process of designing these systems. As far as we are aware, no research has been done that provides a solution that solves each of these requirements, and thus automates the entire process of creating recommender systems. The goal of this thesis is to be an initial exploration into this area of research by providing an end-to-end approach that addresses each of the requirements and experimentally verify its potential.

In this thesis, we introduce RACE:GP, which stands for **R**ecommender systems: **A**utomatic **C**reation and **E**valuation using **G**enetic **P**rogramming. RACE:GP can easily be extended to work as a service, where users can plug in any dataset and minimal configuration regarding 'relevance' to produce a system that performs well in that exact situation. It enables anyone to easily implement hybrid recommendation systems in any situation without the normally required knowledge, resources, and enormous datasets.

1.2. Automating hybrid recommender system design

Hybrid recommender systems are created by combining different recommendation techniques and data in many different ways. For example, using the output of one system as input for another, combining different interactions, such as viewing product details and adding a product to a wishlist, with different weights to build more accurate clusters of similar users, or use some properties of items to increase or decrease the impact that ratings of similar users have on the predicted rating. In this thesis, we propose a method to describe these interactions between different (parts of) recommender systems as functions in a 'programming language'. Given any dataset, we can use these functions, in addition to functions that read properties or interactions from the dataset, to define a language in which one can define hybrid recommender systems for that specific dataset.

RACE:GP is an algorithm that takes a dataset, an interaction to recommend, and a recommendation strategy as input and produces a recommender system capable of accurately recommending items to users for whom it expects that the requested interaction should exist. RACE:GP uses genetic programming, a heuristic search technique for programs in a language based on evolutionary algorithms [30], to efficiently find accurate recommender systems. To evaluate whether or not a program is an accurate recommender system, RACE:GP splits the dataset in a 'training' and a 'validate' set based on the requested interaction and compares the recommendations for each user against the 'validate' set.

This study hypothesizes that, given any dataset, RACE:GP can consistently find non-trivial hybrid recommender systems that outperform the baseline recommender systems *popularity*, *item-based collaborative filtering*, and *user-based collaborative filtering*. Additionally, for our algorithm to satisfy the aforementioned requirements, changing either the recommendation strategy, the requested interaction, or the density of the dataset should produce recommender systems that perform especially well in that specific situation compared to both baseline recommender systems and systems produced on different situations.

In this thesis, we experimentally verify the general applicability of RACE:GP by running experiments on three different datasets often used within recommender system literature. We analyze the results and the produced recommender systems to verify that our approach can produce non-trivial hybrid recommender systems that outperform the chosen baseline recommender systems. Additionally, we run multiple experiments where we change the requested interaction, dataset density, and recommendation strategy to verify that our approach meets these conditions.

The main contributions of this thesis are summarized as follows:

- We define a method to transform any dataset that can potentially be used in a recommendation scenario to a generic data model.
- We propose a set of functions that can be used to define the integration between different (parts of) hybrid recommender systems. Combined with the functions that can read properties and interactions from our generic data model, these make up a novel programming language that defines hybrid recommender systems on a specific dataset.
- We propose an algorithm based on strongly typed genetic programming that searches for the best performing recommender systems in the aforementioned language given a dataset, a requested interaction, and a recommendation strategy.
- We show that our approach consistently finds non-trivial hybrid recommender systems by reviewing the recommender systems produced on three different datasets. Compared to the strongest baselines, the best-performing recommender systems improved accuracy by $\sim 8\%$ on the Movielens dataset, $\sim 192\%$ on the Sobazaar dataset, and matched the baseline on the Filmtrust dataset.

- Finally, we demonstrate our approach's adaptability and general applicability by comparing recommender systems created for specific interactions, dataset densities, and recommendation strategies.

1.3. Thesis outline

Section 2 starts with a review of recent research and literature on (generic) hybrid recommender systems and the application of evolutionary algorithms on the problem of recommendation. Next, Section 3 explains in detail our approach to automatically generating recommender systems. The experiment to validate the method and quantify the performance on different recommendation scenarios is described in Section 4, and the results are shown in Section 5. In Section 6, we discuss the impact of these results, the limitations of the study, and recommendations for further research. Finally, we conclude our thesis in Section 7.

2

Literature review

This chapter aims to provide relevant background information regarding the techniques applied in this thesis. In addition to that, we aim to give an overview of how this research fits within the context of current research regarding recommender systems by giving an overview of recent research related to hybrid recommender systems, genetic programming and its application on recommendation, and generic approaches to recommender systems.

First, we introduce the concept of hybrid recommender systems and elaborate on the different types and some recent applications and developments. Next, we give a short overview of genetic programming, followed by recent applications of genetic programming and other evolutionary computing techniques in the context of recommender systems. Finally, we identify how this thesis fits in the context of recent research regarding automating the process of designing recommender systems.

2.1. Hybrid recommender systems

The problem of recommending relevant items to users given a limited amount of information has been studied for quite a while. Traditional recommender systems are generally based on some form of collaborative filtering (CF). Collaborative filtering, originally coined by Goldberg et al. [13], is a technique inspired by the idea that people generally prefer items that people with similar interests recommend. Other forms of filtering, such as content-based filtering, where items are filtered based on their properties, exist as well. Often, recommendation systems combine both techniques, resulting in *hybrid recommender systems* [9]. This section elaborates on the core concepts of hybrid recommender systems and gives an overview of recent research relevant to this study.

2.1.1. Overview

While there are other approaches regarding recommending [5, 45], at the basis of most recommender systems is the concept of collaborative filtering: Using the knowledge and experience of other users in the system to provide personal recommendations.

In online systems, ‘knowledge’ and ‘experience’ are usually represented in the form of ratings. While some literature uses explicit ratings, such as star ratings (1 means a user dislikes the movie, 5 means a user likes the movie, no value means a user did not rate the movie), often, implicit ratings such as ‘like a movie’ or ‘watch a movie’ are used. In this case, the absence of a ‘rating’ does not necessarily imply a negative association: a user might not have seen a movie yet. To generalize one step further and to get to the definition used in this thesis, we define rating as:

A rating is some interaction between a user and an item where the existence of that interaction indicates that an item is relevant in some context.

Collaborative filtering The most basic form of collaborative filtering (CF) is user-based CF, where the ratings in a system are used to find similar users, and then the items rated by similar users are recommended. A different, slightly better approach is item-based CF [47]. Item-based CF uses ratings to find related items based on the group of users that interacted with the item. The main advantage of item-based CF on user-based CF is that the relevancy between items tends to be more stable than

between users and thus can be precalculated to improve the performance of recommendations significantly. Additionally, systems tend to have fewer items than users, making calculating the similarity between items require less computing power and memory than between users.

The main drawback of collaborative filtering is that it suffers heavily from the cold-start problem. Both when a new user is added to the system and, even worse, when there is a new system where ratings in general are sparse. Item-based CF suffers slightly less when a new user is added since one or two ratings by that user enable the system to recommend similar items, but to accurately find similar users, more ratings are required.

Content-based filtering Since most datasets contain more information about items than just ratings, content-based filtering uses that information to recommend items with similar properties to items that the user rated in the past. Since properties of items are not provided by users in the system, but usually by the system itself, it suffers slightly less from the cold-start problem where there are not many ratings in a system, but it still cannot recommend an item to a user without ratings [9].

Other approaches Some systems also contain information about users, for example, age, gender, or education. Demographic information such as this can also be used to 'group' users. While this does not suffer from the individual cold-start problem, privacy-related issues make it difficult to collect enough information to make this feasible [2].

Another source of information that recommender systems can use is the context in which a rating happened. Context data can be anything, such as the timestamp of the rating, the location, the time of day, or the type of device. This information can both be used to improve the quality of collaborative filtering approaches, as well as to give context-aware recommendations in addition to personal recommendations.

Hybrid recommender systems To address the shortcomings of individual recommender systems, many researchers have tried different ways to combine multiple different techniques to create so-called *hybrid recommender systems*. It has been shown repeatedly that hybrid recommender systems can significantly increase the accuracy compared to using stand-alone recommender systems. For an extended overview of recent research related to hybrid recommender systems, we refer to Çano and Morisio [9]. Since this thesis aims to automate the creation of hybrid recommender systems, the next sections give an overview of techniques used in hybrid recommender systems, as well as some recent research regarding the use of auxiliary information.

2.1.2. Techniques used in hybrid recommender systems

There are multiple ways to combine information from different recommender systems. Çano and Morisio identify several different approaches to combining recommender systems, of which the most occurring are: weighted, feature combination, cascade, switching, and feature augmentation [9]. Refer to their survey for a detailed overview of research that uses each technique.

Weighted Weighted hybrid recommender systems aggregate the scores given by 2 or more recommender systems according to some set of weights. While some implementations just produce some linear combination of the scores given by different systems, some literature proposes training methods that produce different weights per user. Although this is one of the most straightforward approaches to hybrid recommender systems, it is also the one that is most mentioned in literature.

Feature combination Content-based recommendation uses 'features' to determine relevant items. Feature combination is a technique where the output of one recommender system, usually based on collaborative filtering, is added as a feature to a content-based recommender system.

Cascade Cascade recommender systems use a staged process. Often, one technique is employed to make an initial ranking of items, after which a second technique uses the ranked list to produce fine-tuned recommendations. For example, the first stage can be a general ranking technique, while the second stage personalizes these to produce a final ranking of items.

Switching Switching is when some criteria are used to determine which recommender system to use in a certain situation. For example, using a cold-start resistant technique, such as most popular, until a user has a certain number of ratings.

Feature augmentation While similar to feature combination, feature augmentation differs in that the idea is to improve the use of a feature in recommendation. For example, using a recommendation technique to predict ratings for users, and than use those predicted ratings instead of the actual ratings to find similar users for collaborative filtering.

2.1.3. Auxiliary information

Almost all hybrid recommender systems make use of more data than just explicit ratings. Usually, four categories of auxiliary data are recognized: properties of entities, demographic data, context data, and implicit ratings.

Demographic data With the advent of social media, and especially connecting social media to other online applications, many situations in which recommendation is applied have access to demographic user data such as age, gender, and location. Zhao et al. use demographic information to recommend products to buy to users by matching the demographic information extracted from their public profiles to demographic information from products based on blogs and reviews [59, 60]. It is also used in conjunction with collaborative filtering to solve the cold-start problem [38] or improve its accuracy [39, 56].

Context data Salah and Lauw use item context, such as information about items that are browsed in the same session or items bought together, to improve their model and reduce the impact of sparsity in the user-item context [45].

Implicit ratings and other interactions Often, recommender systems make use of explicit ratings. In reality, however, explicit ratings are expensive to retrieve. In contrast, implicit preferences, such as viewing an item, adding something to a wishlist, and viewing a video, are ‘cheap’ to collect. Cheap in this context means that users need (almost) no incentive to provide that information. These ‘implicit preference values’ can be used as input for recommendation systems as well [25].

Nguyen et al. propose a system that infers the explicit rating given the implicit interactions by users in a fashion app called Sobazaar [36]. Their dataset contains the implicit ratings: ‘click’, ‘want’, and ‘purchase’. In their approach, they manually associate explicit ratings to them, which are then used as input for multiple recommendation algorithms.

2.2. Genetic programming

In this thesis, we propose a method based on genetic programming [30]. Genetic programming (GP) is an *evolutionary computing* technique where a population of programs, in a specific language, ‘evolves’ to produce a program that performs a certain task as well as possible. Evolutionary computing is the collective name for algorithms and techniques inspired by nature and, more specifically, evolution.

2.2.1. Overview

Conceptually, genetic programming works by starting with a population of randomly generated programs, evaluating each program to determine their fitness, and then ‘breeding’ new programs by taking programs from the previous generation with some bias towards fitter individuals and combining them to produce the next generation.

Within the concept of genetic programming and related methods such as genetic algorithms, the fitness of an individual program is represented by its ‘fitness score’. The fitness score is a numerical expression of how well a program is able to do the task that is requested. In the case of genetic programming, the fitness score is determined by comparing the output of evaluating an individual program with some expected result.

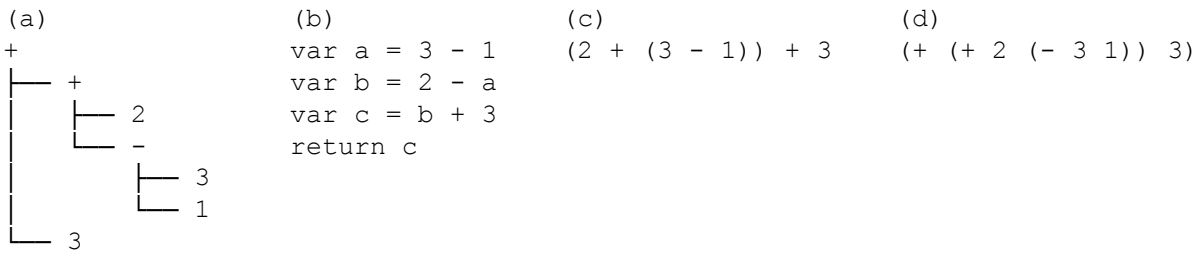


Figure 2.1: Example program in a programming language consisting of the terminals 1,2 and 3, and the functions +(a,b) and -(a,b) described in 4 different ways: (a) the tree structure used within this thesis, (b) pseudocode, (c) mathematical notation, and (d) a notation often used in the context of genetic programming.

A program While, in some applications of genetic programming, a program is represented as software with actual lines of code, more often, a program is represented as a directed acyclic graph, generally called a 'tree', consistent with functional programming concepts. Every node in the tree is some statement in the language. For example, if we assume a basic programming language suited to addition and subtraction. The statements it consists of are 1, 2, 3, + and -. These statements can be split into two types: functions (+, and -) and terminals (1, 2, and 3). Functions require 1 or more arguments. The amount of arguments a function takes is called its 'arity'. + and - have an arity of 2, which means they require 2 arguments as input. Both functions and terminals can be input. Terminals require no input. In the analogy of the tree, every leaf node is a terminal, and all other nodes are functions. The output of child nodes is used as the input of the parent node. To evaluate a program, start at the root and recursively evaluate the child nodes until you have the input, and then evaluate the function using the input to produce the output. See Figure 2.1 for a program in this language displayed in 4 different ways.

Genetic operators After evaluating the fitness of every individual in a generation, a new population is created based on the fittest individuals in the generation. The selected individuals are not copied exactly, but genetic operators are applied to introduce variation in the new generation. Generally, these genetic operators are crossover and mutation. Crossover is a technique where two parents are selected from the original generation, after which a randomly selected subtree from the first parent is switched with a randomly selected subtree from the second parent. Mutation is the process of randomly replacing a subtree in the offspring with a randomly generated subtree. Often, these methods are applied with a certain probability. Figure 2.2 shows an example of crossover and mutation on our previously defined language.

2.2.2. Genetic programming on domain-specific languages

Usually, genetic programming is applied on math-based languages or 'real' programming languages. In this thesis, we define our own 'domain-specific language' for recommender systems. There are some examples in literature where genetic programming is applied on domain-specific languages as well. For example, Hofmann [23] used genetic programming on a high-level programming language he designed to describe music compositions, and Barclay et al. [4] applied genetic programming on a programming language created to describe milling tool paths.

2.2.3. Application of evolutionary computing in recommendation

Additionally, to give a better impression of the landscape, we will mention here some applications of genetic programming and other evolutionary techniques on the problem of recommendations.

One of the applications of evolutionary computing and genetic algorithms and programming in general in recommender systems is to create personalized feature weighting strategies. Hwang et al. use a genetic algorithm for feature weighing the outputs of collaborative filtering in multi-criteria rating systems [24]. Following that, Gupta and Kant compare the genetic algorithm approach with genetic programming and show that although the search space is much larger, genetic programming performs at least as well as genetic algorithms, often improving given the same computational effort [20, 21].

Another application of evolutionary computing on hybrid recommender systems is by combining the outputs of different recommender systems. Da Silva et al. use a genetic algorithm to combine

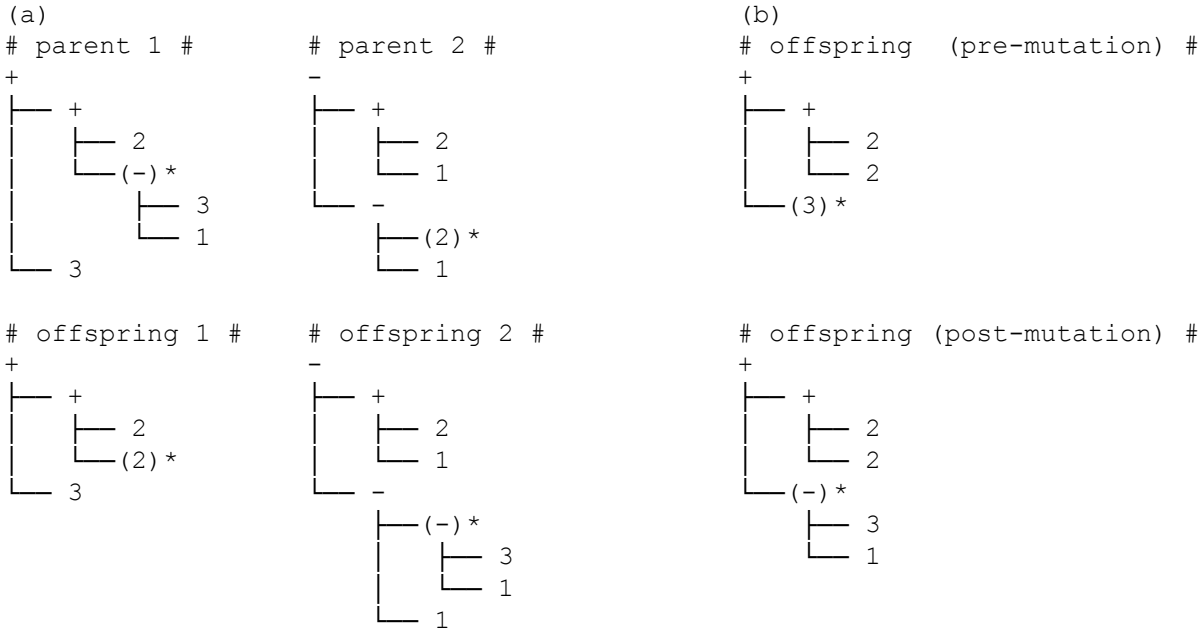


Figure 2.2: Example of crossover (a) and mutation (b) in our previously defined language. (x)* denotes the node where the operator is applied.

the output of multiple CF and CB approaches on the Movielens dataset and show that their approach outperforms all individual methods in each situation [12]. While they use an error-based evaluation metric, Oliveira et al. use genetic programming instead to combine the outputs of multiple rank-based recommender systems [37].

Finally, although less relevant in the context of this research but worth mentioning, some research uses genetic programming to train models within a single recommender system, such as Lara-Cabrera et al. (2020), who propose an alternative method to produce matrix factorizations using genetic programming [31].

While the aforementioned research uses genetic programming or similar techniques within the context of recommendation for things as personalized rating aggregation or personalized feature weighing, there is no research yet that uses genetic programming to 'build' hybrid recommender systems from scratch.

2.3. Generic recommender systems

Recent research on recommender systems often focuses specifically on the application of recommender systems within certain domains, such as movies, music, or webshops [33]. However, since the main goal of this thesis is to present a method that automatically produces a recommender system for any domain possible, this section will review the literature on generic recommendation systems.

Räck et al. propose a generic multipurpose recommender called AMAYA that recommends items based on ratings and clickstreams [42]. AMAYA defines a generic data model based on entities (which contain information on users or items), entries (a single unit of personalization data such as a rating or a click), and profiles, which are context-based subsets of entries associated with a user. The idea is that this model can be applied to any situation without needing to be adapted to specific applications, although the authors do not report any actual results, stating that the implementation is a work in progress.

Gupta and Tripathy provide an approach that uses a standard model in which any property can be defined as a number between 0 and 1 [19]. These properties are then used as input for training a neural network that builds user profiles, which are used as input for collaborative filtering. However, in their research, they only apply the approach on the Movielens dataset, and thus there are no results verifying that it actually is generic. Additionally, their approach uses an error-based evaluation metric, limiting the approach to situations in which numerical ratings are available.

Guo et al. created a framework called LibRec based on generic interfaces and data structures that makes it easy to run different recommendation algorithms and combine them with different evaluation techniques on any dataset [18]. While LibRec makes it easy to test and evaluate different recommender systems, systems need to be manually configured, and combining different approaches is not possible.

2.3.1. Automating the creation of hybrid recommender systems

Some of the research regarding generic recommender systems touches on the idea of automatically combining auxiliary information with user-based and item-based collaborative filtering. For example, Kouki et al. propose HyPER, a probabilistic framework for hybrid recommender systems that combines multiple similarity measures, auxiliary information, and existing recommender systems to produce accurate ratings [29]. Their approach consists of an interface to connect these sources, after which their algorithm learns how to balance them for accurate recommendations. Although their approach automates combining all available information, users of the system are still required to manually implement systems that calculate user and item similarities based on the datasets, as well as implement individual content-based and collaborative filtering methods.

Yu et al. proposed a similar approach based on knowledge graphs [57]. They suggest that item relationships with auxiliary information, such as actors, genres, and directors in the case of movie recommendation, can be used to improve recommendations, which they call the heterogeneous information network. They use this network to generate latent features for users and items, which is then used as input for a matrix factorization-based recommender system. Although they do not automate the entire process, the auxiliary information is automatically translated into a generic dataset that can be used by a generic recommender system.

Following their idea, Catherine and Cohen use the generic knowledge graph as input for a probabilistic logic system [10]. While the heterogeneous information network by Yu et al. required that nodes in the network had types, Catherine and Cohen propose two methods that work on untyped knowledge graphs. Their approach, based on discovering latent factors in the knowledge graph, outperforms the method by Yu et al. by a significant margin.

2.4. Conclusion

While there has been an enormous amount of research on recommender systems in the last couple of decades, literature regarding automatically designing and tuning hybrid recommender systems is scarce. Some of the research on generic recommender systems automate parts of the process, such as automatically combining different information sources, providing a framework to evaluate recommender systems, or automatically combining the outputs of multiple recommender systems. However, as far as we are aware, there is a clear research gap regarding the topic of automating the entire process of creating hybrid recommender systems for any dataset. This thesis aims to fill that gap by providing a starting point, possibly opening up the way for more research in this direction.

3

Methodology

This chapter elaborates on the methodology of applying genetic programming to the problem of automatically creating and evaluating recommender systems given any dataset. The first section will give a conceptual overview, followed by a description of the data model used as input and the programming language that describes a recommender system. Finally, we describe how we can determine the performance of a given program in that programming language and how the data model and programming language can be used within genetic programming.

3.1. Conceptual overview

In this chapter, we propose a method to automate this entire process in a generic way so that it can be applied to any dataset. The idea is to have a high-level programming language specifically for recommending items given a certain dataset in a generic data model and using genetic programming to find the best performing recommender system by only determining which type of interaction you want to recommend. We start by giving a high-level overview of the idea and the assumptions behind our approach of automating the process of creating recommender systems, and then we will elaborate on each part in more detail in the following sections of this chapter.

The first assumption is that any system in which some form of recommendation occurs can be modeled generically based on the notion of ‘entities’ and ‘interactions’. Examples of entities are ‘users’, ‘movies’ and ‘holiday destinations’, and examples of interactions are ‘rated’, ‘clicked’, ‘bought’ and ‘watched’. Both interactions and entities can have properties attached to them, such as ‘age’, ‘genres’ and ‘writer’ for entities and ‘rating’ or ‘date’ for interactions.

The second assumption is that given a dataset in this generic data model, we can define an approach to generate a domain-specific programming language specifically for recommender systems based on this dataset. The functions of this language consist of generic recommender system techniques translated into functions that operate on matrices and vectors, as well as some math-related functions, in addition to functions that transform properties and interactions from the dataset into vectors and matrices. Any valid program in this language is a recommender system for the dataset the language is based on.

The next assumption is that given a certain type of interaction in the system which we want to recommend, such as ‘buy product’, or ‘like the movie’, we can automatically create an evaluation function that uses the dataset to determine the accuracy with which a program recommends items for the requested interaction.

The final assumption is that given the three previous assumptions are true, we can use an automated search strategy, such as genetic programming, to create a recommender system given any dataset without needing any domain knowledge, recommender system knowledge, or trial and error, that performs at least as good as our baseline recommender systems based on collaborative filtering and popularity.

In this thesis, we present RACE:GP, which is a novel approach inspired by the aforementioned assumptions. In this chapter, we will describe how it works in detail.

3.2. Generic data model as input for recommender systems

The first step is defining a generic data model suitable for our purpose and providing an approach to transform any existing dataset into this generic model. This section will first explain the notion behind entities and interactions and then elaborate on different approaches to transforming an existing dataset into this model.

There are two reasons why a generic model is required. First, it is used to define the building blocks of a recommender system in terms of the dataset, and secondly, to automate the data preparation and processing that is required to evaluate the performance of a recommender system. Most recommender systems have a concept of 'users' and 'items', and some evaluation of items by users. This evaluation can be explicit, such as ratings or likes, or implicit, such as 'watching a video' or 'reading an article'. Sometimes additional information about the user (such as demographic information), the item (director of a movie, genre of a book), or the context of an evaluation (location where a product is bought, timestamp of a rating) is used as well. In our data model, all of this is captured in the concept of 'entities' and 'interactions'.

3.2.1. Entities

Usually, in recommender systems, a distinction is made between users and items. However, in our generic data model, we capture both the users and the items within a single concept: 'entities'. Anything in a dataset with properties can be an entity: a movie, a user, a holiday location, or an artist. Our model is not necessarily limited to the 'user' and 'item' entities. For example, on a website where people leave reviews of both movies and actors, we can have the entities 'user', 'movie', and 'actor'.

Every entity has a collection of properties associated with that entity. Examples of these properties are demographic information of users such as age, gender, or location, or details of a movie such as actors, the release date, genres, and director.

3.2.2. Interactions

The evaluation of items by users can be seen as an interaction between the user and the item: 'user rates a movie', 'user likes a book', 'user buys a product'. More often than not, a dataset consists of many types of these interactions that might be relevant in deciding whether or not an item is relevant for a user. For example, a blogging website might store the following interactions: 'user clicked on link of blog', 'user read blog', 'user liked blog', 'user commented on blog', 'user closed blog after 10 seconds'. The goal of recommending something is almost always making sure one of these interactions happen. In this case, the goal might be recommending a blog that the user reads, but depending on what the website wants to achieve, it can also be recommending a blog that the user likes.

Some interactions are unique, either the interaction has taken place or not, such as rating a movie or liking a picture. Other types of interactions can occur more than once, such as buying an item from the grocery store. Additionally, some interactions have an explicit evaluation associated with them, such as a star rating.

3.2.3. Transforming a dataset to the generic model

Now that the concept of interactions and entities are clear, we need a standard way of storing this information so that it can both be used to define the building blocks of our language and to standardize the evaluation process.

A dataset contains for each type of entity T a set of individuals E_T. The type of an entity defines what the potential properties are for each individual entity in the dataset. For each entity type, a dataset can contain many different individual entities of that type and the properties belonging to each individual. Since we mainly care about the properties of each individual, we can model each entity type as a set of vectors of length L, L being the number of individuals of that entity type in the dataset. Each vector contains for a specific property the value for each individual in the dataset (or empty if a property is not available). For example, if we have an entity type 'user', with the properties name, gender, and age, and three individuals in our dataset, the corresponding set of vectors would be E_{user}.

$$E_{user} = \{[\text{'Alice'}, \text{'Bob'}, \text{'Charlie'}], [F, M, M], [31, 42, 27]\}$$

For the entity type 'movie' with the properties title and year we can do the same:

$$E_{movie} = \{[\text{'Titanic'}, \text{'The Shawshank Redemption'}, \text{'The Room'}], [1997, 1994, 2003]\}$$

To model the interactions between two entities, we can use a sparse matrix where each row corresponds to the acting side of the interaction and each column to the receiving side. In the example of 'users' and 'movies', we have, for example, the interactions: 'watch', 'hide', and 'rate'. In these examples, the user is the 'actor', and the movie is the 'receiver' of the interaction:

$$I_{watch} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 3 \\ 1 & 0 & 0 \end{bmatrix}, I_{hide} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, I_{rate} = \begin{bmatrix} 0 & 0 & 5 \\ 3 & 5 & 5 \\ 1 & 0 & 4 \end{bmatrix}$$

Watching a movie can be done multiple times, so in that case, the value is the number of interactions. 'Hide' is an interaction that can only be done once by a user, so it is either 1 (hidden) or 0 (not interacted). Rating is done on a scale of 1 to 5, so the values can reflect that as well. In every case, 0 means that no interaction has taken place.

Properties versus entities In some situations, the line between properties and entities is unclear—for example, movies belonging to a genre or actors that play in a movie. On the one hand, genres and actors can be defined as a property of the entity 'movie' in the form of an array. On the other hand, genres and actors can be seen as an entity, and 'belonging to' a genre or 'acting in' a movie can be modeled as an interaction, where 1 means the relation exists, and 0 otherwise. In our experiments, we include both approaches.

3.3. A programming language for hybrid recommender systems

In the previous chapter we have determined that hybrid recommender systems use different methods to combine recommendation techniques, such as combining the outputs of multiple recommender systems, using the output of one system as input for another, or combining different types of information to be used as input for a recommender system. This suggests that we can model these methods of integration as functions in a language. If we add functions that perform recommendation techniques such as content-based comparisons and clustering, we create a generic language that can describe hybrid recommender systems. These functions can be combined with a set of terminals that can read information from a dataset D, as defined in the previous section, to create a language L_D in which any valid program is a recommender system for D. In this section, we first define the input and output types of this language, followed by a description of the functions and terminals that make up this novel, high-level language for recommender systems.

3.3.1. Input and output types

The first thing any language needs is a typing system. Therefore we start with defining the types a function or terminal can return. Since the generic data model uses vectors and matrices to store the properties and interactions of the dataset, it makes sense to base our typing system on this. Our language is strongly typed, which means that types are used in our language to make sure that every program is valid.

Entity properties As previously mentioned, entity properties are represented in the data model by vectors, so we need a vector type as well. The vector in our language can contain different types of properties, and it can specifically belong to a certain entity as well.

A basic example is the user's age: each item in the vector consists of the age corresponding to the user at that place in the vector. In that case the type is **Vector[User]**. Another option is the genres belonging to a movie: **Vector<String[]>[Movie]**.

Interaction properties Since interactions are modeled as a matrix where the rows and columns represent entities, we define the type **Matrix[A,B]**, where A and B are the entity types between which the interaction takes place. Our data model can, for example, contain **Matrix[User,Movie]** for ratings or likes or **Matrix[User,User]** for trust relations between users. While these examples come directly from the dataset to indicate some interaction, matrices can describe any relation between two entities. For

Name	Input	Output	Parameters
compareString	Vector<String>[A] Vector<String>[B]	Matrix[A,B]	
compareArray	Vector<Array>[A] Vector<Array>[B]	Matrix[A,B]	
popularity	Matrix[A,B]	Vector[B]	
pearsonSimilarity	Matrix[A,B]	Matrix[A,A]	
nearestNeighbour	Matrix[A,A] Matrix[A,B]	Matrix[A,B]	# of neighbours
nearestNeighbour(inverted)	Matrix[B,B] Matrix[A,B]	Matrix[A,B]	# of neighbours
transpose	Matrix[A,B]	Matrix[B,A]	
addVector	Matrix[A,B] Vector[B]	Matrix[A,B]	
scaleMatrix	Matrix[A,B]	Matrix[A,B]	Scalar
sumMatrix	Matrix[A,B] Matrix[A,B]	Matrix[A,B]	

Table 3.1: Selected functions for the language used by RACE:GP.

example, the matrix **Matrix[Movie,Movie]** can represent the overlap in genres between the movies, how similar the movies are in general, or the number of users that liked both movies.

Generic types In addition to this, it is useful for our language to support 'generic' versions of vectors and matrices so that we can add constants unrelated to the dataset as input/output for our functions. This means the following types are also possible: **Vector[*]**, **Matrix[*,*]**. Generic types are also used to specify the domain and codomain of some of the functions and terminals in our language.

3.3.2. Functions

Any programming language consists of functions. A function has 1 or more inputs of predetermined types and outputs a value of a specific type, which in some cases depends on the input. In our cases, the goal of the language is to be a very high-level language for recommender systems, so the functions included are based on ideas found in hybrid recommender systems. See Table 3.1 for an overview of the input, output, and configurable parameters of each function.

compareString Returns a matrix M with for each combination of strings in input V and W 1 if they are the same, or 0 otherwise.

$$M_{ij} = \begin{cases} 1, & \text{if } V_i = W_j \\ 0, & \text{otherwise} \end{cases}$$

compareArray Returns a matrix M where for each combination of arrays in the input V and W:

$$M_{ij} = \frac{|V_i \cap W_j|}{|V_i \cup W_j|}$$

popularity Returns a vector V for input M, where:

$$V_j = \sum_i M_{ij}$$

Name	Output	Parameters
interaction(interaction)	Matrix[A,B]	
property(entity.property)<String>	Vector<String>[A]	
property(entity.property)<Array>	Vector<Array>[A]	
property(entity.property)	Vector[A]	

Table 3.2: Available terminals. Each property or interaction in the dataset translates as one of these terminals.

pearsonSimilarity Returns a matrix M given input matrix N, where:

$$M_{ij} = \text{PCC}(N_{i*}, N_{j*})$$

Here, N_{j*} represents the jth row of matrix N, and PCC is the function that returns the Pearson correlation coefficient between two vectors [6].

nearestNeighbour Inspired by user-based collaborative filtering. Given input matrices $M[A, A]$ and $N[A, B]$, for each row i in M, it picks the n column indices $j \in J$ with the highest values M_{ij}. Then it returns a matrix P, where:

$$P_{ik} = \frac{\sum_{j \in J} M_{ij} N_{jk}}{n}$$

nearestNeighbour(inverted) Inspired by item-based collaborative filtering. Given input matrices $M[B, B]$ and $N[A, B]$, for each row i in M it picks the n column indices $j \in J$ with the highest values M_{ij}. Then it returns a matrix P, where:

$$P_{ik} = \sum_{j \in J} M_{kj} N_{ij}$$

transpose Given a matrix M, it returns M^T.

addVector Adds a vector V and matrix N, it returns a matrix M, where:

$$M_{ij} = N_{ij} + V_j$$

scaleMatrix Given a matrix N and a scalar s, returns sN.

sumMatrix Given two matrices $N[A, B], M[A, B]$, returns $N + M$

3.3.3. Terminals

While the functions make the language suitable for recommending, we also need to make it capable of recommending items in a certain dataset. This is where the terminals come in. We have two types of terminals, which are both dynamically generated based on the provided dataset: ‘interaction’ and ‘property’. An overview can be found in table 3.2.

interaction(<interaction>) Every interaction property in the dataset creates a terminal in the language for that specific property. It returns the matrix in the dataset for that property.

property(<entity.property>) Every entity property in the dataset creates a terminal in the language as well. This returns a vector containing the values for that property in the dataset.

empty Finally, we have an 'empty' terminal which is used in some situations as a fallback terminal. It returns a matrix filled with zeroes. This is never explicitly in a program, but due to sampling of the datasets, there can be situations where a certain entity or interaction terminal does not exist. In that case, it is replaced by this one.

3.3.4. Parameters in functions and terminals

Some of our functions require extra configuration in addition to the input matrices, such as scaleMatrix (the scalar) and nearestNeighbour (the number of neighbors to consider). In regular programming, as well as in the context of genetic programming, these 'parameters' are often added as an additional input value. We have decided to go against the norm and add the concept of 'configuration' to functions. The reason being that adding these values as function input would increase the search space exponentially while not making any sense from a genetic programming structure since these parameters are not really part of the program. Whenever a function is selected, the configuration is randomly picked from a predetermined range of sensible values.

3.3.5. The reasoning behind the selection of functions and terminals

To make our language as effective as possible, we aimed to select the smallest possible set of functions that make sense within the context of hybrid recommender systems. During the development of RACE:GP, we have experimented with a large number of additional functions and terminals, such as a constant terminal and additional math-related functions. Based on the initial results of our trials, we decided to remove anything that did not contribute directly to meaningful recommender systems.

Initially, the set of terminals included a 'constant' terminal, which would return either a Scalar, Vector[*] or matrix[*,*] where every value was the same. However, this lead to programs containing large subtrees that, for example, returned the similarity between rows in a matrix where every value is the same, which always returns a matrix filled with 1's. These subtrees pollute the search space, and because the return value is generic, they can be (and were) applied for every input.

For the same reason, we decided to remove all math-related functions except 'sumMatrix', 'addVector', and 'scaleMatrix'. Initially, our language contained 'sum', 'multiply' and 'subtract' for every possible input combination type. While reasoning about and reviewing literature regarding hybrid recommender systems, we could only map the three aforementioned functions consistently to concepts from hybrid recommender systems and thus removed all other math-related functions.

We do not claim that improving on the selected set of functions is impossible. However, our results indicate that they are able to describe accurate hybrid recommender systems quite well. Future research can be done to improve our selection.

3.4. Evaluating the fitness of a created recommender system

Now that we have defined our data model, a typing system, and a programming language based on that data model, the next step is to define a method to evaluate whether or not a program in that language is suitable to a certain task: the fitness function. Since a program in our language should be a recommending system, the fitness score of a program should represent its ability to accurately recommend relevant items to users. In this section, we will describe our approach to evaluating programs.

The first step is defining a generic way of determining what a 'relevant' recommendation is. The next step is preparing a dataset in our generic model so we can simulate a realistic recommendation scenario (and thus verify if recommended items are indeed relevant). Finally, we specify how we can use the prepared dataset to calculate an actual 'fitness value' of any program in our language.

3.4.1. Defining what relevant means

The goal of recommender systems is usually described as 'finding (a number of) relevant items for a specific user'. However, relevant can mean different things depending on the available data, the viewpoint of the user of such a system, or the viewpoint of the owner of a system. Since our approach should be able to create recommender systems for every situation, we need to generalize what we mean by 'relevant' in terms of the provided dataset.

Suppose we look at the goal of suggesting relevant items to a user. In practice, relevant means that we want the user to interact with these items. Different systems can have different types of interactions that determine 'relevant', such as 'recommend a movie to watch', 'a holiday destination to book', 'or a

book to buy'. Given this information, we define 'relevant' as: 'given an item and a requested interaction type, the likeliness that a user will have an interaction of that type if the opportunity presents itself'. This idea is the basis for our fitness function, and thus, together with the dataset, the requested interaction type is one of the inputs of our algorithm.

3.4.2. Preparing a dataset for evaluation

Now that we have defined what a relevant recommendation is, the next step is using that to process our dataset into a realistic simulation of reality. Normally, a recommender system works by providing the user with some options through a user interface, and if the user interacts with one of the provided options, the recommendation can be seen as relevant. However, unless you have access to massive amounts of users, evaluating the performance of a recommender system can take quite a while this way. In our case, we want to evaluate a system in near real-time. This section describes a method to automatically turn a dataset in our generic model combined with a requested interaction type into a 'training' and 'test' set. Using this approach, we can simulate a real situation, in addition to giving control over the size of the training set so that we can improve the speed of evaluation. On a high level, the process works as follows:

1. Given the requested interaction type, pick a sample of the requesting entity type of the interaction (generally the user).
2. For each sampled user:
   (a) Sort their interactions of the requested type on date if available.
   (b) Split their interactions of the requested type into a 'training' and 'validation' list and add the receiving entities for each interaction in the training set to the 'filter' list and those of the validation list to the 'relevant' list.
   (c) For every entity in the 'relevant' list, remove all interactions between the user and it from the dataset.

This preprocessing is necessary to achieve two goals: Reducing the size of the dataset and simulating a realistic recommendation scenario.

Sampling the dataset Sampling the dataset has two primary purposes: increase the efficiency of evaluation by reducing the runtime and memory usage of the program and reducing the risk of overfitting [14]. To generalize the sampling approach, the idea is to have some parameter $0 < s \leq 1$, which determines the probability that any user in the full dataset is included in the sample. For reproducibility, a seeded PRNG is used to decide which users to pick. The higher the value chosen for s, the more consistent the fitness scores over generations, but the longer the program takes to run. Choosing s close to 1 might increase the risk of overfitting since there is no change in the test set over multiple generations. If a user is not selected during the sampling process, all interactions related to that user are also removed from the dataset.

Simulating a recommendation scenario The best way to validate the performance of a recommender system is by testing it in the real world. You recommend something to a user, and if that user interacts with it, it is an accurate recommendation. As stated before, the time and resources necessary for this make it infeasible for our algorithm. Therefore we use a simulation of reality using the provided dataset. In general, the problem of 'simulating' realistic recommendation scenarios is tackled by making an assumption regarding what would constitute a relevant recommendation for each user based on the available data. Filtering some of these relevant items from the dataset and comparing them with the output of the recommender system enables us to calculate some accuracy score. Usually, the accuracy is defined by a metric such as 'is the selected relevant item in a set of n items provided by the recommender system?' or: 'given a ranking of items for a user, what is the index of the first relevant item?'.

To create a realistic scenario, we split each user's interactions into a set of 'training' and 'validation' interactions. The training interactions simulate the context in which the recommender system has to recommend an item, and the 'validation' interactions simulate the relevant recommendations. This is done per sampled user in three steps: sorting (optional), splitting, and cleaning.

First, if it is relevant for the specific dataset and some timestamp-like property is available, the interactions are sorted on time. This allows us to simulate recommending an item at a certain moment in time by splitting the resulting list at some point. In our experiments we select the first 80% of the interactions as training data, and the remaining 20% as validation interactions. If sorting makes no sense for that dataset, the previously mentioned PRNG is used to randomly select 20% of the interactions for each user as validation interactions.

Secondly, we add the entities related to the interactions in the training set to a 'filter' list since we don't want our program to recommend items that the user already interacted with and the entities for the validation interactions to the 'relevant' list.

Finally, we remove all interactions between the user and the entities in the 'relevant' list from the training dataset. This step is important for two reasons: first, in reality, we usually aim to recommend items with which the user has not yet interacted, and second, recommender systems that just recommend items with which you have already interacted perform unrealistically well. For example, users generally only 'tag' a movie after they have seen it, which often also means they rated it. A recommender system that recommends every movie that a user tagged makes no sense. However, if not all interactions are removed between users and movies in the 'relevant' set, our fitness functions would rate these recommender systems extremely high.

The terminals from the previous section that read from the dataset return only properties and interactions from the training dataset.

3.4.3. Evaluating a program

Now that we have a dataset prepared to efficiently simulate a real situation, we need to assign a fitness score to a program based on that dataset. This is done by first executing the program and then calculating the fitness score on the resulting matrix. To execute a program in our language, we start at the root expression, recursively evaluating the input expressions (if there is input), and then evaluate the expression itself given the input. For a program in our language to be valid according to the dataset and requested interaction type, the output must correspond to the interaction type. For example, if the requested interaction is 'user likes movie', the output must be of type **Matrix<User, Movie>**.

To transform the resulting matrix into a list of items based on relevancy for each user, we can return for each row the entities related to the values in the row, sorted descending based on the values for the corresponding entity. Since our dataset contains a list of the entities a user has already interacted with, the filter list, we can remove those entities from the resulting list for each user, which leaves us with for each user an ordered list of entities that this program recommends. The final evaluation step is then comparing that list for each user with their validation list using some metric based on the requirements. Taking the average of these values returns a fitness score for the entire program.

Selecting an evaluation metric should be done based on what the goals of recommending are. In some situations, the first or second thing you recommend should be relevant, and items lower on the list are ignored. In other situations, you might want to recommend every relevant item to the user, and missing one would be bad. Depending on the context, different evaluation metrics can be chosen, which can roughly be categorized in three ways: accuracy-based, decision-based, and rank-based. Accuracy-based metrics are mostly used in explicit rating situations where a system estimates the rating a user would give a certain item and compares that with what the user actually rated. Decision-based metrics work on the premise of splitting a dataset into relevant and irrelevant items, and finally, rank-based methods are based on the notion that relevant items should be ordered higher than irrelevant items on average.

Since accuracy-based metrics are only applicable in very specific situations, our approach uses precision and recall for decision-based evaluation and mean reciprocal rank for rank-based evaluation.

Mean reciprocal rank Mean reciprocal rank (MRR@n) sums for each relevant item in the validation list of size n the reciprocal of its position in the order outputted by the program. The MRR@n for an ordered list of items $e \in E$ is:

$$\text{MRR@}n = \sum_{i=1}^{i \leq n} \frac{1}{i} e_i, \text{ where } e_i = \begin{cases} 1, & \text{if item } i \text{ is relevant} \\ 0, & \text{otherwise} \end{cases}$$

Precision and recall Precision@n and Recall@n are metrics to determine how accurate a recommendation of a set of items is. Precision@n is the ratio of relevant items in n recommended items, and Recall@n is the ratio of all relevant items that are recommended in the first n recommended items. Both these values can be calculated using the top n relevant items according to the output and comparing them with the validation list.

3.4.4. Summary

In this section, we determined how we can evaluate the performance of any program in our language. We defined what it means for a recommendation to be relevant in our context, how the dataset is used to simulate a real recommendation scenario, and multiple ways to turn the output of a program into a numerical value describing the 'fitness' of a program.

3.5. Finding recommender systems using genetic programming

Now that we have a data model, a programming language, and a way to determine how accurate a program is, we can tackle the last step of our problem: Automating the process of finding a program that performs well. This problem can be split into two subproblems. First, we need a way to randomly generate valid programs, and secondly, an efficient way to search the entire space of available programs given a certain language for a good performing solution. This section will first describe how we can generate random valid programs, and we will then elaborate on how we can use genetic programming to efficiently search for well-performing programs.

3.5.1. Generating a random valid program

The next step of automating recommender systems is by providing a way to randomly generate a valid program in our specific language. In general, the approach to create a random program with a maximum depth is explained by the pseudocode in Algorithm 1. To create a program tree, the algorithm takes a maximum depth and a method. The maximum depth determines the allowed size of the tree, and thus the maximum number of functions that can be used recursively as input. After the maximum depth is reached, the algorithm always selects a terminal. There are two different approaches for creating a random tree: 'GROW' or 'FULL'. If 'GROW' is chosen and the maximum depth is not yet reached, the algorithm selects a random node from either the functions or terminals, but if 'FULL' is chosen, it always selects a function until the maximum depth is reached.

Algorithm 1 Basic program generation, F is list of available functions, T is list of available terminals

```
function GenerateProgram(max_depth, method)
    if max_depth = 1 then
        root ←Random(T)
    else if method = 'FULL' then
        root ←Random(F)
    else
        root ←Random(F ∪ T)
    end if
    for all arguments of root do generate a subtree:
        GenerateProgram(max_depth - 1, method)
    end for
end function
```

While this approach to program generation works if every function and terminal in the language can always be selected (for example, in the previously defined example language containing the functions + and -, and the terminals 1, 2, and 3), this is not the case in our situation. As stated before, our language is strongly typed: it contains the types **Matrix** and **Vector**. This means that terminals output a certain type, and functions require a certain type as input. The straightforward approach is to adjust

Input[0]	Input[1]	Output
Matrix<User,Item>	Matrix<User,Item>	Matrix<User,Item>
Matrix<User,Item>	Matrix<User,Actor>	Invalid input
Matrix<Actor,Item>	Matrix<User,Item>	Invalid input
Matrix<User,Item>	Matrix<*,*>	Matrix<User,Item>
Matrix<User,Item>	Matrix<User,*>	Matrix<User,Item>
Matrix<*,Item>	Matrix<User,*>	Matrix<User,Item>
Matrix<User,*>	Matrix<User,*>	Matrix<User,*>
Matrix<*,*>	Matrix<*,*>	Matrix<*,*>

Table 3.3: Example of output given certain input for function sumMatrix

the previous algorithm to add the required output type as an argument to the GenerateProgram function and filter the functions and terminals based on that output type. We can then generate a valid program in our situation by calling the GenerateProgram function with the type based on the interaction we want to recommend.

Unfortunately, this is not enough since our language also contains so-called ‘generic functions’. Generic functions are functions that can receive multiple (combinations of) valid input types and produce different output types based on the provided argument types. An example of a generic function in our language is the ‘sumMatrix’ function. If we assume our dataset has the entities: ‘User’, ‘Movie’ and ‘Actor’, some examples of inputs and the corresponding output for ‘sumMatrix’ are given in table 3.3. Types with a ‘*’ mean that they are still generic: until they are defined higher or lower in the program tree, they do not have a fixed type yet. This also shows that generic types can be ‘passed through’. This means that filtering the terminals and functions to check if they are potential valid inputs is not as straightforward anymore.

To solve this problem, we need to recursively check if it is possible for a function to produce the required output type based on the maximum depth remaining and the available terminals and functions in the language before we can add it. To achieve this, we use the strategy created by Montana [35], which uses a precalculated table to determine what the possible output types are for each level of a potential program and use that to determine what functions and terminals can be selected given a certain maximum depth. For a more in-depth explanation of this approach, please refer to their paper for a detailed explanation of how we implemented this.

3.5.2. Genetic programming overview

Now that we have a way to generate a random program that outputs a certain type, we can tackle the problem of efficiently searching for problems in the problem space. A naive solution is to repeatedly generate a program and evaluate it until a program with a fitness score above a certain threshold is found. While this can work, it is not very efficient. An alternative solution to this problem that can be more efficient is genetic programming. Genetic programming is an evolutionary computing technique that uses evolution-inspired approaches such as crossover, mutations, and pressured selection to iteratively produce generations of improving maximum and average fitness [30]. In our approach, genetic programming is applied as follows:

1. Generate an initial population of random programs in the language, using the ‘ramped-half-and-half’ method, where 50% of the programs are generated using the ‘GROW’ method, and 50% using the ‘FULL’ method. This ensures that a diverse population is created [34].

2. Generate a validation dataset used to validate the performance of the best-found RS every generation. The validation dataset is used to compare the performance of programs in multiple generations if interleaved sampling is used.

3. Repeat until a certain end-condition is reached (max generation, certain fitness value, or no improvement for a number of generations).

(a) Generate a random sample of the dataset, used to define the relative fitness of individuals in the current generation. This is done for the two reasons stated in the section on dataset preparation: reducing overfitting and reducing the time required to evaluate an individual.

(b) Evaluate each individual in a generation using the fitness function over this generation's sample dataset.

(c) Evaluate the best individual for this generation using the validation dataset, and if the fitness is better than the best found so far, replace the best found with this.

(d) Produce the next generation by selecting individuals using the fitness score over the sampled data and then applying crossover and mutation. This step is discussed in more detail in the next section.

4. Return the individual with the highest fitness value on the validation set found so far.

While most of these steps are standard genetic programming or covered in previous sections, such as the approach to evaluation, step 3d, reproduction, needs some elaboration.

3.5.3. Producing offspring

In item 3d in the previous section, the process of reproduction is mentioned. This process consists of two steps. First, a number of the fittest individuals from the previous generation is copied to the next generation, which is called elitism. After that, we pick two individuals using a selection procedure that prefers individuals with a higher fitness score and then combine these two individuals in some way (crossover) to produce two new individuals. Finally, to keep some diversity and reduce the chance that a local maximum is reached, some parts of the offspring can be randomly changed (mutation). Step two of this process is repeated until the size of the new generation is the same as the previous one. This section describes in more detail how elitism, selection, crossover, and mutation are implemented in our approach.

Elitism Elitism in genetic programming is used to make sure that the best individuals of a generation remain a part of the population. In our implementation, before we start selection and crossover and/or mutation, we select the best individuals from the current population, and without applying evolutionary operators, we put them in the next generation. The amount of individuals taken is based on the elitism ratio. An additional advantage of using elitism is that it reduces 'bloat' in later generations [40][54]. Bloat happens when over the course of generations, the average size of programs tends to increase.

Selection The goal of the selection step is to make sure that the average and maximum fitness of the next generation is higher than that of the current generation. This is done by preferring fitter individuals during selection, also called 'selection pressure'. While there exist multiple approaches to this, in our experiment, we use 'tournament selection'. To pick an individual, we randomly select k programs from the parent generation and pick from those the one with the highest fitness for reproduction. Since we need two individuals for crossover, we repeat this process to get the other parent.

Crossover After two individuals are selected, we can produce their offspring. The idea is that we select a node in both parents and replace the respective subtrees with each other to produce the offspring. Since our programs are strongly typed, this does not always produce a valid tree. Therefore it first selects a node in one parent and then randomly picks a subtree from the other parent that produces the same output type. If that does not exist, both parents are copied into the new generation. In case the depth of one of the offspring programs is larger than the configured maximum depth, only the offspring that falls within the restrictions remain. This approach is inspired by Montana [35].

Subtree mutation Subtree mutation is relatively straightforward. With a predefined probability, select a random node from a program, and generate a new subtree that falls within the maximum depth with the same output type as the node that is replaced.

Parameter mutation In addition to subtree mutation, we define a second form of mutation not found in standard genetic programming: parameter mutation. For parameter mutation we pass through each node and, with a predefined probability, we mutate each (numerical) parameter. Since in the initial tree generation parameters are chosen in a certain (relatively small) range, it might be possible that better values lie outside that range, therefore this step allows us to go outside these ranges. Given a parameter value N, a mutation rate P_{pm}, a mutation speed S_{pm} and two random values $0 < r_1, r_2 < 1$, the new parameter value N' is calculated as follows:

$$N' = \begin{cases} N, & \text{if } r_1 > P_{pm} \\ \lceil ((1 - S_{pm}) + 2S_{pm}r_2)N \rceil, & \text{else if } r_2 > 0.5 \\ \lfloor ((1 - S_{pm}) + 2S_{pm}r_2)N \rfloor, & \text{else} \end{cases}$$

3.5.4. Improving the performance of program evaluation

Genetic programming depends on being able to relatively quickly evaluate an entire generation of individuals. These generations often consist of 100's or even 1000's of individuals. Thus, to effectively use genetic programming as a search strategy, it is vital that calculating the fitness score of any program is as efficient as possible, both in terms of CPU and memory. Reducing unnecessary processing makes the time to evaluate a single program shorter, and reducing memory means that it is much cheaper and easier to evaluate many programs in parallel.

Reducing CPU usage Genetic programming is generally quite CPU intensive. One of the main techniques often applied in genetic programming to speed up evaluation is subtree caching [27, 43]. The basic idea is that there is some lookup table in which the output of subtrees is stored. Before calculating a subtree, the algorithm checks the lookup table if the output is already there and just returns that if it is. Otherwise, it calculates the output and stores it in the lookup table. Since the building blocks (the subtrees) are often similar over generations, this approach can speed up calculations significantly at the cost of memory or disk space.

Since the output of subtrees in our programs are generally large matrices, storing the output of each subtree is infeasible and does not necessarily increase the speed for every function. Reading the output from disk takes $O(nm)$ for a matrix with size n by m, and the sumMatrix function also takes $O(nm)$. However, the pearsonSimilarity function is quite slow and takes $O(n^2m)$. In this case, reading from disk is a significant speed improvement. In practice, we only cache the output of subtrees where the root node has a run time larger than $O(nm)$.

For the lookup table, we use a hashing algorithm H to check for subtree equivalence, which takes internal function parameters into account, as well as the dataset used, interaction requested, and the interleaved sampling parameters (seed and size). We serialize the output to disk since keeping it in memory is infeasible. Due to our hashing algorithm, our caching approach is also safe to use over multiple runs. Additionally, to reduce the required disk space, the hash algorithm uses some optimization techniques inspired by Wong [55] to make different subtrees return an equivalent hash if their outputs are the same, for example:

$$H(\text{transpose}(\text{transpose}(A))) = H(A)$$
$$H(\text{sumMatrix}(A, B)) = H(\text{sumMatrix}(B, A))$$

Additionally, we also cache the fitness scores of fully evaluated programs. Since these are only a few bytes, we can cache them all. Especially with elitism enabled, which keeps individuals in the population without changes, this can save a significant amount of unnecessary evaluations and thus increase the performance.

Reducing memory Another result of working with large matrices is that during the evaluation of a tree, a number of these matrices need to be stored in memory. To reduce the memory usage and provide an upper bound on the maximum memory of a program evaluation, we prioritize evaluating functions over terminals in the input. This gives an upper bound on the memory usage for any program in our language:

$$O((a-1)dn^2)$$

Here, d is the maximum depth, a is the largest arity of any function in our language, and n is the size of the largest set of entities in D.

3.6. Summary

In this chapter, we have described every step necessary to automatically generate well-performing recommender systems given any dataset that can be modeled in our generic data model. First, we defined a way to transform any dataset to our generic model based on entities and interactions. Then we proposed a programming language based on the underlying dataset and described its functions and terminals. Additionally, to verify whether or not a program in the language could be successfully used as a recommender system, we described a way to automatically evaluate the fitness of a recommender system given only a dataset and a requested interaction type. Finally, we described a way to search the complete space of available programs in an efficient way using genetic programming. The next step is experimentally verifying whether or not the assumptions made in this chapter are correct and that this is indeed a viable approach to automatically generate accurate hybrid recommender systems. The next chapter elaborates on the design and implementation of these experiments.

4

Experiment design

Now that we have an overview of how RACE:GP works, it is time to experimentally verify that the approach works. This chapter starts by elaborating on the goals of our experiment and their corresponding success metrics. Next, it describes the parameters of our algorithm that can be adjusted during experiments. Finally, we describe the datasets used for the experiments and how they are modeled in our previously defined generic model.

4.1. Experiment goals and success metrics

The main idea behind our approach is that it is flexible and thus usable for every situation in which recommender systems could be used to determine the relevancy of items to a user given a certain interaction. Thus, the main goal of our experiments is first to prove that our algorithm can be used to find non-trivial hybrid recommender systems that outperform certain baseline recommender systems. The next step is to verify that adjusting certain parameters before the search produces recommender systems that perform well in different circumstances.

4.1.1. Main goal

Before we can verify our approach's flexibility, we need to compare our approach with some default baselines. The baselines we use are popularity, user-based CF, and item-based CF. For our approach to work, it needs to consistently find recommender systems that outperform all three baselines. Additionally, it should find these recommender systems using fewer evaluations than a random search on the solution space would require on average.

4.1.2. Flexibility

After verifying that our approach can consistently find non-trivial recommender systems using a default dataset, the next step is verifying that our algorithm provides the flexibility to find different programs for different situations. In the experiment, we will look at three different situations:

- Different situations in which the actual recommending takes place, e.g.
  - A system shows only a single item to a user
  - It is vital that all relevant items are selected
  - The order of relevant items is important
- Different situations regarding the available data, e.g.
  - A cold-start situations with a limited amount of available interactions
  - An dense dataset with many interactions
- Recommending different types of interactions, e.g.
  - Recommending an item to view

– Recommending an item to buy

Within our experiment, we will simulate these situations by adjusting parameters or modifying the datasets. To verify whether or not these goals are achieved, we first compare the found recommender systems with the baselines for each situation and then compare the produced recommender systems for different situations. Ideally, it should produce different, specialized recommender systems for each situation.

4.2. Experiment hyperparameters

Now that we have defined the goal of the experiments, this section elaborates on which parameters of the algorithm can be adjusted to achieve these goals. We broadly categorize these parameters into three categories: search strategy hyperparameters, solution space hyperparameters, and recommender system hyperparameters. This section will elaborate on the parameters that can be adjusted for each category.

4.2.1. Search strategy hyperparameters

The search strategy category contains all hyperparameters that influence how the potential search space is explored. Since we are using genetic programming to explore the search space, the hyperparameters in this category are relatively standard in genetic programming.

Crossover and mutation rates Since genetic programming depends on crossover and mutation to 'evolve' better programs each generation, the probabilities with which these take place have a big impact on the exploration of the search space. In our algorithm, 4 parameters can be adjusted: P_{sm}, or the probability of subtree mutation in an offspring program, P_{co}, or the probability that crossover takes place between two selected programs for reproduction, P_{pm}, or the probability that a parameter inside a function or terminal mutates, and S_{pm}, or the speed with which parameter mutations can change values. Each parameter can take a value between 0 and 1, 0 meaning it is skipped completely, and 1 meaning it happens every time. Finally, we have the elitism ratio R_e, which determines the amount of best-performing individuals of a generation to add to the next generation without applying crossover or mutation.

Population size The next step is choosing a population size. The population size has a strong impact on the reproducibility of genetic programming approaches. Since genetic programming algorithms depend on the initially randomly generated population to be able to consistently explore the solution space, there needs to be enough diversity of potential subtrees (sometimes called building blocks) in the initial population [48]. Thus, choosing a population size large enough makes sure that running the algorithm multiple times produces similar results.

Tournament size The tournament size impacts the selection pressure. Setting the tournament size too large leads to selecting a small subset of each generation's best individuals for reproduction. In turn, the algorithm loses diversity and converges too fast. Conversely, a tournament size too small reduces the speed with which the average fitness of the population increases. While almost all research in genetic programming uses a tournament size of 2, recent research indicates that larger tournaments perform better in some situations [32]. Therefore we add tournament size as a hyperparameter as well.

(Interleaved) sampling Another hyperparameter related to the learning process is sampling. Sampling training and test data within recommender systems and genetic programming is usually done for two reasons: increasing evaluation speed and reducing overfitting. Our approach is based on the idea of interleaved sampling [14]. The idea is that the entire dataset is used for the initial generation, and from there on, every other generation uses a sample to create the training and test data to be used by the fitness function. Every individual in a generation is always evaluated on the same sample, but over multiple generations, different samples exist. In our algorithm, this value is a value between 0 and 1, where 1 means that the entire dataset is used in every generation. Any value below 1 is the proportion of the dataset used for the generations where the data is sampled.

4.2.2. Solution space hyperparameters

Next, we look at the parameters that impact the potential solution space. These parameters must be chosen so that the hypothetical 'best' recommender systems are part of this solution space. However, there is a trade-off because the larger this solution space is, the larger the population size needs to be to have enough diversity to consistently find good results. It is thus vital to select these as strictly as possible, but not so strict that it potentially excludes good solutions.

Available functions and terminals The first solution space parameter is not exactly a numerical parameter, but it is the set of functions and terminals the algorithm can choose from. If a function or terminal (almost) never appears in high-performing programs, removing it from the available set reduces the solution space significantly.

Maximum program depth The maximum program depth impacts the maximum nested functions a program created by the algorithm can have. In our implementation, the crossover and subtree mutation operators are also limited by this depth. Thus it is important that programs within the selected depth can capture the complexity of the dataset. We define two parameters, initial program depth, which is used during the generation of the initial population, and maximum program depth, which is used during crossover and mutation.

4.2.3. Recommender system hyperparameters

Finally, we have hyperparameters that influence the properties of the resulting recommender systems. These can be adjusted based on what is expected from the produced recommender system and are entirely dependent on the dataset and the recommendation problem.

Requested interaction The first parameter is the interaction one wants to recommend. The requested interaction is used as input for the fitness function and therefore makes sure that the algorithm evolves recommender systems that recommend the chosen interaction.

Evaluation metric The fitness function uses the selected evaluation metric to determine an individual's fitness score. The evaluation metric can be selected based on the use case. For example, if a system only shows a single item to a user, it makes sense to evaluate a recommender system based on the relevancy of the most relevant item according to the system. Alternatively, if a system needs to return all 10 relevant items for each user, it makes sense to put a recall-based evaluation metric in there. In this thesis, we use the mean reciprocal rate of the first 10 items (MRR@10), the precision score of the first recommendation (P@1), and the recall given the first 10 items (Recall@10).

Dataset preparation The last 'parameter' is not a hyperparameter in the strictest sense, but different methods of preprocessing the training data can influence the resulting recommender system. For example, it is common practice within recommender system research to filter users and items with a small number of interactions to increase the performance of collaborative filtering techniques. However, the opposite is also possible: if one filters the training data to simulate a cold-start situation, the algorithm will produce recommender systems that perform well in cold-start situations.

4.3. Datasets

We use three datasets often used in recommender system literature for our experiments: Movielens, Sobazaar, and Filmtrust. This section will elaborate on each dataset, why we chose it for our experiments, and how it is transformed into our data model. Additionally, we show the structure of the recommender systems we use as a baseline for each dataset. The reason we chose these three datasets is that they are quite different in structure. Movielens has multiple properties related to movies, Sobazaar has 7 different interaction types between users and products, and Filmtrust contains interactions between users, the trust relation, in addition to ratings between users and movies.

In general, for each dataset, we split the data based on the requested interaction. For each user, we sort the requested interactions on the corresponding timestamp and take the first 80% of the interactions as training data and the last 20% as test data. Afterward, we remove all interactions between the user and the test entities, as stated in the previous chapter.

4.3.1. Movielens

The Movielens dataset is quite well-known and well-studied within the realm of recommender systems. However, the original dataset only contains ratings and tags and, except for the title, no other metadata. Since we want to validate the capabilities of our algorithm to produce hybrid recommender systems, having a dataset with some auxiliary data would be preferred. To achieve that, we use the version of Movielens as gathered by Sun et al. [51]. They used the Movielens dataset and added information about genre, actors, and director to each movie. Their version of the dataset consists of 943 users, 1674 movies, and 100k ratings.

The dataset mapped to our generic model can be seen in table 4.1. Since our approach is based on recommending relevant items and not on guessing an explicit value such as a rating, we model ratings as 1 if a user has rated a movie and 0 otherwise. The idea behind that is that if a user has rated a movie, we can assume the movie is relevant, regardless of whether or not the user actually liked it. This is the same approach as taken by Sun et al.

entity	#	properties	type
user	943	-	-
movie	1674	genres actors director	string[] string[] string
actor	3946	-	-
genre	26	-	-

interaction	#	from	to	values
rating	80348	user	movie	0, 1
acts	6544	actor	movie	0, 1
genre	3975	genre	movie	0, 1

Table 4.1: Movielens: table containing the entities and interactions for training, with rating as requested interaction.

The three baseline recommender systems for Movielens are found in Figure 4.1. Of the three, item-based collaborative filtering performed the strongest, with an MRR@10 of 0.5672.

```
### Popularity: ###
addVector
├── randomMatrix {"seed":0}
└── popularity
    └── interaction(rating)
MRR@10: 0.339, P@1: 0.1442, P@10: 0.1043

### User-based CF: ###
nearestNeighbour {"N":15}
├── pearsonSimilarity
│   └── interaction(rating)
└── interaction(rating)
MRR@10: 0.5316, P@1: 0.2078, P@10: 0.1604

### Item-based CF: ###
nearestNeighbour(inverted) {"N":15}
├── pearsonSimilarity
│   └── transpose
│       └── interaction(rating)
└── interaction(rating)
MRR@10: 0.5672, P@1: 0.2344, P@10: 0.167
```

Figure 4.1: Baselines for Movielens based on popularity, user-based CF and item-based CF

4.3.2. Sobazaar

The second dataset we use in our experiments is the Sobazaar dataset [36]. Sobazaar is a social fashion app in which users can view, like, want, and purchase articles of clothing. The dataset consists

of users, products, and 7 interaction types. For our purpose, we created three datasets with different densities from Sobazaar: sobazaar-default, sobazaar-dense, and sobazaar-sparse. Their properties can be seen in Table 4.2. sobazaar-default is found by removing all users and products with less than 5 buy_clicked interactions. For sobazaar-dense, we took the 1000 users and products with the most interactions, and for sobazaar-sparse, we filtered all users with zero requested interactions (buy_clicked in this experiment), and then took the 1000 users with the least interactions and the 1000 products with the most interactions.

entity	#	# sparse	# dense	property	type
user	784	916	1000	-	-
product	811	888	1000	-	-

interaction	#	# sparse	# dense	from	to	values
prod_clicked	172	8	40	user	product	0, 1
prod_detail_clicked	17207	2603	42735	user	product	0, 1
prod_detail_viewed	7619	785	15027	user	product	0, 1
pixel-init	932	47	502	user	product	0, 1
prod_wanted	10813	418	48229	user	product	0, 1
interact:prod_wanted	434	42	505	user	product	0, 1
buy_clicked	3279	191	1206	user	product	0, 1

Table 4.2: Sobazaar dataset: Table containing the entities and interactions used for training, with buy_clicked as requested interaction.

Our baselines, in Figure 4.2, are again based on popularity, user-based collaborative filtering, and item-based collaborative filtering. The difference with Movielens is that we use the product_detail_clicked interaction to find similar users instead of the requested interaction buy_clicked. With some trial and error, we discovered that that interaction produced the strongest baseline.

4.3.3. Filmtrust

The last dataset we use for our experiments is Filmtrust [15]. Filmtrust contains movie ratings, as well as directional trust relationships between users. Contrary to the Movielens dataset, we use numerical ratings in the training data and only use ratings of 3 and higher as 'relevant' interactions. Table 4.3 contains the details of the training data. The baselines, found in Figure 4.3, for Filmtrust are the same as for the Movielens dataset. However, the performance of the baselines is curiously enough the other way around. Popularity performs the best by a large margin, followed by user-based collaborative filtering, and item-based collaborative filtering performs the worst.

entity	#	properties	type
user	1002	-	-
movie	2071	-	-

interaction	#	from	to	values
rating	27010	user	movie	0, 1-5
trust	1100	user	user	0, 1

Table 4.3: Filmtrust: table containing the entities and interactions for training, with rating as requested interaction.

4.4. Summary

In this chapter, we defined the main goal of our experiment: producing accurate non-trivial hybrid recommender systems. Additionally, we defined the parameters that can be adjusted to achieve our goal, and finally, the datasets used in the experiment. The next chapter will go into detail on the

```
### Popularity: ###
addVector
├── randomMatrix {"seed":0}
└── popularity
    └── interaction(buy_clicked)
MRR@10: 0.0054, P@1: 0, P@10: 0.0023

### User CF: ###
nearestNeighbour {"N":15}
├── pearsonSimilarity
│   └── interaction(product_detail_clicked)
└── interaction(buy_clicked)
MRR@10: 0.001, P@1: 0, P@10: 0.0005

### Item CF: ###
nearestNeighbour(inverted) {"N":15}
├── pearsonSimilarity
│   └── transpose
│       └── interaction(product_detail_clicked)
└── interaction(buy_clicked)
MRR@10: 0.0023, P@1: 0, P@10: 0.0013
```

Figure 4.2: Baselines for Sobazaar based on popularity, user-based CF and item-based CF, MRR@10 is calculated on sobazaar-default.

experiments ran and the corresponding results. Additionally, we will discuss these results in relation to the goals stated in this chapter.

```
### Popularity: ###
addVector
├── randomMatrix {"seed":0}
└── popularity
    └── interaction(rating)
MRR@10: 0.9868, P@1: 0.4162, P@10: 0.2839

### User CF: ###
nearestNeighbour {"N":15}
├── pearsonSimilarity
│   └── interaction(rating)
└── interaction(rating)
MRR@10: 0.8502, P@1: 0.3443, P@10: 0.2571

### Item CF: ###
nearestNeighbour(inverted) {"N":15}
├── pearsonSimilarity
│   └── transpose
│       └── interaction(rating)
└── interaction(rating)
MRR@10: 0.5257, P@1: 0.1776, P@10: 0.1803
```

Figure 4.3: Baselines for Filmtrust based on popularity, user-based CF and item-based CF

5

Experiment results

This chapter presents the results from our experiments. First, we show the results from running a grid search on several hyperparameters and compare that with our baselines. Following that, we present the results of our experiments to determine the impact of changing the recommender system hyperparameters on the produced recommender systems. Additionally, we analyse some of the produced recommender systems.

5.1. Hyperparameter tuning and baseline results

Before running more in-depth experiments, we have to find a tuple of hyperparameters that works as best as possible for our genetic programming implementation. This is done by performing a grid search over a range of sensible possible values for each hyperparameter. We select one dataset to run this grid search on, so we can compare the performance of individual runs with each other. This search is performed on the Movielens dataset. Although some hyperparameters might have different optimal values on different datasets, such as maximum depth, population size and maximum generation, the hyperparameters included in the grid search only affect the algorithm's ability to effectively search the solution space, while simultaneously making sure that the average and maximum fitness increases each generation. Thus, we assume that these parameters translate well to different datasets. Following that, we will go over the best-performing recommender systems produced by the algorithm during the grid search and compare them with the defined baselines.

The hyperparameters have a strong influence on the ability of our genetic programming algorithm to consistently search and evolve in the right direction, therefore it is vital to find a combination that performs well. In Table 5.1 we list a couple of sensible potential values for each hyperparameter. We performed a grid search on every combination possible, with the only requirement that the crossover probability P_{co} and subtree mutation probability P_{sm} satisfies $P_{co} + P_{sm} \geq 1$, since having at least one of them is a requirement for genetic programming to function at all.

5.1.1. Hyperparameter tuning results

There are 60 possible combinations of hyperparameters based on the possible values for each parameter we selected, and for each combination we used our algorithm to produce a recommender system. The complete results for each of the 60 runs of the grid search can be found in the appendix in Table A.1. To compare the performance of different combinations of hyperparameters we look at three different values for each run:

- **MRR@10**: The score of the best performing recommender system found during the run. Higher is better.

- **ΔMean**: The average increase of the mean performance for each generation during the entire run. The higher this value, the better the upwards pressure is for the algorithm. The expected value of ΔMean using a random search is 0.

Parameter	Values
Training fitness function	MRR@10
Initial depth	5
Max depth	8
Interleaved sampling	1.0
Max generation	20
Population size	100
Tournament size	2
P_{co} crossover probability*	1, 0, 0.1, 0.9, 1
P_{sm} subtree mutation probability*	1, 1, 0.9, 0.1, 0
P_{pm} parameter mutation probability	0.1, 0.5, 0.9
S_{pm} parameter mutation speed	0.1, 0.5
R_e elitism ratio	0, 0.05

Table 5.1: Potential values for hyperparameters used for the grid search, resulting in 60 unique combinations of hyperparameters.
* The possible values for crossover and subtree mutation probability in the table are linked.

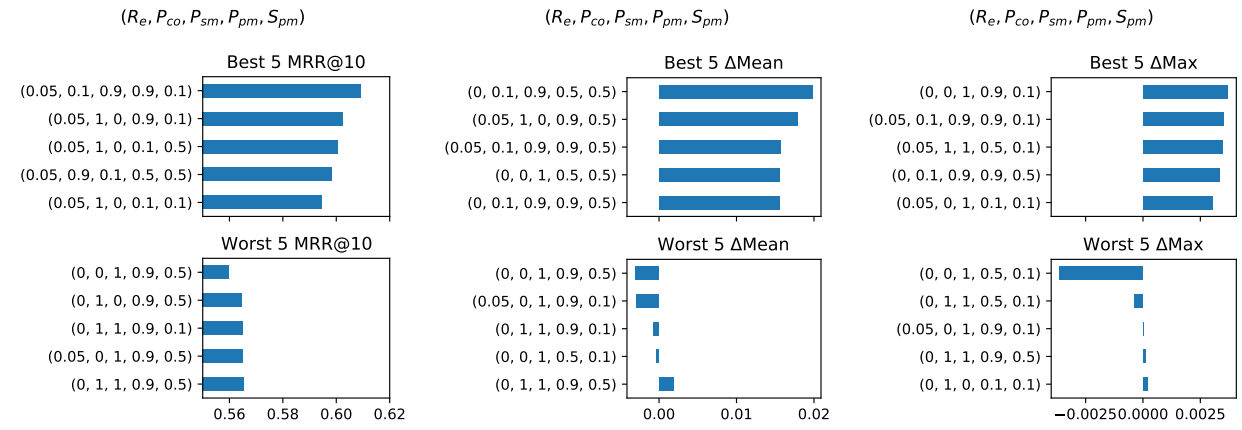
- **ΔMax**: The average increase of the max performance for each generation during the entire run. The expected value of ΔMax using a random search is 0. A higher value means the algorithm can consistently find improvements over generations.

The goal of the grid search is to find a combination of hyperparameters that leads to the best search performance and results. In the next paragraphs, we will discuss the best and worst combinations found in our experiments, and the impact individual (combinations of) parameters have on the performance. Finally, we extrapolate the tuple of hyperparameters that we use in the rest of the experiments based on the previous findings.

Best and worst performing The 5 best- and worst-performing combinations of hyperparameters can be seen in Figure 5.1a (MRR@10), 5.1b (ΔMean), and 5.1c (ΔMax). Out of the 60 runs performed in the grid search, 53 ($\approx 88\%$) produced a recommender system that performed better than all three of our chosen baselines. The ones performing worse are all within 1% of the MRR@10 of the strongest baseline. Additionally, 56 ($\approx 93\%$) of the runs has a positive ΔMean, and 57 ($= 95\%$) has a positive ΔMax. Thus, most combinations of hyperparameters chosen for the grid-search lead to hill-climbing results over multiple generations.

Elitism Looking at Figure 5.1a, the 5 best results all include elitism, and 4 out of the worst-performing 5 have no elitism. This is confirmed if we look at the average values for runs with and without elitism in Figure 5.2a. Regardless of the other parameters, runs with elitism score on average 1.7% better than runs without elitism on MRR@10, and 59% better with regards to ΔMax. Interestingly, contrary to what intuition would say, ΔMean is not impacted by whether elitism is enabled or not. Ultimately, setting $R_e = 0.05$ is strictly better than $R_e = 0$.

Crossover and Subtree mutation Crossover and subtree mutation have the same purpose: creating variation over different generations. For this reason, we linked the potential values in the grid search together, and look at their combined impact on the resulting recommender systems and evolution process. Figure 5.2b shows the average results for each combination of P_{co}, P_{sm}. Looking at MRR@10, using only crossover produces the best results on average, closely followed by (0.9, 0.1) and (0.1, 0.9) for (P_{co}, P_{sm}). These three tuples also produce the best average increase over generations, suggesting that they are suitable for our approach. Although not strictly the best tuple found in our grid search, we select $P_{co} = 0.9, P_{sm} = 0.1$ for the rest of our experiments. The reason for this is that without subtree



(a) MRR@10 (b) ΔMean (c) ΔMax

Figure 5.1: Grid search, best and worst results for MRR@10, ΔMean and ΔMax.

mutation the first randomly generated population can limit the search space if certain potential subtrees are not generated, therefore this is chosen instead of using just crossover.

Parameter mutation In Figure 5.2c the average results are shown for each combination of parameter mutation chance P_{pm}, and parameter mutation speed S_{pm}. $P_{pm} = 0.9, S_{pm} = 0.1$ outperforms every other combination with regards to MRR@10 and ΔMax. This suggests that slowly but consistently searching relatively close around parameters of systems that are performing well produces the best recommender systems. Interestingly, a smaller chance of parameter mutation increases the upwards selection pressure of the entire generation, since $P_{pm} = 0.1$ gives a good result for ΔMean. However, the explorative nature of higher parameter mutation rates and speeds leads to better-performing systems at the upper end.

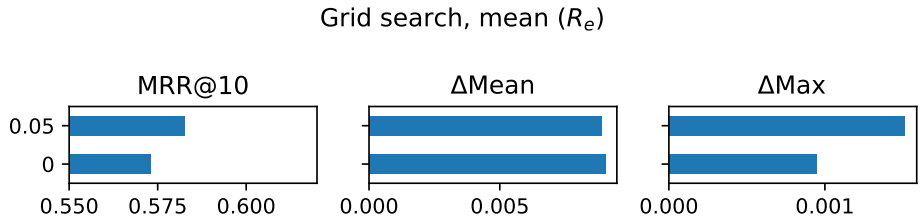
Hyperparameters for next experiments Based on the results discussed in the previous paragraphs, we select the hyperparameters for the rest of our experiments. The complete list can be found in Table 5.2. Additionally, we increase the max generation from 20 to 50, the population size from 100 to 200, and the tournament size from 2 to 4. These changes are made to improve the consistency of the results of the algorithm.

5.1.2. Example non-trivial hybrid recommender system

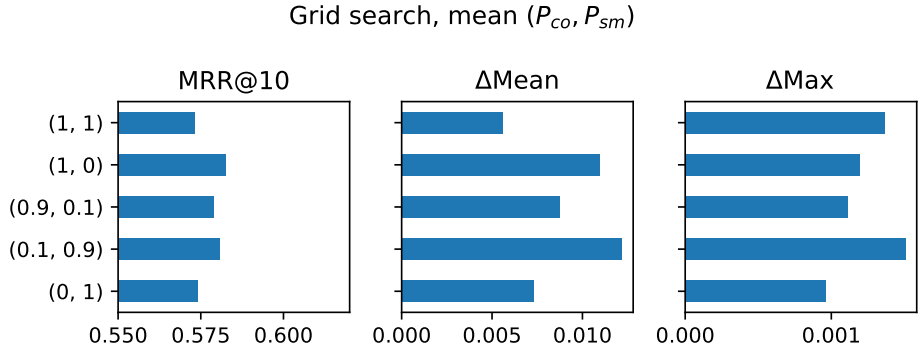
The grid search found many recommender systems that outperform our baseline, many of which are indeed non-trivial hybrid recommender systems. The following program, *grid-search-1* is one of the smaller programs found, with a $\sim 4.4\%$ improvement on the item-based CF baseline.

```
### grid-search-1 ###
transpose (6)
└── nearestNeighbour {"N":30} (5)
    ├── pearsonSimilarity (4)
    │   └── transpose
    │       └── interaction(rating)
    └── nearestNeighbour {"N":12} (3)
        ├── compareArray (1)
        │   ├── property(movie.actors)
        │   └── property(movie.actors)
        └── transpose (2)
            └── interaction(rating)
MRR@10: 0.592, P@1: 0.2577, P@10: 0.171
```

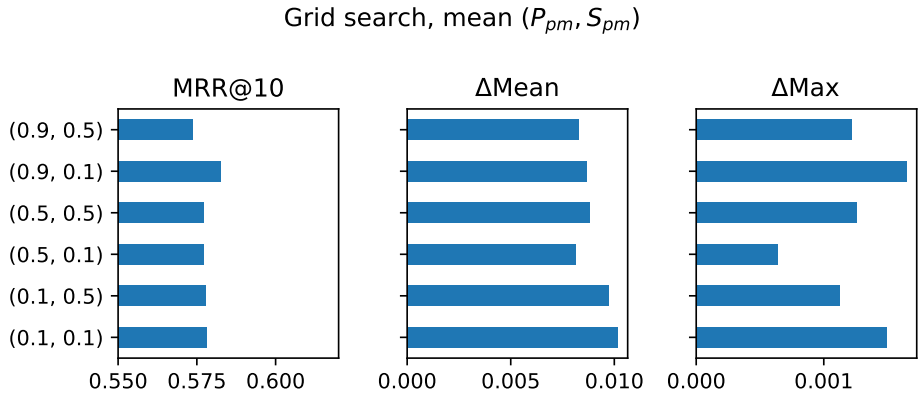
This program is a clear example of a non-trivial hybrid recommender system, and since it is quite small it lends itself perfectly to showcase the individual parts.



(a) Grid search results for elitism



(b) Grid search results for crossover and subtree mutation



(c) Grid search results for parameter mutation

Figure 5.2: Average results for parameter combinations of the grid search

- (1) First, it uses the actors that play in movies to calculate the similarity between movies.
- (2) Then, it takes the ratings each movie has received by each user.
- (3) Next, it takes the averages of these ratings and applies them to the users that rated some of the 12 movies with similar actors.
- (4) Following, it calculates the similarity between movies based on the ratings given by users.
- (5) Finally, it averages the clustered ratings from (3) for each movie based on the 30 most similar movies from (4).
- (6) The final results are transposed to produce a rating (or likeliness of 'liking') of every movie for each user.

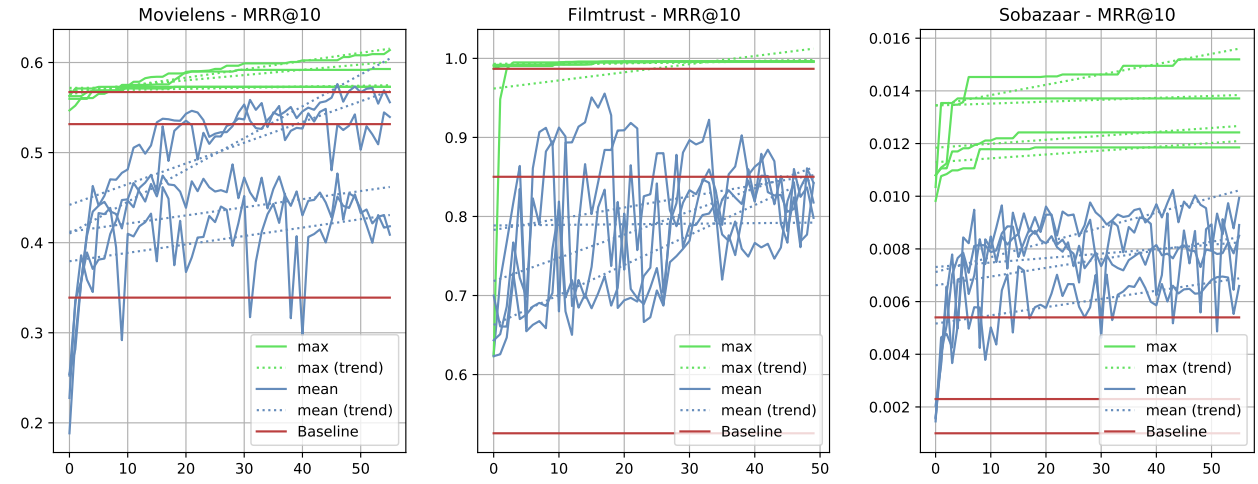
It is clear that this system combines collaborative filtering with content-based filtering in a non-trivial way: it uses content (1) as input for collaborative filtering (3), of which the output is then used as part of a different collaborative filtering strategy (5).

Parameter	Values
Training fitness function	MRR@10
Initial depth	5
Max depth	8
Interleaved sampling	1.0
Max generation	50
Population size	200
Tournament size	4
P_{co} crossover probability	0.9
P_{sm} subtree mutation probability	0.1
P_{pm} parameter mutation probability	0.9
S_{pm} parameter mutation speed	0.1
R_e elitism ratio	0.05

Table 5.2: Hyperparameters chosen for our main experiments based on the grid search

5.2. Main experiments

Since the main goal of our approach is that it is applicable to any dataset, we run RACE:GP on each of the datasets described in Section 4.3 using the hyperparameters found in the grid search. Since one of the requirements is that our algorithm is able to *consistently* outperform our baseline, we repeat the experiment so that it produces 4 recommender systems per dataset. That way, we can compare the results between runs, and verify the consistency of our approach. Figure 5.3 shows the average and maximum MRR@10 scores for each generation during the search per dataset. In the next subsections, we discuss the results on each dataset.



(a) Movielens dataset (b) Filmtrust dataset (c) Sobazaar dataset

Figure 5.3: Average and maximum fitness over generations for four runs on each of the three datasets.

5.2.1. Movielens

Although we have run many experiments on the Movielens dataset during the grid search, in this section we review the results of 4 additional runs using the selected hyperparameters. In Figure 5.3a the average and maximum MRR@10 for each of the runs are plotted per generation. Each run produced

a system that outperforms the baseline within 10 generations. The best performing system produced by each run can be seen in Appendix A.2.1.

Interestingly, there is some variance between the runs. In two of the runs, the average fitness per generation seems to get stuck between 0.4 and 0.5 after 20 generations, and in the other two, it gets stuck between 0.5 and 0.6. The maximum fitness for the latter two runs is also significantly better than the former.

The recommender with the best score can be found in Appendix A.2.1. With an MRR@10 of 0.6138, it is an improvement of 8.2% on the item-based baseline, which is significant. It combines multiple clustering approaches, using both actors and ratings to create different 'similarity' measures between movies and users, and combines them all in different ways to produce the final recommendations.

5.2.2. Filmtrust

The Filmtrust dataset only has two types of interactions: ratings and trust, and no other properties for users or movies. If we look at the results in Figure 5.3b, it seems that the available functions in our language are unable to properly make use of that trust relationship. It finds the 'best' recommender system within 4 generations, on which it does not improve in later generations. Each run produced a recommender system with approximately the same score, which is very close to the MRR@10 score from the popularity baseline. Looking at the produced recommender systems in Appendix A.2.3, 3 of the 4 runs produced a system that, after stripping the functions that have no impact on what is recommended, are slight variations of popularity. The only exception being *filmtrust-3*, which has two subtrees that make use of the trust interaction to recommend movies. This does not, however, influence the resulting MRR@10 score by any significant margin.

These results suggest that our language cannot describe recommender systems that perform well on this particular dataset. In the discussion, we will elaborate on possible reasons for this, and potential adjustments that would solve this limitation.

5.2.3. Sobazaar

The final dataset on which we ran the main experiment is the Sobazaar dataset, which includes 6 different interaction types between users and products. The results in Figure 5.3c indicate that RACE:GP is well suited to datasets with this structure: it relatively consistently finds improvements during the first 20 generations of each run, and in one of them, it also improves multiple times in later generations. Our baseline, trivial item-based collaborative filtering, results in an MRR@10 of 0.0023, while naive popularity is the strongest baseline, with an MRR@10 of 0.0054. The best found recommender system, *sobazaar-3*, with an MRR@10 of 0.0158, improves on item-based CF with 586%, and on popularity with 192%. *sobazaar-3*, found in Figure 5.4, is a great example of the success of our method on datasets like this.

First, it sums the similarity between users based on three different interactions (1), then it uses that to increase the score of items that these similar users have bought (2). Next, it uses similar users based on purchases (3) to increase the score of the clustered items that those similar users would likely buy (4). Finally, it adds these personalized scores to items that are popular based on the product_wanted interaction (5) to produce the final recommendations.

5.3. Changing recommender system hyperparameters

Now that we verified that our approach produces non-trivial hybrid recommender systems, we will explore the adaptability of our approach. In this section we review the results of three different experiments to validate the potential of our approach to be applied to different recommendation situations without needing additional configuration or manual testing. This exploration is done in three dimensions. First, we use scores from different evaluation functions in our selection process on the Movielens dataset. Secondly, we request different interactions to recommend on the Sobazaar dataset. Third, we test the impact of selecting datasets with different densities on the recommender systems produced by our algorithm on the Sobazaar dataset.

5.3.1. Using different evaluation functions

Different evaluation functions represent different recommendation strategies. For example, Precision@1 models a scenario where an application can only recommend a single item, and that item

```
### sobazaar-3 ###
addVector (5)
├── nearestNeighbour {"N":8} (4)
│   ├── pearsonSimilarity (3)
│   │   └── addVector
│   │       ├── interaction(purchase:buy_clicked)
│   │       └── popularity
│   │           └── interaction(product_detail_clicked)
│   └── nearestNeighbour {"N":9} (2)
│       ├── scaleMatrix {"scale":4}
│       │   └── sumMatrix (1)
│       │       ├── scaleMatrix {"scale":1}
│       │       │   └── pearsonSimilarity
│       │       │       └── interaction(purchase:buy_clicked)
│       │       └── sumMatrix
│       │           ├── pearsonSimilarity
│       │           │   └── interaction(content:interact:product_detail_viewed)
│       │           └── pearsonSimilarity
│       │               └── interaction(content:interact:product_wanted)
│       └── interaction(purchase:buy_clicked)
└── popularity
    └── interaction(product_wanted)
MRR@10=0.0158
```

Figure 5.4: Recommender system *sobazaar-3*.

must be relevant, Recall@10 suggests a situation where all relevant items must be recommended. For this experiment, we ran three runs, where the score used by tournament selection was respectively MRR@10, Precision@1, and Recall@10. The rest of the parameters were the same as in the main experiments from the previous section. In Figure 5.5, the scores for each recommender system for every evaluation metric can be seen.

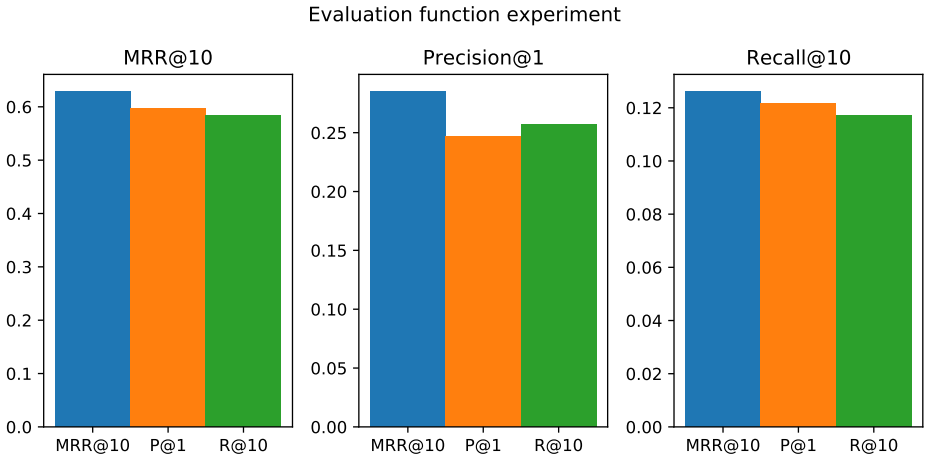


Figure 5.5: MRR@10, Precision@1, and Recall@10 scores for the recommender systems produced when MRR@10, Precision@1, and Recall@10 were used as evaluation functions during the selection process.

The results give no clear indication that using a specific evaluation function during the training produces a recommender system that performs especially well on that evaluation metric. Except for Precision@1, the results indicate that the system that performs the best on MRR@10 also scores the best on Recall@10 and Precision@1. This suggests a correlation between those evaluation metrics. This might be related to this specific dataset, or it might be due to our approach regarding splitting the data into training and test data.

5.3.2. Requesting different interactions

The second goal of RACE:GP is to make it easy to create recommender systems for different situations within a single dataset. In this experiment, we used 4 different interaction types of the Sobazaar dataset to create different train and test sets used during genetic programming, resulting in four recommender systems: *product_detail_viewed*, *product_detail_clicked*, *product_wanted* and *buy_clicked*. Then, for each produced recommender system, we calculated the MRR@10 on the three data splits on which it was not trained. Figure 5.6 shows the results. The resulting recommender systems can be seen in Appendix A.3.2. Immediately it is clear that for each data split, the recommender system that was produced by running RACE:GP specifically on that split performs best on that split. These results suggest that our approach can automatically find recommender systems that perform well on recommending a specific interaction. Additionally, it shows that in the Sobazaar dataset, the different interactions are not directly correlated. If that was the case, then the relative scores of each recommender system on a single data split should be similar for each data split.

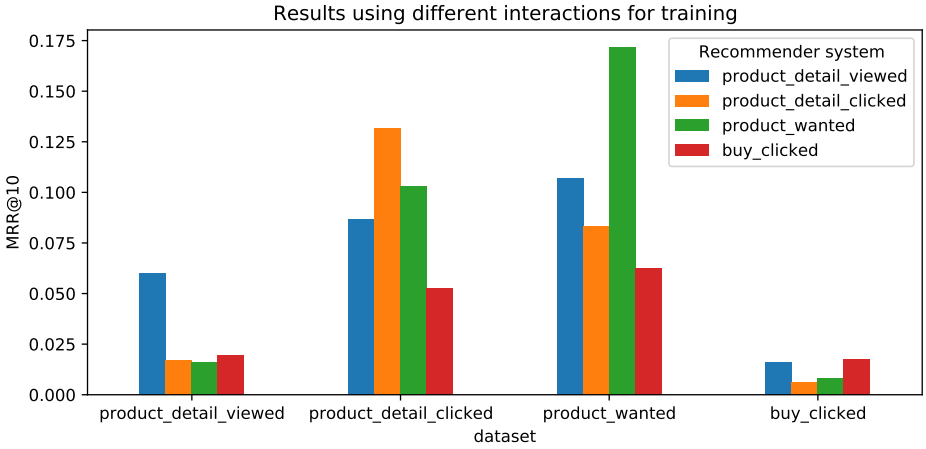


Figure 5.6: The MRR@10 score for each interaction for the best recommender system found in each run.

5.3.3. Impact of data density

The final adaptability experiment of our approach is based on dataset density. We use the two variations of the Sobazaar dataset described in Section 4.3: sobazaar-dense and sobazaar-sparse. On both datasets, we run the experiment twice to reduce variance. This results in 4 recommender systems: *dense-0*, *dense-1*, *sparse-0* and *sparse-1*. In Figure 5.7 the MRR@10 scores of each recommender system on both datasets. It is immediately clear that *dense-1* and *dense-2* perform significantly better on the dense dataset than their sparsely trained counterparts, which is to be expected, however, one would expect that a recommender system that performs well on a dense dataset would also perform well on a sparse dataset, which is in contrast to the results of this experiment: *dense-1* is the best performing system on sobazaar-dense, but the worst performing system on sobazaar-sparse. Additionally, the converse is certainly not true, *sparse-0* and *sparse-1* perform quite well on the sparse dataset, but on the dense dataset, they score significantly worse than their dense counterparts.

These results suggest that RACE:GP is indeed able to create recommender systems that perform especially well on datasets with certain density properties. This is especially applicable in situations where the density of a dataset changes over time, which is generally the case for most platforms that use some form of recommendation.

5.4. Summary

In this chapter, we gave an overview of the experiments we ran and their results. We performed a grid search over several sensible hyperparameters to find the best performing combination. Then we showed the general applicability of RACE:GP by using it to create 4 recommender systems each for the Movielens, Filmtrust, and Sobazaar datasets. All produced recommender systems were at least as accurate as our strongest baselines. For Movielens and Sobazaar, it produced non-trivial hybrid

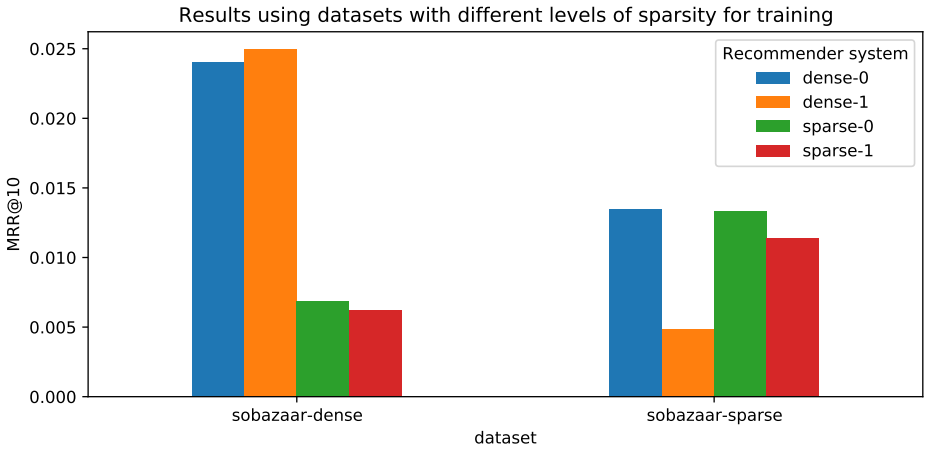


Figure 5.7: The MRR@10 scores on both the sparse and dense version of the Sobazaar datasets for the systems produced using either one of them as the training set.

recommender systems that improved on the strongest baselines by 8.2% and 192% respectively. Finally, we showed that our approach can automatically create recommender systems that are optimized for different scenarios, such as recommender systems for specific interactions in a dataset or recommender systems for the same dataset with different interaction densities. In the next chapter, we will discuss the impact of these findings, the limitations of the experiment, and recommendations for further research.

6

Discussion

In the previous chapter, we have shown the results of running RACE:GP on different recommendation scenarios. In this chapter, we will discuss how we can interpret these results and what their implications are, as well as reviewing the unexpected results. Next, we elaborate on the limitations of our experiments and potential ways to solve these limitations. Finally, we will extrapolate some recommendations for future research directions to improve our approach.

6.1. Discussion of results

As stated in the introduction, this thesis aims to show that our approach can find non-trivial hybrid recommender systems that outperform the selected baselines on any dataset. The previous chapter shows that the recommender systems found by RACE:GP outperform the Movielens baselines by 8.2% and the Sobazaar baselines by 192%. Additionally, it matches the accuracy of the Filmtrust baselines. Furthermore, we have demonstrated that the produced recommender systems are non-trivial hybrid recommender systems that combine different recommendation techniques. We have demonstrated the adaptability and general applicability in the experiments where we produced recommender systems on the Sobazaar dataset with different densities and with different requested interactions. These results clearly indicate that our approach confirms the hypotheses stated in the introduction. This section first discusses the results that are in line with our hypotheses, followed by unexpected results.

6.1.1. General interpretations

Finding non-trivial hybrid recommender systems An important requirement for our approach is that the produced recommender systems incorporate *non-trivial* combinations of different recommendation techniques. Looking at the recommender systems produced on the Movielens and Sobazaar dataset, we find non-trivial hybrid recommenders using multiple techniques often used in hybrid recommender system research. If we look at the categories regarding combining techniques as defined by Çano and Morisio [9], we find the following examples in our produced recommender systems:

- **Weighted** *evaluation-mrr@10* combines item-based collaborative filtering based on clusters of movies with similar actors with user-based collaborative filtering.
- **Feature combination** *sobazaar-2* combines multiple features to calculate the similarity between products, which is then used as input for item-based collaborative filtering.
- **Cascade** *sobazaar-3* uses collaborative filtering to first produce a list of recommendations based on the combined similarity on different interactions. Then it refines that list using users that specifically purchased similar items.
- **Feature augmentation** *movielens-3* uses the expected ratings on clustered movies by actors and item-based collaborative filtering as a feature for user-based collaborative filtering.

There is one technique that does not appear in any of the recommender systems: ‘switching’. The reason for this is that our language does not contain a function that can emulate this technique. A

potential function that would solve this could be an 'if-else' function that takes three inputs: Vector<A>, Matrix<A,B>, Matrix<A,B>. The output it produces would be a Matrix<A,B> where it takes row i from the first input matrix if value i in the vector is higher than some threshold, and row i of the second matrix otherwise. In combination with the popularity function to produce the input vector, this could emulate the behavior of selecting a different recommender system based on how many items a user has rated.

Capability of our programming language to describe accurate hybrid recommender systems Our experiments show that the functions and terminals we chose for our language are capable of producing accurate hybrid recommender systems for the Movielens and Sobazaar datasets. This indicates that the multiple types of interactions in the Sobazaar dataset and the properties in the Movielens dataset lend themselves especially well to the selected set of functions for our experiments. However, looking at the recommender systems produced for Filmtrust, the selected functions are unable to effectively use the 'trust' interaction between users, even though literature suggests that using the trust relationship improves the recommendation accuracy [53].

Efficiency of finding accurate recommender systems In our experiments, our language consisted of 10 functions. The number of terminals depends on the dataset: 6 for Movielens, 7 for Sobazaar, and 2 for Filmtrust. For any maximum depth d, an upper bound of the number of potential programs P_d is

$$|P_d| \leq \#\text{functions} \cdot |P_{d-1}|^2 + \#\text{terminals}$$

The actual value is smaller since not every function in our language requires two inputs, and since our language is strongly typed, not every function or terminal is a valid input for every other function. However, with a maximum depth of 5, this gives an upper bound of $\sim 10^{26}$. The realistic number is likely much smaller, but it indicates the enormous search space. Add the fact that some functions include numerical parameters, such as scaleMatrix and nearestNeighbour, making an exhaustive search impossible. Fortunately, the grid search results indicate that genetic programming is an effective search strategy. Regardless of the chosen hyperparameters, in most situations, the ability of genetic programming to find accurate recommender systems is significantly better than what a random search would produce.

Adapting to different requested interactions The recommender systems produced by running RACE:GP with different requested interactions on the Sobazaar dataset clearly show that a recommender system that accurately recommends products to buy does not necessarily accurately recommend items to view. This suggests that there is no correlation between the 'implicit feedback' provided by the different interaction types. This contradicts the hypothesis by Nguyen et al. that there is a linear relationship between 'clicks', 'wants', and 'purchases' [36]. Furthermore, this validates our decision to model 'relevance' as the probability that a certain interaction should take place between a user and an item. If recommending an item were the same as recommending an interaction, the recommender systems produced on different interactions should have performed similarly (relative to each other) on each interaction.

Adapting to different densities Results indicate that the structure of a dataset: the interaction types and properties, is not the only thing that determines the structure of an accurate hybrid recommender system. For example, the produced recommender systems trained on the dense version of the dataset, *dense-0* and *dense-1*, perform $\sim 250\%$ better on the dense dataset than the systems produced on the sparse version. The other way around, the systems trained on the sparse dataset match *dense-0* and improve on *dense-1* by $\sim 275\%$. This suggests that if the density of a dataset used for recommendation changes over time, the accuracy might be improved when updating or retraining the recommender system periodically.

In practice, the density of interactions within a dataset varies not only over time but also varies between users. Hybrid recommendation techniques such as switching can be used to adjust results based on the number of interactions a user has within the dataset. The 'if-else' function mentioned before combined with the popularity function could potentially create systems that dynamically adjust the recommendations based on their amount of interactions.

6.1.2. Unexpected results

Although, in general, our hypothesis is confirmed, there are also some unexpected results. For example, the performance on the Filmtrust dataset and the impact of changing the evaluation function on the produced recommender systems. In this section, we discuss these results and hypothesize their cause and potential solutions.

Performance on Filmtrust While the systems produced on the Movielens and Sobazaar datasets are true hybrid recommender systems, the best systems found for Filmtrust are all slight variations on *popularity* and thus trivial. This suggests that our language is unable to formulate recommender systems that perform well on that dataset. Guo et al. [16, 17] show that it is definitely possible to produce recommender systems on the Filmtrust dataset that outperform *popularity*. Although in their research they use numerical evaluation methods (MAE and RMSE), they compare their method (and several other methods) to ItemAVG, which is the numerical variant of *popularity*, and show that it performs significantly better. Their approach is based on singular value decomposition, which in turn is a form of matrix factorization. Since our language does not contain any matrix factorization functions, our language is likely unable to describe recommender systems that perform well on the Filmtrust dataset. In general: our approach is limited by the ability of the language to describe potential recommender systems. The fact that much research regarding recommender systems uses approaches based on matrix factorization suggests that adding functions to support these systems can significantly improve our approach.

Changing the evaluation function While the results confirm that our approach can find recommender systems for specific scenarios related to the requested interaction type or the density of a dataset, our results indicate that changing the evaluation function does not impact the produced recommender systems. In fact, the relative performance of the produced systems on the three evaluation functions used in our experiments, MRR@10, Precision@1, and Recall@10, is quite similar. This suggests that the correlation between these metrics is too significant for our approach to produce meaningfully distinct recommender systems. It might make sense to introduce different strategies regarding splitting the dataset into training and validation data in conjunction with different evaluation metrics. For example, for precision@1, only select a single interaction for each user for the validation set. Additionally, recent research suggests that there exist less biased evaluation metrics than precision and recall [41]. This indicates that exploring different evaluation metrics makes sense as well. Finally, the most accurate way to evaluate recommender systems is by using them in practice and tracking whether users interact with the provided recommendations or not. In a system with enough users, it might even be possible to incorporate real user feedback in the calculation of the fitness scores of individuals.

6.1.3. Implications of our findings

As far as we are aware, this thesis is the first exploration of finding an approach that completely automates the process of finding hybrid recommender systems for specific situations and datasets. Furthermore, our results suggest that our methodology is capable of consistently finding these systems.

With a small amount of development work, RACE:GP can be improved to a complete 'recommendations as a service' platform or project. The only manual work required is mapping the dataset to our model based on entities and interactions, which is quite straightforward. Anyone with a dataset and an idea of which interaction they would like to recommend can create these recommender systems, making them accessible for everyone, regardless of available resources.

Another potential application for our approach is assisting experienced recommender system designers or researchers with manually creating and optimizing highly tuned recommender systems. The recommender systems produced by RACE:GP can inspire or suggest interesting hybrid techniques to combine certain properties and interaction types given a dataset. These suggestions can then be used in manually designed systems that focus on performance or handle extremely large-scale datasets.

6.2. Experimental limitations

Although our experiments have shown that it is definitely possible to automate the process of creating recommender systems, our experiments were still subject to some limitations that, if removed or reduced, could significantly improve the results of our approach. The three main limitations are based on

the chosen functions and their implementation, the size of the datasets used in our experiments, and the performance of evaluating an individual program. In this section, we will discuss these limitations and elaborate on their impact on the validity of our conclusions.

6.2.1. Chosen functions and their implementation

The first limitation of our experiments is the functions of our language. The success of RACE:GP depends on whether or not the language is sufficiently able to express well-performing recommender systems in it. This is clearly the case in our experiment on the Filmtrust dataset. The best-found recommender systems are all variations of the 'popularity' approach, while Guo et al. are able to significantly improve on popularity [16, 17]. It must be said that in their research, they use numerical evaluation, namely mean absolute error and root mean squared error, and thus they compare those values with ItemAVG, which is similar to our popularity baseline. Nevertheless, these results suggest that it is possible to improve on these baselines. However, our language is unable to find these recommender systems. Their approach is based on singular value decomposition, and the baselines they compare against are based on matrix factorization. Within our experiments, both of these techniques are not included as functions of the language.

Related to that, in early experiments, we included a matrix product function in our language, which allowed for the creation of interesting systems as well. The idea being that it could model relations between different interactions, such as actors in movies that users rated highly. However, since calculating a matrix product has a runtime of about $O(n^3)$, it significantly increased the time it took to evaluate an individual that had one or more matrix product functions in it, and thus we decided to remove it from the pool of available functions for our experiments.

The fact that our implementation is able to find non-trivial hybrid recommender systems for the Movielens and Sobazaar dataset that perform significantly better than the baselines, even with the limited set of functions and terminals, shows that our approach has a lot of potential. More research regarding the set of functions and experimentation with different combinations of functions and terminals can ensure that our approach can produce accurate recommender systems for every dataset.

6.2.2. Size of the datasets in our experiments

The second limitation in our experiments is the size of the datasets. Most functions in our experiment implementation require one or more matrices as input and produce a matrix as output. Our implementation uses dense matrices, which require a significant amount of memory: ratings between 1000 users and 1000 movies require $1000 \cdot 1000 \cdot 64$bits $= 8$MB of memory, regardless of the number of actual ratings. And while storing a number of those matrices in memory is easily achievable on a modern system, most realistic recommendation datasets have exponentially more users and items, making the implementation used in our experiments infeasible in reality.

However, this does not mean that our approach can not be used on larger systems. There are multiple ways to mitigate this problem in practice. First, RACE:GP can be used on a smaller set of data with similar characteristics as the complete dataset to evaluate individual programs, similar to our experiments, and only using the resulting system on the complete dataset. At that point, the evaluation speed is less critical than during the genetic programming stage since intermediate values can be cached efficiently.

Another possibility is to use sparse matrices and modify functions to efficiently use sparse matrices as well. Intuitively, only the outlying values are relevant to recommending. Keeping only the outlying values for each row and column before returning a matrix in a function can significantly reduce memory usage and evaluation speed. Research is necessary to confirm this.

6.2.3. Performance of evaluating an individual program

The last limitation of our approach is related to the previous limitation: the amount of time it takes to evaluate an individual program and the impact that has on our population size. For genetic programming to consistently find the best approaches, the initial population must have enough diversity [48]. Our experiment results show quite a lot of variance in the result of multiple runs of the same experiment. This is especially visible in the main experiment, where we performed 4 runs on each dataset with a population size of 200. For Sobazaar, *Sobazaar-3* found a system with an MRR@10 of 0.0158, which is 30% better than the best system found by *Sobazaar-1*, which has an MRR@10 of 0.0121. For Movielens, the system from *Movielens-3* outperformed *Movielens-1* by 7%. These differences are

significant and suggest that the chosen population size of 200 is too small to consistently find the best performing recommender systems.

Schweim et al. [48] recommend population sizes of 1000-3000 individuals given languages with similar or fewer functions and terminals, and it is safe to assume that according to them, our language would require at least a similar population size to produce the best possible results consistently. Unfortunately, with our current implementation, using these population sizes would not be feasible within the time span for this thesis. However, there are many possibilities to reduce the time required to evaluate an individual, such as improved methods for caching or trading disk space for performance by caching the output of every subtree, or, as mentioned before, using sparse matrices instead of dense matrices.

Even with the limited population sizes used in our experiments, almost every run produced a system that performed better than our baselines, and in some cases, significant improvements. Thus, our experiments show that it would find these systems even more consistently with larger population sizes, verifying its success.

6.3. Conclusion and recommendations

In this chapter, we discussed the results of our experiments, what their implications are, and the limitations of our experiments. We conclude this chapter by providing recommendations for further research based on our findings and limitations. Since this thesis is an initial exploration into a novel approach regarding recommendation, there are many different areas where further research is necessary. Our recommendations can be broadly categorized in two ways: research to improve the capability of our language to define hybrid recommender systems and research that improves the efficiency with which our approach can find the most accurate hybrid recommender systems.

Improving our language As stated earlier in this chapter, the accuracy of the recommender systems our approach can find depends on the ability of our language to define these systems. Since this is a new area of research, the functions and terminals defined in this thesis for the language are based on intuition and reasoning on hybrid recommender systems. The effect of adding or removing functions on the performance and general applicability requires more research. Examples of new functions are the earlier discussed matrix factorization or 'if-else' functions. In this thesis, the functions are based on traditional collaborative filtering, but it might also make sense to introduce different techniques as functions based on classifiers or neural networks, for example.

Additionally, in recent literature, context is often used to improve recommendations. The context in our data model translates to information related to interactions, such as type of device, time of day, or location. Our proposed data model does not support context data, and the functions in our language are unable to deal with it. Being able to automatically include context data in our approach to automatically create recommender systems would strongly increase the general applicability.

Improving the efficiency of our algorithm In the previous section, we stated that the population sizes in our experiments were not sufficient to eliminate the variation in the resulting recommender systems. To increase the population size without increasing the runtime of our algorithm, its efficiency needs to be improved. We suggest three research directions that can potentially increase the efficiency of RACE:GP.

First, our implementation currently uses dense matrices to pass information between functions. This is because of functions such as pearsonSimilarity and nearestNeighbour, which output potentially unique values for each cell in the matrix. We hypothesize that only outlying values within a matrix or within each row or column are relevant for recommending or ranking. Thus removing all values except the outliers before returning a matrix in a function would significantly reduce the number of calculations necessary in the function that uses that result as input. Further research to confirm or deny this is necessary, as well as the impact this has on performance.

The second area where improvement is possible is the actual implementation of functions within our algorithm. For example, in the implementation used for our experiments, pearsonSimilarity is a bottleneck. We reduced the impact by introducing a caching mechanism that caches the output of every subtree with pearsonSimilarity at its root. However, with larger program sizes, this starts to become infeasible due to the disk space required. Replacing functions like that with alternatives with similar accuracy but an improved runtime could improve efficiency.

Finally, another way to increase the efficiency of our search strategy is by reducing the search space. During our evolution process, many (parts of) recommender systems are evaluated that make no sense in the context of recommendation or matrix calculations. Applying stricter rules to our language so that subtrees that can never contribute to recommendation accuracy are invalid could greatly reduce the search space, thus increasing the efficiency.

Applying the resulting recommender systems in practice As mentioned before, one of the potential applications for our approach is assisting engineers with the design of non-trivial hybrid recommender systems that can be used in practice for large-scale recommendations, since our matrix based implementation to evaluate the performance of a recommender system does not directly translate to situations with millions of users. We are confident that it is also possible to create a different interpreter of our high-level language that can be used on a very large scale. Since this was outside the scope of this thesis, we strongly recommend researching the possibilities regarding this, since that is the last step to using our approach in practical situations.

7

Conclusion

In this thesis, we have presented RACE:GP, an approach based on genetic programming to automatically create non-trivial hybrid recommender systems that can be applied to any recommendations scenario. We have shown that RACE:GP can produce recommender systems that are more accurate than standard item-based and user-based collaborative filtering recommender systems as well as popularity-based recommender systems on three different datasets often used in literature. Additionally, we have shown that our approach can produce accurate recommender systems that perform especially well on specific recommendation scenarios, such as recommending different interactions within a dataset and recommending in scenarios with different densities of the same dataset. Based on these results, we can conclude that our approach is an effective and generally applicable method of producing recommender systems without any specific knowledge of the recommendation domain or recommender systems in general. Thus, we have succeeded in making recommender systems more accessible for everyone, regardless of expertise. Our main contributions can be summarized as follows:

- We have proposed a method to automatically derive a high-level programming language from any dataset in which any valid program is a hybrid recommender system on that dataset. Additionally, we have defined an initial set of functions based on techniques often seen in hybrid recommender systems. Our results have shown that programs in the language can be accurate recommender systems, and analysis of the produced recommender systems show that our set of functions is able to represent most techniques generally found in hybrid recommender systems.
- We introduced a generic approach for defining what ‘relevant’ means in the context of recommending relevant items based on interactions. Furthermore, we have shown a method that can automatically evaluate the accuracy of a recommender system in the aforementioned language based on a requested interaction.
- We have experimentally verified that our version of genetic programming can consistently find accurate non-trivial hybrid recommender systems in the languages that outperform our chosen baselines by a significant margin. Additionally, we have shown the flexibility of our approach by applying it to different recommendation scenarios and verifying that it produces recommender systems that are especially accurate in those situations.

With this thesis, we have provided an initial exploration into the realm of automating the process of designing and creating accurate hybrid recommender systems. In addition, we have identified areas where future research might improve on our approach, such as adjusting the set of functions in the language to incorporate more recommendation techniques, improving the performance of evaluating a recommender system, or applying the recommender systems produced by RACE:GP in practice on large-scale real-life scenarios.

A

Experiment results

A.1. Grid search

R_e	P_{sm}	P_{co}	P_{pm}	S_{pm}	δmean	δmax	MRR@10	P@1	P@10	Generation
Baseline: Popularity							0.3390	0.1442	0.1043	-
Baseline: User based CF							0.5316	0.2078	0.1604	-
Baseline: Item based CF							0.5672	0.2344	0.1670	-
0	0	1	0.1	0.1	0.0151	0.0002	0.5714	0.2407	0.1679	19.0
0	0	1	0.1	0.5	0.0095	0.0015	0.5815	0.2513	0.1666	13.0
0	0	1	0.5	0.1	0.0101	0.0008	0.5769	0.2513	0.1664	8.0
0	0	1	0.5	0.5	0.009	0.0003	0.5692	0.2471	0.162	14.0
0	0	1	0.9	0.1	0.0094	0.001	0.5721	0.2577	0.1618	10.0
0	0	1	0.9	0.5	0.0121	0.0003	0.5646	0.2322	0.1663	2.0
0	0.1	0.9	0.1	0.1	0.0077	0.0005	0.5795	0.2344	0.1736	16.0
0	0.1	0.9	0.1	0.5	0.0109	0.0003	0.5687	0.245	0.1635	2.0
0	0.1	0.9	0.5	0.1	0.007	0.0008	0.5711	0.2439	0.1637	14.0
0	0.1	0.9	0.5	0.5	0.0033	0.0015	0.5674	0.2344	0.1678	10.0
0	0.1	0.9	0.9	0.1	0.0117	0.0012	0.5798	0.2481	0.1665	14.0
0	0.1	0.9	0.9	0.5	0.01	0.0005	0.576	0.246	0.1652	16.0
0	0.9	0.1	0.1	0.1	0.0113	0.0014	0.575	0.2428	0.1683	14.0
0	0.9	0.1	0.1	0.5	0.0125	0.0019	0.5805	0.2407	0.1676	18.0
0	0.9	0.1	0.5	0.1	0.0142	0.0008	0.5724	0.2503	0.1636	17.0
0	0.9	0.1	0.5	0.5	0.0198	0.0008	0.574	0.2524	0.1621	18.0
0	0.9	0.1	0.9	0.1	0.0079	0.0003	0.568	0.2322	0.1674	12.0
0	0.9	0.1	0.9	0.5	0.0155	0.0034	0.5871	0.2524	0.1738	15.0
0	1	0	0.1	0.1	0.0116	0.0006	0.5754	0.2397	0.1673	15.0
0	1	0	0.1	0.5	0.0138	0.0007	0.5714	0.246	0.165	15.0

Table A.1: Results of the grid search applied on the Movielens dataset.

R_e	P_{sm}	P_{co}	P_{pm}	S_{pm}	δmean	δmax	MRR@10	P@1	P@10	Generation
Baseline: Popularity							0.3390	0.1442	0.1043	-
Baseline: User based CF							0.5316	0.2078	0.1604	-
Baseline: Item based CF							0.5672	0.2344	0.1670	-
0	1	0	0.5	0.1	-0.0003	-0.0036	0.5711	0.2397	0.1691	18.0
0	1	0	0.5	0.5	0.0156	0.0023	0.5817	0.2513	0.166	18.0
0	1	0	0.9	0.1	0.0143	0.0037	0.5845	0.2556	0.1684	18.0
0	1	0	0.9	0.5	-0.003	0.001	0.5598	0.2259	0.1668	8.0
0	1	1	0.1	0.1	0.0092	0.0029	0.5658	0.2481	0.1625	14.0
0	1	1	0.1	0.5	0.004	0.001	0.5761	0.246	0.1671	1.0
0	1	1	0.5	0.1	0.0022	-0.0004	0.5665	0.2428	0.1617	4.0
0	1	1	0.5	0.5	0.005	0.001	0.5715	0.2439	0.1644	6.0
0	1	1	0.9	0.1	-0.0007	0.0017	0.5648	0.2397	0.1648	5.0
0	1	1	0.9	0.5	0.0019	0.0001	0.5652	0.2481	0.1628	0.0
0.05	0	1	0.1	0.1	0.0103	0.0023	0.5944	0.2577	0.1749	19.0
0.05	0	1	0.1	0.5	0.0103	0.0021	0.6005	0.2704	0.1682	12.0
0.05	0	1	0.5	0.1	0.0095	0.0009	0.5823	0.2619	0.1632	17.0
0.05	0	1	0.5	0.5	0.006	0.0017	0.5861	0.2492	0.1673	7.0
0.05	0	1	0.9	0.1	0.0123	0.0022	0.6023	0.2534	0.1745	17.0
0.05	0	1	0.9	0.5	0.0179	0.0013	0.5878	0.2577	0.1671	18.0
0.05	0.1	0.9	0.1	0.1	0.0057	0.0005	0.5688	0.2386	0.1646	18.0
0.05	0.1	0.9	0.1	0.5	0.0079	0.0008	0.5713	0.2492	0.1636	18.0
0.05	0.1	0.9	0.5	0.1	0.0147	0.0008	0.5829	0.2513	0.1689	19.0
0.05	0.1	0.9	0.5	0.5	0.0077	0.0027	0.5984	0.2428	0.1762	15.0
0.05	0.1	0.9	0.9	0.1	0.0112	0.0015	0.5944	0.2598	0.1685	19.0
0.05	0.1	0.9	0.9	0.5	0.0074	0.0024	0.5894	0.2598	0.1718	19.0
0.05	0.9	0.1	0.1	0.1	0.0098	0.0016	0.5894	0.2524	0.1698	17.0
0.05	0.9	0.1	0.1	0.5	0.0144	0.0006	0.5779	0.2418	0.1674	17.0
0.05	0.9	0.1	0.5	0.1	0.0072	0.0017	0.5874	0.2672	0.1651	16.0
0.05	0.9	0.1	0.5	0.5	0.0061	0.0005	0.5734	0.2333	0.1682	13.0
0.05	0.9	0.1	0.9	0.1	0.0119	0.0035	0.6093	0.2683	0.1735	13.0
0.05	0.9	0.1	0.9	0.5	0.0157	0.0015	0.572	0.2428	0.1643	18.0
0.05	1	0	0.1	0.1	0.0088	0.003	0.5787	0.2492	0.1685	17.0
0.05	1	0	0.1	0.5	0.0098	0.0013	0.5735	0.2397	0.1638	18.0
0.05	1	0	0.5	0.1	0.0082	0.0012	0.5768	0.2418	0.1676	12.0
0.05	1	0	0.5	0.5	0.0097	0.0009	0.5802	0.2333	0.1729	19.0
0.05	1	0	0.9	0.1	-0.0029	0.0	0.5712	0.2407	0.1676	17.0

Table A.1: Results of the grid search applied on the Movielens dataset.

R_e	P_{sm}	P_{co}	P_{pm}	S_{pm}	δmean	δmax	MRR@10	P@1	P@10	Generation
Baseline: Popularity							0.3390	0.1442	0.1043	-
Baseline: User based CF							0.5316	0.2078	0.1604	-
Baseline: Item based CF							0.5672	0.2344	0.1670	-
0.05	1	0	0.9	0.5	0.0021	0.0004	0.5651	0.2375	0.1646	15.0
0.05	1	1	0.1	0.1	0.0119	0.0019	0.5828	0.2503	0.1692	15.0
0.05	1	1	0.1	0.5	0.0043	0.0011	0.5784	0.245	0.1659	11.0
0.05	1	1	0.5	0.1	0.0085	0.0035	0.5858	0.2577	0.1662	18.0
0.05	1	1	0.5	0.5	0.006	0.0008	0.5699	0.246	0.1612	15.0
0.05	1	1	0.9	0.1	0.0113	0.0014	0.579	0.246	0.167	11.0
0.05	1	1	0.9	0.5	0.0033	0.0014	0.5706	0.2365	0.1665	9.0

Table A.1: Results of the grid search applied on the Movielens dataset.

A.2. Main experiment recommender systems

A.2.1. Movielens

```
### Movielens-0 ###
nearestNeighbour(inverted) {"N":17}
├── scaleMatrix {"scale":3}
│   └── pearsonSimilarity
│       └── scaleMatrix {"scale":2}
│           └── transpose
│               └── sumMatrix
│                   ├── interaction(rating)
│                   └── interaction(rating)
└── sumMatrix
    ├── sumMatrix
    │   ├── interaction(rating)
    │   └── interaction(rating)
    └── interaction(rating)
MRR@10=0.5745


### Movielens-1 ###
nearestNeighbour(inverted) {"N":17}
├── pearsonSimilarity
│   └── transpose
│       └── sumMatrix
│           ├── sumMatrix
│           │   ├── interaction(rating)
│           │   └── interaction(rating)
│           └── interaction(rating)
└── interaction(rating)
MRR@10=0.5732
```

```
### Movielens-2 ###
transpose
└── nearestNeighbour(inverted) {"N":34}
    ├── pearsonSimilarity
    │   └── sumMatrix
    │       ├── addVector
    │       │   ├── sumMatrix
    │       │   │   ├── interaction(rating)
    │       │   │   └── interaction(rating)
    │       │   └── popularity
    │       │       └── interaction(genre)
    │       └── sumMatrix
    │           ├── sumMatrix
    │           │   ├── interaction(rating)
    │           │   └── interaction(rating)
    │           └── interaction(rating)
    └── transpose
        └── nearestNeighbour {"N":19}
            ├── sumMatrix
            │   ├── pearsonSimilarity
            │   │   └── interaction(rating)
            │   └── pearsonSimilarity
            │       └── interaction(rating)
            └── interaction(rating)
MRR@10=0.5931
```

```
### Movielens-3 ###
nearestNeighbour {”N”:26}
├── pearsonSimilarity
│   └── sumMatrix
│       ├── sumMatrix
│       │   ├── interaction(rating)
│       │   └── nearestNeighbour(inverted) {”N”:8}
│       │       ├── pearsonSimilarity
│       │       │   └── transpose
│       │       │       └── interaction(rating)
│       │       └── interaction(rating)
│       └── nearestNeighbour(inverted) {”N”:18}
│           ├── compareArray
│           │   ├── property(movie.actors)
│           │   └── property(movie.actors)
│           └── nearestNeighbour(inverted) {”N”:8}
│               ├── pearsonSimilarity
│               │   └── transpose
│               │       └── interaction(rating)
│               └── interaction(rating)
└── transpose
    └── sumMatrix
        ├── addVector
        │   ├── nearestNeighbour {”N”:5}
        │   │   ├── compareArray
        │   │   │   ├── property(movie.actors)
        │   │   │   └── property(movie.actors)
        │   │   └── nearestNeighbour {”N”:6}
        │   │       ├── compareArray
        │   │       │   ├── property(movie.actors)
        │   │       │   └── property(movie.actors)
        │   │       └── transpose
        │   │           └── interaction(rating)
        │   └── popularity
        │       └── transpose
        │           └── interaction(rating)
        └── transpose
            └── interaction(rating)
MRR@10=0.6138
```

A.2.2. Sobazaar

```
### Sobazaar-0 ###
transpose
└── nearestNeighbour {”N”:21}
    ├── pearsonSimilarity
    │   └── transpose
    │       └── interaction(product_wanted)
    └── scaleMatrix {”scale”:6}
        └── transpose
            └── interaction(content:interact:product_wanted)
MRR@10=0.0137
```

```
### Sobazaar-1 ###
addVector
├── transpose
│   └── transpose
│       └── interaction(purchase:buy_clicked)
└── popularity
    └── pearsonSimilarity
        └── nearestNeighbour(inverted) {"N":21}
            ├── addVector
            │   ├── scaleMatrix {"scale":1}
            │   │   └── pearsonSimilarity
            │   │       └── interaction(product_wanted)
            │   └── popularity
            │       └── pearsonSimilarity
            │           └── interaction(content:interact:product_wanted)
            └── transpose
                └── interaction(content:interact:product_clicked)
MRR@10=0.0121
```

```
### Sobazaar-2 ###
nearestNeighbour(inverted) {"N":12}
├── addVector
│   ├── pearsonSimilarity
│   │   └── transpose
│   │       └── sumMatrix
│   │           ├── interaction(content:interact:product_detail_viewed)
│   │           └── interaction(product_detail_clicked)
│   └── popularity
│       └── scaleMatrix {"scale":7}
│           └── interaction(product_wanted)
└── addVector
    ├── interaction(product_wanted)
    └── popularity
        └── interaction(product_wanted)
MRR@10=0.0124
```

```
### Sobazaar-3 ###
addVector
├── nearestNeighbour {"N":8}
│   ├── pearsonSimilarity
│   │   └── addVector
│   │       ├── interaction(purchase:buy_clicked)
│   │       └── popularity
│   │           └── interaction(product_detail_clicked)
│   └── nearestNeighbour {"N":9}
│       ├── scaleMatrix {"scale":4}
│       │   └── sumMatrix
│       │       ├── scaleMatrix {"scale":1}
│       │       │   └── pearsonSimilarity
│       │       │       └── interaction(purchase:buy_clicked)
│       │       └── sumMatrix
│       │           ├── pearsonSimilarity
│       │           │   └── interaction(content:interact:product_detail_viewed)
│       │           └── pearsonSimilarity
│       │               └── interaction(content:interact:product_wanted)
│       └── interaction(purchase:buy_clicked)
└── popularity
    └── interaction(product_wanted)
MRR@10=0.0158
```

A.2.3. Filmtrust

```
### Filmtrust-0 ###
addVector
├── sumMatrix
│   ├── interaction(rating)
│   └── interaction(rating)
└── popularity
    └── addVector
        ├── sumMatrix
        │   ├── interaction(rating)
        │   └── interaction(rating)
        └── popularity
            └── scaleMatrix {"scale":0}
                └── interaction(rating)
MRR@10=0.9961
```

```
### Filmtrust-1 ###
sumMatrix
├── addVector
│   ├── nearestNeighbour {"N":9}
│   │   ├── sumMatrix
│   │   │   ├── interaction(trust)
│   │   │   └── interaction(trust)
│   │   └── interaction(rating)
│   └── popularity
│       └── nearestNeighbour {"N":1}
│           ├── interaction(trust)
│           └── interaction(rating)
└── scaleMatrix {"scale":2}
    └── scaleMatrix {"scale":9}
        └── scaleMatrix {"scale":0}
            └── scaleMatrix {"scale":2}
                └── interaction(rating)
MRR@10=0.996


### Filmtrust-2 ###
addVector
├── interaction(rating)
└── popularity
    └── sumMatrix
        ├── interaction(rating)
        └── interaction(rating)
MRR@10=0.9955


### Filmtrust-3 ###
sumMatrix
├── addVector
│   ├── transpose
│   │   └── transpose
│   │       └── interaction(rating)
│   └── popularity
│       └── interaction(rating)
└── addVector
    ├── sumMatrix
    │   ├── nearestNeighbour {"N":19}
    │   │   ├── sumMatrix
    │   │   │   ├── interaction(trust)
    │   │   │   └── interaction(trust)
    │   │   └── interaction(rating)
    │   └── nearestNeighbour {"N":20}
    │       ├── interaction(trust)
    │       └── interaction(rating)
    └── popularity
        └── sumMatrix
            ├── interaction(rating)
            └── nearestNeighbour {"N":18}
                ├── interaction(trust)
                └── interaction(rating)
MRR@10=0.9963
```

A.3. Adaptability experiments recommender systems

A.3.1. Evaluation

```
### evaluation-mrr@10 ###
sumMatrix
├── nearestNeighbour(inverted) {"N":4}
│   ├── pearsonSimilarity
│   │   └── transpose
│   │       └── interaction(rating)
│   └── addVector
│       ├── nearestNeighbour(inverted) {"N":7}
│       │   ├── compareArray
│       │   │   ├── property(movie.actors)
│       │   │   └── property(movie.actors)
│       │   └── interaction(rating)
│       └── popularity
│           └── interaction(acts)
└── nearestNeighbour {"N":22}
    ├── scaleMatrix {"scale":3}
    │   └── pearsonSimilarity
    │       └── interaction(rating)
    └── scaleMatrix {"scale":8}
        └── interaction(rating)
MRR@10=0.6293



### evaluation-precision@1 ###
sumMatrix
├── addVector
│   ├── nearestNeighbour(inverted) {"N":16}
│   │   ├── pearsonSimilarity
│   │   │   └── transpose
│   │   │       └── interaction(rating)
│   │   └── transpose
│   │       └── transpose
│   │           └── interaction(rating)
│   └── popularity
│       └── interaction(rating)
└── addVector
    ├── addVector
    │   ├── nearestNeighbour(inverted) {"N":10}
    │   │   ├── compareArray
    │   │   │   ├── property(movie.actors)
    │   │   │   └── property(movie.actors)
    │   │   └── sumMatrix
    │   │       ├── interaction(rating)
    │   │       └── interaction(rating)
    │   └── popularity
    │       └── sumMatrix
    │           ├── interaction(rating)
    │           └── interaction(rating)
    └── popularity
        └── interaction(rating)
MRR@10=0.5966
```

```
### evaluation-recall@10 ###
nearestNeighbour {”N”:21}
├── pearsonSimilarity
│   └── nearestNeighbour(inverted) {”N”:11}
│       ├── pearsonSimilarity
│       │   └── transpose
│       │       └── interaction(acts)
│       └── scaleMatrix {”scale”:3}
│           └── scaleMatrix {”scale”:5}
│               └── interaction(rating)
└── transpose
    └── sumMatrix
        ├── sumMatrix
        │   ├── transpose
        │   │   └── interaction(rating)
        │   └── transpose
        │       └── interaction(rating)
        └── nearestNeighbour(inverted) {”N”:3}
            ├── pearsonSimilarity
            │   └── interaction(rating)
            └── transpose
                └── interaction(rating)
MRR@10=0.5843
```

A.3.2. Interaction

```
### interaction-product_detail_viewed ###
nearestNeighbour {”N”:29}
├── sumMatrix
│   ├── sumMatrix
│   │   ├── pearsonSimilarity
│   │   │   └── scaleMatrix {”scale”:0}
│   │   │       └── interaction(purchase:buy_clicked)
│   │   └── sumMatrix
│   │       ├── pearsonSimilarity
│   │       │   └── interaction(purchase:buy_clicked)
│   │       └── pearsonSimilarity
│   │           └── interaction(product_wanted)
│   └── pearsonSimilarity
│       └── addVector
│           ├── scaleMatrix {”scale”:0}
│           │   └── interaction(content:interact:product_clicked)
│           └── popularity
│               └── interaction(product_wanted)
└── transpose
    └── nearestNeighbour(inverted) {”N”:8}
        ├── scaleMatrix {”scale”:4}
        │   └── pearsonSimilarity
        │       └── interaction(product_detail_clicked)
        └── scaleMatrix {”scale”:7}
            └── transpose
                └── interaction(content:interact:product_detail_viewed)
sobazaar-product_detail_viewed MRR@10: 0.0601
sobazaar-product_detail_clicked MRR@10: 0.0866
sobazaar-product_wanted MRR@10: 0.1068
sobazaar-buy_clicked MRR@10: 0.0163
```

```
### interaction-product_detail_clicked ###
nearestNeighbour {”N”:26}
├── sumMatrix
│   ├── pearsonSimilarity
│   │   └── interaction(product_detail_clicked)
│   └── pearsonSimilarity
│       └── scaleMatrix {”scale”:0}
│           └── interaction(content:interact:product_detail_viewed)
└── nearestNeighbour {”N”:5}
    ├── sumMatrix
    │   ├── pearsonSimilarity
    │   │   └── scaleMatrix {”scale”:0}
    │   │       └── interaction(product_detail_clicked)
    │   └── pearsonSimilarity
    │       └── interaction(product_wanted)
    └── scaleMatrix {”scale”:2}
        └── sumMatrix
            ├── interaction(product_detail_clicked)
            └── interaction(content:interact:product_detail_viewed)
sobazaar-product_detail_viewed MRR@10: 0.0171
sobazaar-product_detail_clicked MRR@10: 0.1316
sobazaar-product_wanted MRR@10: 0.0834
sobazaar-buy_clicked MRR@10: 0.006
```

```
### interaction-product_wanted ###
nearestNeighbour {”N”:30}
├── pearsonSimilarity
│   └── addVector
│       ├── nearestNeighbour {”N”:11}
│       │   ├── pearsonSimilarity
│       │   │   └── interaction(product_wanted)
│       │   └── interaction(product_wanted)
│       └── popularity
│           └── interaction(content:interact:product_wanted)
└── nearestNeighbour {”N”:20}
    ├── sumMatrix
    │   ├── pearsonSimilarity
    │   │   └── interaction(product_detail_clicked)
    │   └── sumMatrix
    │       ├── pearsonSimilarity
    │       │   └── interaction(product_detail_clicked)
    │       └── pearsonSimilarity
    │           └── interaction(product_detail_clicked)
    └── interaction(product_wanted)
sobazaar-product_detail_viewed MRR@10: 0.0159
sobazaar-product_detail_clicked MRR@10: 0.1032
sobazaar-product_wanted MRR@10: 0.1717
sobazaar-buy_clicked MRR@10: 0.0081
```

```
### interaction-buy_clicked ###
nearestNeighbour {"N":6}
├── sumMatrix
│   ├── pearsonSimilarity
│   │   └── interaction(content:interact:product_detail_viewed)
│   └── sumMatrix
│       ├── scaleMatrix {"scale":10}
│       │   └── pearsonSimilarity
│       │       └── interaction(purchase:buy_clicked)
│       └── pearsonSimilarity
│           └── sumMatrix
│               ├── interaction(purchase:buy_clicked)
│               └── interaction(content:interact:product_clicked)
└── transpose
    └── nearestNeighbour(inverted) {"N":11}
        ├── pearsonSimilarity
        │   └── interaction(purchase:buy_clicked)
        └── nearestNeighbour(inverted) {"N":20}
            ├── pearsonSimilarity
            │   └── interaction(purchase:buy_clicked)
            └── transpose
                └── interaction(purchase:buy_clicked)
sobazaar-product_detail_viewed MRR@10: 0.0196
sobazaar-product_detail_clicked MRR@10: 0.0525
sobazaar-product_wanted MRR@10: 0.0625
sobazaar-buy_clicked MRR@10: 0.0176
```

A.3.3. Dense and sparse experiment recommender systems

```
### dense-0 ###
nearestNeighbour {”N”:21}
├── addVector
│   ├── addVector
│   │   ├── pearsonSimilarity
│   │   │   └── scaleMatrix {”scale”:5}
│   │   │       └── interaction(purchase:buy_clicked)
│   │   └── popularity
│   │       └── pearsonSimilarity
│   │           └── interaction(product_detail_clicked)
│   └── popularity
│       └── sumMatrix
│           ├── pearsonSimilarity
│           │   └── interaction(product_detail_clicked)
│           └── pearsonSimilarity
│               └── interaction(product_wanted)
└── nearestNeighbour {”N”:9}
    ├── pearsonSimilarity
    │   └── interaction(purchase:buy_clicked)
    └── nearestNeighbour {”N”:14}
        ├── pearsonSimilarity
        │   └── interaction(product_detail_clicked)
        └── nearestNeighbour {”N”:12}
            ├── pearsonSimilarity
            │   └── interaction(product_detail_clicked)
            └── scaleMatrix {”scale”:3}
                └── interaction(purchase:buy_clicked)
sobazaar-dense MRR@10: 0.0241
sobazaar-sparse MRR@10: 0.0135
```

```
### dense-1 ###
nearestNeighbour(inverted) {”N”:13}
├── addVector
│   ├── pearsonSimilarity
│   │   └── nearestNeighbour(inverted) {”N”:16}
│   │       ├── pearsonSimilarity
│   │       │   └── interaction(content:interact:product_detail_viewed)
│   │       └── transpose
│   │           └── interaction(product_detail_clicked)
│   └── popularity
│       └── interaction(purchase:buy_clicked)
└── transpose
    └── nearestNeighbour(inverted) {”N”:24}
        ├── pearsonSimilarity
        │   └── interaction(content:interact:product_detail_viewed)
        └── transpose
            └── interaction(product_detail_clicked)
sobazaar-dense MRR@10: 0.025
sobazaar-sparse MRR@10: 0.0049
```

```
### sparse-0 ###
nearestNeighbour {”N”:5}
├── addVector
│   ├── scaleMatrix {”scale”:0}
│   │   └── pearsonSimilarity
│   │       └── scaleMatrix {”scale”:2}
│   │           └── interaction(product_wanted)
│   └── popularity
│       └── transpose
│           └── interaction(purchase:buy_clicked)
└── addVector
    ├── scaleMatrix {”scale”:6}
    │   └── transpose
    │       └── transpose
    │           └── interaction(purchase:buy_clicked)
    └── popularity
        └── interaction(content:interact:product_wanted)
sobazaar-dense MRR@10: 0.0069
sobazaar-sparse MRR@10: 0.0134

### sparse-1 ###
nearestNeighbour(inverted) {”N”:16}
├── sumMatrix
│   ├── pearsonSimilarity
│   │   └── transpose
│   │       └── interaction(purchase:buy_clicked)
│   └── sumMatrix
│       ├── pearsonSimilarity
│       │   └── transpose
│       │       └── interaction(pixel-init)
│       └── pearsonSimilarity
│           └── transpose
│               └── interaction(content:interact:product_wanted)
└── scaleMatrix {”scale”:3}
    └── nearestNeighbour {”N”:26}
        ├── sumMatrix
        │   ├── pearsonSimilarity
        │   │   └── interaction(product_detail_clicked)
        │   └── pearsonSimilarity
        │       └── interaction(content:interact:product_wanted)
        └── sumMatrix
            ├── interaction(purchase:buy_clicked)
            └── interaction(content:interact:product_clicked)
sobazaar-dense MRR@10: 0.0062
sobazaar-sparse MRR@10: 0.0114
```

Bibliography

[1] Behnoush Abdollahi and Olfa Nasraoui. Transparency in Fair Machine Learning: the Case of Explainable Recommender Systems. pages 21–35. Springer, Cham, 2018. doi: 10.1007/978-3-319-90403-0{_}2. URL https://doi.org/10.1007/978-3-319-90403-0_2.

[2] E. Aïmeur, G. Brassard, J. M. Fernandez, and F. S. Mani Onana. Privacy-preserving demographic filtering. In *Proceedings of the ACM Symposium on Applied Computing*, volume 1, pages 872–878. Association for Computing Machinery, 2006. ISBN 1595931082. doi: 10.1145/1141277.1141479.

[3] Krisztian Balog, Filip Radlinski, and Shushan Arakelyan. Transparent, scrutable and explainable user models for personalized recommendation. In *SIGIR 2019 - Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 265–274. Association for Computing Machinery, Inc, 7 2019. ISBN 9781450361729. doi: 10.1145/3331184.3331211. URL https://doi.org/10.1145/3331184.3331211.

[4] Jack Barclay, Vimal Dhokia, and Aydin Nassehi. Generating milling tool paths for prismatic parts using genetic programming. In *Procedia CIRP*, volume 33, pages 490–495. Elsevier B.V., 1 2015. doi: 10.1016/j.procir.2015.06.060.

[5] Vito Bellini, Angelo Schiavone, Tommaso Di Noia, Azzurra Ragone, and Eugenio Di Sciascio. Knowledge-aware Autoencoders for Explainable Recommender Systems. In *ACM International Conference Proceeding Series*, pages 24–31. Association for Computing Machinery, 10 2018. ISBN 9781450366175. doi: 10.1145/3270323.3270327. URL https://doi.org/10.1145/3270323.3270327.

[6] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[7] James Bennett and Stan Lanning. The Netflix Prize. *KDD Cup and Workshop*, pages 3–6, 2007. ISSN 1554351X. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.6998&rep=rep1&type=pdf.

[8] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 7 2013. ISSN 09507051. doi: 10.1016/j.knosys.2013.03.012.

[9] Erion Çano and Maurizio Morisio. Hybrid recommender systems: A systematic literature review, 1 2017. ISSN 15714128. URL https://recsys.acm.org/.

[10] Rose Catherine and William Cohen. Personalized recommendations using knowledge graphs: A probabilistic logic programming approach. In *RecSys 2016 - Proceedings of the 10th ACM Conference on Recommender Systems*, pages 325–332, New York, NY, USA, 2016. ACM. ISBN 9781450340359. doi: 10.1145/2959100.2959131. URL http://dx.doi.org/10.1145/2959100.2959131.

[11] Benjamin Paul Chamberlain, Stephen R. Hardwick, David R. Wardrope, Fabon Dzogang, Fabio Daolio, and Saúl Vargas. Scalable Hyperbolic Recommender Systems. *ACM Reference Format*, 2 2019. URL http://arxiv.org/abs/1902.08648.

[12] Edjalma Queiroz Da Silva, Celso G. Camilo-Junior, Luiz Mario L. Pascoal, and Thierson C. Rosa. An evolutionary approach for combining results of recommender systems techniques based on collaborative filtering. *Expert Systems with Applications*, 53:204–218, 7 2016. ISSN 09574174. doi: 10.1016/j.eswa.2015.12.050.

[13] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to Weave an Information tapestry. *Communications of the ACM*, 35(12):61–70, 1 1992. ISSN 15577317. doi: 10.1145/138859.138867.

[14] Ivo Gonçalves and Sara Silva. Balancing learning and overfitting in genetic programming with interleaved sampling of training data. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7831 LNCS, pages 73–84. Springer, Berlin, Heidelberg, 2013. ISBN 9783642372063. doi: 10.1007/978-3-642-37207-0{_}7. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/978-3-642-37207-0_7.

[15] G. Guo, J. Zhang, and N. Yorke-Smith. A novel bayesian similarity measure for recommender systems. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2619–2625, 2013.

[16] Guibing Guo, Jie Zhang, and Neil Yorke-Smith. TrustSVD: Collaborative filtering with both the explicit and implicit influence of user trust and of item ratings. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, pages 123–129, 2015. ISBN 9781577356998. URL www.public.asu.edu/.

[17] Guibing Guo, Jie Zhang, and Neil Yorke-Smith. A Novel Recommendation Model Regularized with User Trust and Item Ratings. *IEEE Transactions on Knowledge and Data Engineering*, 28(7): 1607–1620, 7 2016. ISSN 15582191. doi: 10.1109/TKDE.2016.2528249.

[18] Guibing Guo, Jie Zhang, and Neil Yorke-Smith. A novel evidence-based Bayesian similarity measure for recommender systems. *ACM Transactions on the Web*, 10(2), 2016. ISSN 1559114X. doi: 10.1145/2856037.

[19] Anant Gupta and B. K. Tripathy. A generic hybrid recommender system based on neural networks. In *Souvenir of the 2014 IEEE International Advance Computing Conference, IACC 2014*, pages 1248–1252. IEEE Computer Society, 2014. doi: 10.1109/IAdCC.2014.6779506.

[20] Shweta Gupta and Vibhor Kant. A Comparative Analysis of Genetic Programming and Genetic Algorithm on Multi-Criteria Recommender Systems. In *2020 5th International Conference on Communication and Electronics Systems (ICCES)*, pages 1338–1343. IEEE, 6 2020. ISBN 978-1-7281-5371-1. doi: 10.1109/icces48766.2020.9138051. URL https://ieeexplore.ieee.org/document/9138051/.

[21] Shweta Gupta and Vibhor Kant. An aggregation approach to multi-criteria recommender system using genetic programming. *Evolving Systems*, 11(1):29–44, 3 2020. ISSN 18686486. doi: 10.1007/s12530-019-09296-3. URL https://doi.org/10.1007/s12530-019-09296-3.

[22] Mohammed Hassan and Mohamed Hamada. Evaluating the performance of a neural network-based multi-criteria recommender system. *International Journal of Spatio-Temporal Data Science*, 1(1):54, 2019. ISSN 2399-1275. doi: 10.1504/ijstds.2019.097617.

[23] David M. Hofmann. A genetic programming approach to generating musical compositions. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9027, pages 89–100. Springer Verlag, 2015. ISBN 9783319164977. doi: 10.1007/978-3-319-16498-4{_}9. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/978-3-319-16498-4_9.

[24] Chein Shung Hwang. Genetic algorithms for feature weighting in multi-criteria recommender systems. *Journal of Convergence Information Technology*, 5(8):13, 2010. ISSN 19759320. doi: 10.4156/jcit.vol5.issue8.13. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.499.6627.

[25] Gawesh Jawaheer, Peter Weller, and Patty Kostkova. Modeling user preferences in recommender systems: A classification framework for explicit and implicit user feedback. *ACM Transactions on Interactive Intelligent Systems*, 4(2):1–26, 7 2014. ISSN 21606463. doi: 10.1145/2512208. URL https://dl.acm.org/doi/10.1145/2512208.

[26] Jose L. Jorro-Aragoneses, Juan A. Recio-García, Belén Díaz-Agudo, and Guillermo Jimenez-Díaz. RECOLIBRY-CORE: A component-based framework for building recommender systems. *Knowledge-Based Systems*, 182:104854, 10 2019. ISSN 09507051. doi: 10.1016/j.knosys.2019.07.025.

[27] Maarten Keijzer. Alternatives in Subtree Caching for Genetic Programming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3003:328–337, 2004. ISSN 16113349. doi: 10.1007/978-3-540-24650-3{_}31. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/978-3-540-24650-3_31.

[28] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. ISSN 00189162. doi: 10.1109/MC.2009.263.

[29] Pigi Kouki, Shobeir Fakhraei, James Foulds, Magdalini Eirinaki, and Lise Getoor. HyPER: A flexible and extensible probabilistic framework for hybrid recommender systems. In *RecSys 2015 - Proceedings of the 9th ACM Conference on Recommender Systems*, pages 99–106, New York, New York, USA, 9 2015. Association for Computing Machinery, Inc. ISBN 9781450336925. doi: 10.1145/2792838.2800175. URL http://dl.acm.org/citation.cfm?doid=2792838.2800175.

[30] John R Koza. *Genetic programming: On the programming of computers by means of natural selection*, volume 1. MIT Press, 1992. doi: 10.1016/0303-2647(94)90062-0. URL https://books.google.com/books?hl=en&lr=&id=Bhtxo60BV0EC&oi=fnd&pg=PR11&dq=genetic+programming+book&ots=9rcSdvj-MT&sig=hBWMimHuKCTbhXQuslLblH14ZQk.

[31] Raúl Lara-Cabrera, angel gonzalez prieto@upm es Ángel González-Prieto, Fernando Ortega, and jesus bobadilla@upm es Jesús Bobadilla. Evolving matrix-factorization-based collaborative filtering using genetic programming. *Applied Sciences (Switzerland)*, 10(2):675, 1 2020. ISSN 20763417. doi: 10.3390/app10020675. URL https://www.mdpi.com/2076-3417/10/2/675.

[32] Yuri Lavinas, Claus Aranha, Tetsuya Sakurai, and Marcelo Ladeira. Experimental Analysis of the Tournament Size on Genetic Algorithms. In *Proceedings - 2018 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2018*, pages 3647–3653. Institute of Electrical and Electronics Engineers Inc., 1 2019. ISBN 9781538666500. doi: 10.1109/SMC.2018.00617.

[33] Jie Lu, Dianshuang Wu, Mingsong Mao, Wei Wang, and Guangquan Zhang. Recommender system application developments: A survey. *Decision Support Systems*, 74:12–32, 6 2015. ISSN 01679236. doi: 10.1016/j.dss.2015.03.008.

[34] Sean Luke and Liviu Panait. A survey and comparison of tree generation algorithms. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88, 2001. URL http://www.cs.gmu.edu/∼sean/http://www.cs.gmu.edu/∼lpanait/.

[35] David J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 6 1995. ISSN 1063-6560. doi: 10.1162/evco.1995.3.2.199. URL http://direct.mit.edu/evco/article-pdf/3/2/199/1492842/evco.1995.3.2.199.pdf.

[36] Hai Thanh Nguyen, Thomas Almenningen, Martin Havig, Herman Schistad, Anders Kofod-Petersen, Helge Langseth, and Heri Ramampiaro. Learning to rank for personalised fashion recommender systems via implicit feedback. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8891, pages 51–61. Springer Verlag, 2014. ISBN 9783319138169. doi: 10.1007/978-3-319-13817-6{_}6. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/978-3-319-13817-6_6.

[37] Samuel Oliveira, Victor Diniz, Anisio Lacerda, and Gisele L. Pappa. Evolutionary rank aggregation for recommender systems. In *2016 IEEE Congress on Evolutionary Computation, CEC*

2016, pages 255–262. Institute of Electrical and Electronics Engineers Inc., 11 2016. ISBN 9781509006229. doi: 10.1109/CEC.2016.7743803.

[38] Anand Kishor Pandey and Dharmveer Singh Rajpoot. Resolving Cold Start problem in recommendation system using demographic approach. In *2016 International Conference on Signal Processing and Communication, ICSC 2016*, pages 213–218. Institute of Electrical and Electronics Engineers Inc., 2016. ISBN 9781509026845. doi: 10.1109/ICSPCom.2016.7980578.

[39] Nymphia Pereira and Satishkumar L. Varma. Financial planning recommendation system using content-based collaborative and demographic filtering. In *Advances in Intelligent Systems and Computing*, volume 669, pages 141–151. Springer Verlag, 2019. ISBN 9789811089671. doi: 10.1007/978-981-10-8968-8{_}12. URL https://doi.org/10.1007/978-981-10-8968-8_12.

[40] Riccardo Poli, Nicholas Freitag McPhee, and Leonardo Vanneschi. Elitism reduces bloat in genetic programming. In *GECCO'08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation 2008*, pages 1343–1344. Association for Computing Machinery (ACM), 2008. ISBN 9781605581309. doi: 10.1145/1389095.1389355.

[41] David M. W. Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. 10 2020. URL http://arxiv.org/abs/2010.16061.

[42] Christian Räck, Stefan Arbanowski, and Stephan Steglich. A generic multipurpose recommender system for contextual recommendations. In *Proceedings - Eighth International Symposium on Autonomous Decentralized Systems, ISADS 2007*, pages 445–450, 2007. ISBN 076952804X. doi: 10.1109/ISADS.2007.2.

[43] Mark E. Roberts. The effectiveness of cost based subtree caching mechanisms in typed genetic programming for image segmentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2611:444–454, 2003. ISSN 16113349. doi: 10.1007/3-540-36605-9{_}41. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/3-540-36605-9_41.

[44] Maryam Sadeghi and Seyyed Amir Asghari. Recommender Systems Based on Evolutionary Computing: A Survey. *Journal of Software Engineering and Applications*, 10(05):407–421, 5 2017. ISSN 1945-3116. doi: 10.4236/jsea.2017.105023. URL http://www.scirp.org/journal/jsea.

[45] Aghiles Salah and Hady W. Lauw. A Bayesian latent variable model of user preferences with item context. In *IJCAI International Joint Conference on Artificial Intelligence*, volume 2018-July, pages 2667–2674, 2018. ISBN 9780999241127. doi: 10.24963/ijcai.2018/370.

[46] Aghiles Salah, Quoc Tuan Truong, and Hady W. Lauw. Cornac: A comparative framework for multimodal recommender systems. Technical report, 2020. URL http://jmlr.org/papers/v21/19-805.html.

[47] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web, WWW 2001*, pages 285–295, 2001. ISBN 1581133480. doi: 10.1145/371920.372071.

[48] Dirk Schweim, David Wittenberg, and Franz Rothlauf. On sampling error in genetic programming. *Natural Computing*, pages 1–14, 1 2021. ISSN 15729796. doi: 10.1007/s11047-020-09828-w. URL https://doi.org/10.1007/s11047-020-09828-w.

[49] Xiaoyu Shi, Qiang He, Xin Luo, Yannai Bai, and Mingsheng Shang. Large-scale and Scalable Latent Factor Analysis via Distributed Alternative Stochastic Gradient Descent for Recommender Systems. *IEEE Transactions on Big Data*, 2020. ISSN 23327790. doi: 10.1109/TBDATA.2020.2973141.

[50] Monika Singh. Scalability and sparsity issues in recommender datasets: a survey. *Knowledge and Information Systems*, 62(1):1–43, 1 2020. ISSN 02193116. doi: 10.1007/s10115-018-1254-2. URL https://doi.org/10.1007/s10115-018-1254-2.

[51] Zhu Sun, Jie Yang, Jie Zhang, Alessandro Bozzon, Long Kai Huang, and Chi Xu. Recurrent knowledge graph embedding for effective recommendation. In *RecSys 2018 - 12th ACM Conference on Recommender Systems*, pages 297–305, New York, NY, USA, 2018. ACM. ISBN 9781450359016. doi: 10.1145/3240323.3240361. URL https://doi.org/10.1145/3240323.3240361.

[52] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1): 42–47, Feb 2011. doi: 10.5281/zenodo.16303. URL http://www.gnu.org/s/parallel.

[53] Frank Edward Walter, Stefano Battiston, and Frank Schweitzer. A model of a trust-based recommendation system on a social network. *Autonomous Agents and Multi-Agent Systems*, 16(1): 57–74, 2008. ISSN 13872532. doi: 10.1007/s10458-007-9021-x.

[54] Peter A. Whigham and Grant Dick. Implicitly controlling bloat in genetic programming. *IEEE Transactions on Evolutionary Computation*, 14(2):173–190, 4 2010. ISSN 1089778X. doi: 10.1109/TEVC.2009.2027314.

[55] Phillip Wone and Menejie Zhang. SCHEME: Caching subtrees in genetic programming. In *2008 IEEE Congress on Evolutionary Computation, CEC 2008*, pages 2678–2685, 2008. ISBN 9781424418237. doi: 10.1109/CEC.2008.4631158.

[56] Billy Yapriady and Alexandra L. Uitdenbogerd. Combining demographic data with collaborative filtering for automatic music recommendation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3684 LNAI, pages 201–207. Springer Verlag, 2005. ISBN 354028897X. doi: 10.1007/11554028{_}29. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/11554028_29.

[57] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. Personalized entity recommendation: A heterogeneous information network approach. In *WSDM 2014 - Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pages 283–292. Association for Computing Machinery, 2014. ISBN 9781450323512. doi: 10.1145/2556195.2556259. URL http://dx.doi.org/10.1145/2556195.2556259.

[58] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives, 2 2019. ISSN 15577341. URL https://doi.org/10.1145/3285029.

[59] Wayne Xin Zhao, Sui Li, Yulan He, Liwei Wang, Ji Rong Wen, and Xiaoming Li. Exploring demographic information in social media for product recommendation. *Knowledge and Information Systems*, 49(1):61–89, 10 2016. ISSN 02193116. doi: 10.1007/s10115-015-0897-5. URL http://www.brandwatch.com/wp-content/uploads/2013/02/Twitter-Landscape-2013-Extended-Version.

[60] Xin Wayne Zhao, Yanwei Guo, Yulan He, Han Jiang, Yuexin Wu, and Xiaoming Li. We know what you want to buy: A demographic-based system for product recommendation on microblogs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume 14, pages 1935–1944, New York, NY, USA, 2014. ACM. ISBN 9781450329569. doi: 10.1145/2623330.2623351. URL http://dx.doi.org/10.1145/2623330.2623351.