Dynamische generatie van gepersonaliseerde hybride aanbevelingssystemen

Dynamic Generation of Personalized Hybrid Recommender Systems

Simon Dooms

Promotor: prof. dr. ir. L. Martens
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. D. De Zutter
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2014 - 2015

UNIVERSITEIT
GENT

ISBN 978-90-8578-755-6
NUR 984, 988
Wettelijk depot: D/2014/10.500/101

Department of Information Technology
WiCa Research Group
Faculty of Engineering and Architecture
Ghent University

WiCa
Universiteit Gent - iMinds
Campus Zuiderpoort, Blok C
Gaston Crommenlaan 8 bus 201
9050 Gent
België

This work was funded by a PhD grant of the Agency for Innovation by Science and Technology (IWT).

The Road Not Taken

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I-
I took the one less traveled by,
And that has made all the difference.

Robert Frost

Acknowledgements

They say that writing the acknowledgements is the hardest part of a dissertation, and while I don't completely agree with that (I struggled for days to get the title right), it's certainly one of the more sensitive parts. Scientifically speaking, everyone who I interacted with between September 2009 and December 2014 in some way influenced me, which either directly or indirectly may have aided the realization of this book. So let's start with a big thanks to everyone!

While it would be quite funny to stop at this point, common courtesy dictates that I highlight some of the more influential contributors and supporters of both my academic and personal development of these very special 5 years of my life. So here goes. Thanks to my advisor prof. dr. ir. Luc Martens for offering the great opportunity to embark on a PhD quest after I had successfully completed my master thesis at the WiCa research lab. Also a huge thanks to all of my WiCa colleagues (you know your names), either for the support, company, or the delightful take-your-mind-off-work conversations that often took place at the WiCa lunch table. Thanks to all my co-authors for the collaboration, proof-reading and improving suggestions to my published work. Thanks to the Flemish government (and by extension all tax payers) for financially supporting my research. Thanks to all Ghent University administrative and supporting staff for helping out with the bureaucratic burden, especially Isabelle who often times had to decrypt foreign expense claims in the most exotic languages. Special mention to the HPCUGent Team who were always available for technical support and made possible the high-performance track of this work.

Thanks to all members of the recsys community for making me feel at home every year at the ACM RecSys conference starting from Barcelona 2010. It was a great pleasure getting to know the faces behind the papers, thanks for the many many mind-expanding conversations, research ideas and cultural enrichment both in and outside the conference halls.

Verder had ik ook graag alle externe instellingen en partners bedankt waar ik gedurende deze 5 jaar mee in aanraking kwam. Bijvoorbeeld het team van CultuurNet Vlaanderen, hartelijk bedankt voor de hoge bereidheid tot samenwerking, het aanbieden van een onderzoeksplatform en de aanzienlijke inspanningen die jullie leverden voor de ondersteuning van experimenten en onderzoek naar een betere gebruikerservaring. Super bedankt!

Een mega merci aan al mijn vrienden in cirkels groot en klein, voor plezier moest ik steeds bij jullie zijn. Op vrijdag, zaterdag of gewoon tijdens de week, pintjes en Duvels waren er altijd, zo bleek. Zij het in Puglia, Keulen, het Veerse Meer, de Ardennen, Maarkedal, De Pinte, Zwijnaarde, 't Dreupelkot, Hot Club de Gand, 't Einde, of gelijk waar we samen zijn beland, fijn is het altijd en ge zijt allemaal stuk voor stuk, *stief wel* bedankt! En 't is waar, want het rijmt.

Extra merci aan Bert en Bart, mijn doctoraatscompagnions doorheen dit avontuur op wie ik steeds kon rekenen voor wijze raad, overleg en het aflaten van PhD-stoom.

Dankjewel ook aan alle familie zowel ver weg als dichtbij voor de vele aangename niet-werk momenten tussen de weken door. Dankjewel Mama, Papa, Emma, July, Willem, Joost, Eliane, Jolan, Jesse en Sander voor de bergen liefde, steun en vertrouwen. De allerlaatste dankjewel, tenslotte, is gereserveerd voor mijn allerbeste maatje, mijn dagelijkse steun en toeverlaat, het zout op mijn nootjes en de liefde van mijn leven, mijn Mieke. Dankjewel!

Simon Dooms, 19 december 2014

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

API	Application Programming Interface

B

BS	Best Switching

C

CAMRA	Context-Aware Movie Recommendation
CB	Content-Based
CF	Collaborative Filtering
CSS	Cascading Style Sheets

D

DCF	Distributed Collaborative Filtering
DCFLA	Distributed Collaborative Filtering neighbor-Locating Algorithm
DLNA	Digital Living Network Alliance
DMR	Digital Media Renderer

F

FWLS	Feature-Weighted Linear Stacking

G

GPFS	General Parallel File System
GUI	Graphical User Interface

H

HCI	Human-Computer Interaction
HDFS	Hadoop Distributed File System
HPC	High-Performance Computing
HRI	Human-Recommender Interaction
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol

I

ID	Identifier
IMDb	Internet Movie Database
IP	Internet Protocol
IR	Information Retrieval

K

KNN	K-Nearest Neighbor

L

LAMP	Linux Apache MySQL PHP
LRU	Least Recently Used

M

MAE	Mean Absolute Error
MAP	Mean Absolute Precision
ML	MovieLens
MPAA	Motion Picture Association of America
MPMD	Multiple Program Multiple Data
MT	MovieTweetings

N

NDCG	Normalized Discounted Cumulative Gain
NP	Nondeterministic polynomial Time

O

OMDb	Open Movie Database
OMUS	Optimized MUltimedia Service

R

RAID	Redundant Array of Inexpensive Disks
RAM	Random-Access Memory
RDBMS	Relational DataBase Management System
REQ	REQuirement
RMSE	Root Mean Square Error
RPC	Remote Procedure Call

S

SPMD	Single Program Multiple Data
STREAM	Stacking Recommendation Engines with Additional Meta-features
SVD	Singular Value Decomposition

U

UBCF	User-Based Collaborative Filtering
UPnP	Universal Plug and Play
UPnP AV	Universal Plug and Play Audio Video
URL	Uniform Resource Locator
UI	User Interface

X

XML	Extensible Markup Language

English Summary

Nederlandstalige samenvatting

English Summary

Partly due to the Internet, we are nowadays virtually surrounded with more information than we could possibly process in one lifetime. Every day an increasing number of books, news articles, movies, music and many more scream for attention, while our attention time itself remains limited. Since manually browsing for interesting content is no longer viable, the research and development of recommender systems has become a hot topic in both academia and industry. Recommender systems try to bridge the gap between users and content by modeling user preferences and autonomously unearthing interesting items which would otherwise have remained deeply buried in vast content catalogs.

Since the introduction of the first recommender systems in the early 90s, many recommendation strategies and algorithms have become available. There are algorithms that focus on user ratings, item information, location, context, social relations and many more. In fact, there are so many nowadays, that the challenge for researchers has now shifted from designing new algorithms to selecting the most optimal strategy for a given scenario, user and context. While, in the past, this task most often required manual involvement to configure one or multiple (i.e., hybrid) recommender systems, with this work we strive towards a future where a recommender system is capable of integrating many individual recommendation algorithms and dynamically fine-tunes itself for a given situation.

Most recommendation strategies start from the idea that users in a system behave very similarly and thus can be averaged out or clustered together. Throughout this work however on multiple occasions we found users to be very different from each other and so wherever possible we opted for a user-specific approach where every user is considered unique and therefore requires a unique approach.

We started by exploring the components of a recommender system,

one of which was the collection of user input data. While essential for the recommendation process, user input data or ‘feedback’ proved hard to collect and public rating datasets were shown to be outdated. We ran a number of experiments on a popular Belgian events website, and noted the difference between implicit and explicit user feedback. While the first was easier to collect, its information value proved less than the latter. We bundled the collected data from the experiments as an event rating dataset. A time analysis of the data however showed how user feedback on events had some very specific properties which may limit its usefulness to one-and-only items. To obtain a recent, relevant, and more generalizable rating dataset we turned to social media. We constructed a movie rating dataset called *MovieTweetings* which mined IMDb movie ratings that were posted on Twitter in a semi-structured format. Our mining approach was shown to be effective in other item domains (e.g., books or music) as well, and even showed potential for cross-domain item recommendation.

Next, we analyzed the importance of user interfaces in the recommendation process and evaluated how a sense of control and system transparency could be integrated in a hybrid recommendation scenario. We focused on user-recommender interaction both from the perspective of feedback mechanisms and recommendation presentation. Above all, users proved to have very diverging opinions on their ideal interaction process. While some users want to be in control and heavily interact with a recommender system, other users want a lean-back experience and expect the system to just ‘work’.

Because of the high computational complexity and hardware requirements, most experiments were executed on a high-performance computing (HPC) infrastructure, which was available for researchers at our university. We showed how even complex recommendation calculations could be mapped onto such an infrastructure. While we experimented with both functional and data parallelism, we found the latter to be more interesting because it allowed in-memory computation, improved load balancing and reduced synchronization overhead between multiple calculation phases. We showed how a recommendation task distributed over 200 worker nodes (each of which had 8 processing cores) still obtained parallel efficiency values up to 70%, which surpassed existing state-of-the-art distributed recommendation algorithms that used a MapReduce paradigm.

With the availability of a dataset, user interface and computing in-

frastructure, we could then focus on our original research goal of developing self-configuring hybrid recommender systems. We approached this challenge in three phases. First, we considered the hybrid configuration of a recommender system as an optimization problem in an offline setting. We showed how individual recommendation algorithms could be combined automatically, while allowing users to manually influence the outcome. Second, we revisited our optimization strategy in an online context. We introduced a client-server architecture and showed how it allowed the system to meet four requirements typically found in an online environment i.e., scalability, responsiveness, user control and system transparency. Third, we exposed the self-learning hybrid recommender system to actual users in an online evaluation experiment that was set in a movie recommendation context. We linked a HPC back end to a Google Chrome extension front end called *MovieBrain* by means of a middleware webserver. Hybrid recommender configurations were automatically optimized in real-time, and personalized for every user. We logged all user interaction and activity patterns of 70 users over a period of 107 days. An analysis of both implicitly and explicitly collected user feedback showed how users enjoyed the improved recommendation experience over time.

While, ironically, recommender systems are designed to avoid manual involvement in the content selection process as much as possible, we believe active user involvement to be the key which will enable next generation recommender systems to push beyond the current limits and towards a true and optimal recommendation experience.

This research was performed in the Wireless & Cable (WiCa) research group. WiCa is a research group in the Information Technology (INTEC) department of the Faculty of Engineering and Architecture (FEA) of Ghent University, Belgium. The work presented here resulted in 5 international journal publications, 8 conference or workshop papers, 1 demo and 1 newsletter (all first author).

Nederlandstalige samenvatting

Eén van de grootste problemen waar internetgebruikers tegenwoordig mee geconfronteerd worden, is de overdaad aan informatie die op hen afkomt. Elke dag opnieuw schreeuwen een steeds maar toenemende hoeveelheid boeken, nieuwsartikels, films, muziek enz. om aandacht. De hoeveelheid informatie is zo hoog dat zelf een volledig mensenleven niet voldoende is om alles te verwerken. Als persoon is het dus onmogelijk om zelf door het volledige beschikbare aanbod te bladeren en manueel te selecteren wat interessant is. Om dit steeds actueler wordende probleem aan te pakken, wordt er met groeiende belangstelling gewerkt aan de ontwikkeling van aanbevelingssystemen; en dit zowel in de academische wereld als in de bedrijfswereld. Het primaire doel van aanbevelingssystemen is de kloof tussen gebruiker en aanbod te dichten door geautomatiseerd op zoek te gaan naar de meest interessante informatie voor elke gebruiker.

Sinds de introductie van de eerste aanbevelingssystemen in de vroege jaren negentig zijn reeds een groot aantal aanbevelingsalgoritmes en strategieën ontworpen. Sommige algoritmes focussen zich op de score, of *ratings*, die gebruikers geven, anderen baseren zich eerder op specifieke informatie zoals: *item* metadata (bijvoorbeeld een filmgenre), locatie, context, sociale relaties, enz. Tegenwoordig is het aantal beschikbare aanbevelingsalgoritmes zo hoog dat de werkelijke uitdaging voor onderzoekers zich almaar meer verplaatst van het ontwerpen van nieuwe algoritmes, naar het optimaal selecteren van het beste algoritme voor elke gegeven situatie, context en gebruiker. De beste methode kan één enkel algoritme zijn, of uit een combinatie van meerdere algoritmes bestaan. Combinaties van meerdere algoritmes worden vaak gebruikt omdat ze de individuele voordelen van elk van de geïntegreerde algoritmes kunnen combineren. Een aanbevelingssysteem dat meerdere verschillende aanbevelingsalgoritmes integreert, noemen we dan een hybride systeem. Dergelijke hybride systemen vereisen tot op vandaag meestal nog manuele configuratie om de verschillende onderliggende algoritmes op elkaar af te stellen. Met dit werk richten we ons echter op de toekomst, waar een aanbevelingssysteem in staat zou moeten zijn om, gegeven een aantal individuele aanbevelingsalgoritmes, zichzelf automatisch samen te stellen tot een uniek hybride systeem dat optimaal afgesteld is op een gegeven situatie.

De basis van de meeste aanbevelingsstrategieën is dat gebruikers in een systeem zich gelijkaardig gedragen en dus uitgemiddeld of in clusters gegroepeerd kunnen worden. Doorheen dit werk echter, kwamen we meerdere keren tot de omgekeerde vaststelling dat gebruikers er soms volledig tegengestelde meningen en gedragspatronen blijken op na te houden. We probeerden dan ook zoveel mogelijk van een gebruikersspecifieke aanpak uit te gaan, waar we elke gebruiker als uniek beschouwden en een unieke methode van aanbevelen aanboden.

De eerste stap in ons onderzoek bestond uit het bestuderen van de verschillende facetten of componenten van een aanbevelingssysteem. Een van deze componenten was het verzamelen van gebruikersdata. Hoewel deze data van essentieel belang is voor de goede werking van een aanbevelingssysteem, bleek het verzamelen ervan niet evident te zijn. Er bestaan publiek beschikbare datasets die vaak voor experimentele doeleinden worden gebruikt, maar deze zijn ofwel erg verouderd ofwel weinig algemeen inzetbaar. We voerden een eerste reeks van experimenten en analyses uit op een populaire Belgische evenementen website (uitinvlaanderen.be), en ondervonden zo het verschil tussen impliciete en expliciete gebruikersdata. Terwijl impliciete gegevens gemakkelijk te verzamelen zijn, vereist expliciete feedback steeds bewuste interactie van gebruikers. Hoewel expliciete feedback daardoor meestal minder voorhanden is, ligt de informatiewaarde ervan wel een stuk hoger. De data die volgde uit de experimenten werd gebundeld in een dataset die in verder onderzoek zou kunnen worden gebruikt. Een tijdsanalyse van de data bracht echter enkele eigenschappen aan het licht die te specifiek van toepassing waren op plaats- en tijdsgebonden items zoals evenementen. Om een meer algemenere dataset van gebruikersvoorkeuren te bekomen, richtten we ons uiteindelijk op sociale media. Een tweede reeks van experimenten leidde tot de ontwikkeling van een eigen dataset genaamd *MovieTweetings*, die bestond uit verzamelde filmvoorkeuren die via IMDb gepost werden op Twitter in een semi-gestructureerd formaat. We toonden bovendien aan hoe onze manier van werken eveneens toegepast kon worden op andere item domeinen (zoals boeken en muziek), en zelfs erg nuttig zou kunnen zijn voor aanbevelingssystemen die over meerdere domeinen heen werken.

In een volgende stap, analyseerden we het belang van gebruikersinterfaces in het aanbevelingsproces en we evalueerden hoe het idee van controle en systeemtransparantie kon worden geïntegreerd in een hybride aanbevelingsscenario. De focus lag hierbij op de interactie tussen gebruiker en aanbevelingssysteem, zowel vanuit het perspectief van feedback mechanismes als van de visualisatie van de aanbevelingen zelf. Uit meerdere concrete *use cases* bleek hoe gebruikers er vaak erg tegengestelde opinies op na houden. Terwijl sommige gebruikers in controle willen zijn en actief interageren met het systeem, verkiezen anderen een meer *lean-back* ervaring terwijl ze verwachten dat het systeem gewoon 'werkt'.

Vanwege de vaak hoge computationele vereisten van aanbevelingsalgoritmes, drong de noodzaak van een *high-end computing* infrastructuur zich op. De meeste experimenten werden dan ook uitgevoerd op de supercomputer infrastructuur (*High-Performance Computing* of HPC) die beschikbaar werd gesteld door de universiteit. We toonden aan hoe de meestal erg complexe aanbevelingsberekeningen konden worden aangepast zodat ze parallel en gedistribueerd op de rekeninfrastructuur konden worden uitgevoerd. We experimenteerden hierbij zowel met een functionele als met een op data gefocuste aanpak van parallellisatie. Deze laatste bleek het meest interessant omdat het *in-memory* berekeningen ondersteunde, *load balancing* verbeterde en de *overhead* afkomstig van synchronisatie tussen meerdere opeenvolgende fases kon vermijden. We illustreerden hoe een aanbevelingsberekening gedistribueerd over 200 computers (elk met 8 processors) nog steeds een parallelle efficiëntiewaarde van 70% wist te bereikten. Dit bleek hoger te zijn (en dus beter) dan huidige state-of-the-art gedistribueerde aanbevelingsalgoritmes die werken met een *MapReduce* aanpak.

Met de beschikbaarheid van een dataset, een gebruikersinterface en een rekeninfrastructuur konden we onze aandacht vervolgens vestigen op het ontwerpen van een zichzelf configurerend hybride aanbevelingssysteem. We benaderden deze uitdaging in drie fasen. In een eerste fase beschouwden we het configureren van een hybride aanbevelingssysteem als een optimalisatieprobleem in een offline context. We onderzochten hoe de resultaten van individuele aanbevelingsalgoritmes automatisch gecombineerd konden worden terwijl ook ondersteuning voor manuele gebruikersinvloed gegarandeerd werd. In een tweede fase werd de optimalisatiestrategie herbekeken vanuit een online context. We introduceerden een *client-server* architectuur en toonden aan hoe het systeem kon voldoen aan vier typische eisen voor online omgevingen zijnde schaalbaarheid, responsiviteit, gebruikerscontrole en systeemtransparantie. In een derde en laatste fase werd ons autonoom lerend hybride aanbevelingssysteem blootgesteld aan echte gebruikers in een online experiment toegepast op film aanbevelingen. Een achterliggende supercomputer infrastructuur werd verbonden met een zelfgemaakte Google Chrome extension genaamd *MovieBrain* via een tussenliggende webserver. Hybride configuraties van individuele aanbevelingsalgoritmes werden automatisch geoptimaliseerd in ware tijd, en dit voor elke unieke gebruiker. Gebruikers waren bovendien in staat om de volgens het systeem optimale configuratie te wijzigen en beter op hun eigen voorkeuren af te stellen. Van in totaal zeventig gebruikers verzamelden we interactie- en activiteitspatronen over een periode van 107 dagen. Uit een analyse van zowel impliciete gegevens als expliciet verzamelde gebruikersvoorkeuren bleek dat gebruikers de verbeterde aanbevelingservaring erg wisten te appreciëren.

Aanbevelingssystemen werden ontworpen om zo veel mogelijk automatisch en autonoom ondersteuning te bieden bij het selecteren van interessante items. Niet zonder enige ironie zijn we echter van mening dat net manuele interactie in het aanbevelingsproces de sleutel vormt tot het doorbereken van de huidige limieten en uiteindelijk kan leiden tot het bereiken van een ware en zo optimaal mogelijke aanbevelingservaring.

Dit onderzoek werd uitgevoerd in de Wireless & Cable (WiCa) onderzoeksgroep. WiCa is een onderzoeksgroep binnen het departement Informatietechnologie (INTEC) in de Faculteit Ingenieurswetenschappen en Architectuur (FEA) van Universiteit Gent, België. Dit hier gepresenteerde werk heeft geleid tot 5 artikels in internationale wetenschappelijke tijdschriften, 8 artikels gepresenteerd op conferenties of workshops, 1 demo en 1 wetenschappelijke nieuwsbrief (allemaal eerste auteur).

List of Publications

Journal Papers

[1] **Dooms S**, De Pessemier T, Verslype D, Nelis J, De Meulenaere J, Van den Broeck W, Martens M, Develder C. Omus: an optimized multimedia service for the home environment *Multimedia Tools and Applications* 2014;72(1):281-311.

[2] **Dooms S**, Audenaert P, Fostier J, De Pessemier T, Martens L. In-memory, distributed content-based recommender system *Journal of Intelligent Information Systems* 2014;42(3):645-669.

[3] **Dooms S**, De Pessemier T, Martens L. Offline optimization for user-specific hybrid recommender systems *Multimedia Tools and Applications* 2013; accepted.

[4] **Dooms S**, De Pessemier T, Martens L. Online optimization for user-specific hybrid recommender systems *Multimedia Tools and Applications* 2014; accepted.

[5] **Dooms S**, Bellogin A, De Pessemier T, Martens L. A Framework for Dataset Benchmarking and its Application to a New Movie Rating Dataset *ACM Transactions on Intelligent Systems and Technology* (under review).

International Conferences

[1] **Dooms S**, De Pessemier T, Martens L. An online evaluation of explicit feedback mechanisms for recommender systems *Proc. of Conf. on Web Information Systems and Technologies (WEBIST)* 2011;391-394.

[2] **Dooms S**, De Pessemier T, Martens L. Caching strategies for in-memory neighborhood-based recommender systems *Proc. of Conf. on Web Information Systems and Technologies (WEBIST)* 2013;435-440.

[3] **Dooms S**. Dynamic generation of personalized hybrid recommender systems *Proc. of Conf. on Recommender systems (RecSys)* 2013;443-446.

[4] **Dooms S**. Improving IMDb movie recommendations with interactive settings and filters *Proc. of Conf. on Recommender systems (RecSys)* 2014.

International Workshops

[1] **Dooms S**, De Pessemier T, Martens L. A file-based approach for recommender systems in high-performance computing environments *Proc. of Conf. on Database and Expert Systems Applications (DEXA) - Workshop on Recommender Systems meet Databases (RSmeetDB)* 2011;529-533.

[2] **Dooms S**, De Pessemier T, Martens L. A user-centric evaluation of recommender algorithms for an event recommendation system *Proc. of Conf. on Recommender Systems (RecSys) - Workshop on User-Centric Evaluation of Recommender Systems and Their Interfaces-2 (UCERSTI)* 2011;67-73.

[3] **Dooms S**, De Pessemier T, Martens L. MovieTweetings: a movie rating dataset collected from twitter *Proc. of Conf. on Recommender systems (RecSys) - Workshop on Crowdsourcing and Human Computation for Recommender Systems (CrowdRec)* 2013.

[4] **Dooms S**, De Pessemier T, Martens L. Mining cross-domain rating datasets from structured data on twitter *Proc. of Conf. on World wide web (WWW) - Workshop on Modeling Social Media (MSM)* 2014.

Demos

[1] **Dooms S**, De Pessemier T, Martens L. Demonstrating contextual group recommendations for media in a home environment *Proc. of Dutch-Belgian Information Retrieval Workshop (DIR)* 2012;83-84.

Newsletters

[1] **Dooms S**, Martens L. Harvesting movie ratings from structured data in social media *SIGWEB Newsletter* 2014.

Chapter 1

Introduction

1.1 About the Recommender Systems Domain

It all starts with the *information overload* problem. Since the Internet became an integrated aspect of daily life, people are overwhelmed with information. Thousands of news articles, Facebook updates, tweets, emails and many many more scream for attention every day. While our *attention time* remains the same (i.e., limited to the number of awake hours per day), the amount of information proliferated through our communication networks expands at exponential rates. This situation inevitably leads to information overload and thus forces people to decide what deserves their attention and what not. But how can such decisions be made thoroughly? Recent statistics e.g., revealed that 100 hours of YouTube videos are uploaded every minute[1], every day 58 million tweets are posted on Twitter[2] and 55 million status updates are made on Facebook[3]. With such dazzling numbers in mind, how can people be expected to make informed decisions about what content to consume and what to ignore? The short answer is: they can not. At least not manually, there is simply not enough time.

People can however be assisted in the browsing process, by algorithms that automatically filter out interesting content based on learned user preferences; this is where recommender systems come in. While information retrieval (IR) systems typically guide users to the content (e.g., web search engines), recommender systems constitute a more modern

[1]`https://www.youtube.com/yt/press/statistics.html`
[2]`http://www.statisticbrain.com/twitter-statistics`
[3]`http://blog.kissmetrics.com/facebook-statistics`

paradigm of bringing the content to the users, often proactively. Nowadays, many of the choices we make are actually governed by recommender systems that in some way influence our decision making process. Sometimes recommendations are provided very openly such as the Twitter '*Who to follow*' feature or the YouTube '*Recommended videos*' section, but more often recommendations are integrated in more subtle ways such as personalized advertisements or the ranking of friend status updates on Facebook.

The first occurrences of recommender systems in scientific research literature are commonly accepted to be the Tapestry system [1] in 1992 and the GroupLens system [2] in 1994. Before that, research on information overload typically focused more on information filtering and often entailed semi-autonomous (i.e., human involvement) approaches to filter content. Both the Tapestry system (mail recommendation) and the GroupLens system (news recommendation) introduced the concept of *collaborative filtering* in the recommender systems domain. In its literal sense, the term refers to the collaborative effort of people helping each other to perform filtering of content e.g., moderated newsgroups where moderators manually filter content for other users. The term is now mainly used to indicate recommendation approaches where feedback information of multiple users is used to guide the recommendation process. Such approaches are based on the idea that users who had similar tastes in the past (e.g., their ratings were similar) are likely to have similar tastes in the future. Therefore, to find interesting items we can find similar users and recommend what they have liked.

The concept of collaborative filtering became popular and was soon followed by many other recommendation strategies such as content-based filtering [3], knowledge-based filtering [4], hybrid techniques and many variants of all of these. As the added value of recommender systems became more and more apparent, they were introduced in a wide number of divergent domains including restaurants [5], food [4], recipes [6], books [7], comic books [8], music [9], travel [10], tourism [11], perfume [12], movies [13], cultural events [14], conferences [15], research papers [16], crime suspects [17], jobs [18], video clips [19], tv shows [20], network data [21], news [22] and e-commerce [23].

1.1.1 The Goal of a Recommender System

Recommender systems link items and users in vast content collections. But what is their actual goal? While the first goal that comes to mind is often ‘to make users happy’, there are many other goals that a recommender system may try to achieve. Alan et al. [24] suggested to evaluate recommender systems in 3 dimensions with each dimension having different goals. They proposed a user perspective, a business perspective and a perspective focusing on technical constraints.

From the perspective of the user, recommender systems should strive for overall user satisfaction, which comes from reducing their information overload problem by finding interesting items and facilitating their catalog browsing experience. **For a business** however, the overall main goal is to make money. Businesses that deploy a recommender system expect to increase sales in some way or another. A good recommender system may point users to more interesting items to buy or stimulate customer retention in the long run, both of which result in increased sales and thus revenue. Less ethical approaches – sometimes referred to as *evil* recommender systems – might design the recommender system in such a way that mostly expensive items are recommended, or e.g., only items with high profit margins. In the extreme case recommender systems could even be applied to artificially increase prices for products that the system is almost certain a user will buy e.g., batteries for electronics, power cables, etc. **The technical constraints** linked with recommender systems are not really goals per se, but should be more regarded as minimum requirements for recommender systems to be successful. Examples of such technical constraints are scalability, robustness, reactivity and so on.

While recommender systems can be used for good or evil, in this work, we adopt the formal Google corporate motto of “Don’t be evil” and primarily focus on the user perspective of recommender systems, with the main goal of maximizing user satisfaction in the long run. This goal leads us to some new questions. What is best for a user? And, what are good recommendations?

In the past, recommendation quality was almost uniquely defined in terms of *recommendation accuracy* i.e., how accurate the system is able to predict interesting items. Typically two recommendation scenarios can be defined, *rating prediction* and *item prediction*. In the case of rating prediction, a recommender system is tasked with predicting how interesting a given item is for a given user, while the item prediction task entails generating a list of interesting items. The accuracy of rating prediction based systems is usually expressed in terms of metrics as Root Mean Square Error (RMSE) or Mean Absolute Error (MAE). Such metrics express the error between the predicted rating and actual user rating usually averaged over all items and users. The item prediction use case resembles that of typical IR tasks, where for a given search query a list of most interesting search results must be presented. The accuracy of such recommendation scenarios is therefore often expressed in terms of *precision* and *recall* [25], which are evaluation metrics borrowed from the IR domain.

The problem for recommender systems that blindly optimize for recommendation accuracy is the *limit of noticeable difference*. Recommender systems research has focused a lot on developing new recommendation algorithms that improve overall recommendation accuracy. Very often the reported improvement over existing state-of-the-art algorithms is only a few percent (or even less). Having statistically significant results however, does not imply that users are actually capable of noticing the difference. Above a certain threshold, users might be insensitive to changes in the recommendation accuracy. Users don't think in terms of RMSE or percentages, they rather intuitively evaluate the quality of results, which sometimes makes comparing multiple results difficult. We illustrate this in Fig. 1.1, where we present movie recommendation results from three different recommendation strategies. Each row represents the top-5 recommended movies. In mathematical terms, the difference of these results is easily calculated, but actual people will have a harder time comparing the accuracy of these movie recommendation lists. While the objectively calculated accuracy values may be significantly different, for actual people these lists may be appreciated equally, or vice versa differently perceived lists may actually result in the same calculated accuracy value. Thus the usefulness of optimizing recommendation accuracy is inherently limited by the user perception of changes in the recommendation list.

Another problem linked with optimizing recommendation accuracy is that the approach focuses primarily on user ratings. In the movie domain e.g., recommender systems try to predict which movies would obtain the highest user ratings for a given user and then these movies are recommended. When typical user behavior (in the movie domain) however is analyzed, data often shows that users actually tend to watch more 3-star rated movies than 5-star rated movies[4]. So again, taking only recommendation accuracy – in terms of being able to predict high rated items – into account may limit the improvement potential of recommendation

[4]As noted by Guy Shani in his tutorial presentation at the ACM RecSys conference in Barcelona 2010 [26]

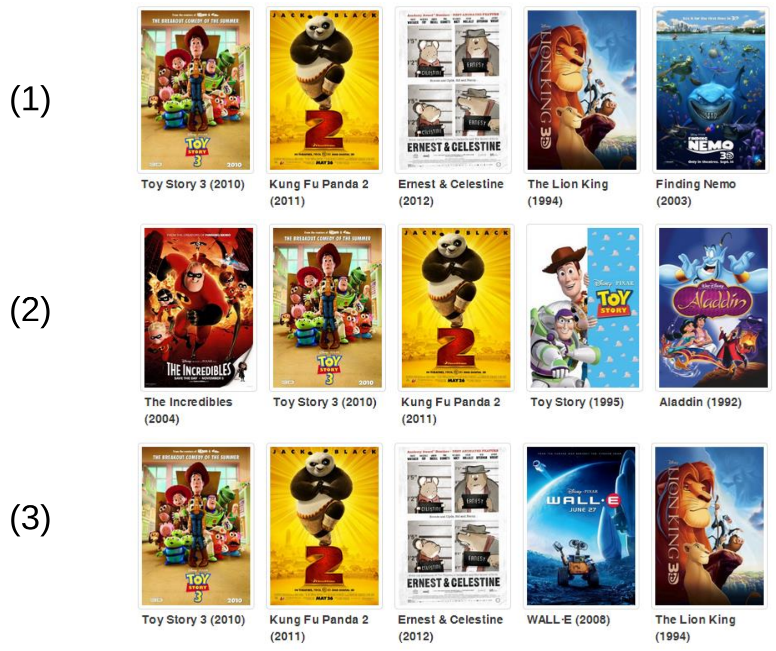


Figure 1.1: Movie recommendation results from three different recommendation strategies. Each row presents the top-5 recommended movies. While it is easy to calculate the accuracy of these results, people may actually be insensitive to the (small) difference.

quality.

In [27], Shani et al. therefore suggests many additional evaluation metrics that may be taken into account or be important to users of recommender systems. These metrics include novelty (i.e., how new are the recommended items to the user?), serendipity (i.e., how surprising?), diversity (i.e., how similar?) and many others. While research has shown that recommendation accuracy is positively correlated with user satisfaction, recent research has also shown other factors such as e.g., transparency and trust to be involved and therefore important in regulating user satisfaction. Different users may furthermore have different expectations e.g., some users may find obvious recommendations[5] annoying, while for others it may inspire trust of the recommender system.

In conclusion we note that there is no *silver bullet* in the recommender

[5]An example of an obvious recommendation in the movie domain is the movie Avatar (2009), which almost everyone likes but is unlikely to be an original suggestion.

systems domain. No single approach will be best for every user in every situation. The 'best choice' recommendation algorithm for a given situation will depend on the user, application domain, definition of *best* and most of all on the goal of the recommender system.

1.1.2 Rising Trends

From Offline Evaluation to Online, User-centric Experiments

As previously mentioned, in the early days of the recommender systems domain, evaluating recommendation quality focused mostly on expressing recommendation accuracy in terms of offline calculable metrics. Researchers devised new recommendation algorithms and reported their accuracy improvement towards baseline approaches. Oddly enough, actual users were rarely involved in the process. Public rating datasets (containing historical user feedback data) were split in training and test datasets, and then used to determine how accurate the algorithms were able to predict the ratings in the test dataset if given the training dataset as input.

In the last decade, more researchers from divergent fields of study e.g., psychology and Human-Computer Interaction (HCI) have joined the recommender systems domain. Increasing multi-disciplinary involvement caused the domain to shift its heavy algorithm-oriented focus to a more user-central focus. The importance of user studies gained traction and user involvement has now become an indispensable component of the recommendation quality evaluation process. As Shani et al. [27] suggests, offline evaluation (based on training and test datasets) is easy, cheap and fast and thus may be used early on in the algorithm design process for testing purposes or e.g., rapidly benchmarking multiple strategies. It is however through online and user-centric experiments that true user satisfaction can be evaluated by measuring user interaction behavior or explicitly asking users their opinion (i.e., questionnaires) [28]. When real users are involved (instead of only their historically captured feedback data), the actual *influence* of the recommendation process on user behavior can be measured. The most interesting recommendations are those that a user would not have found without the help of the recommender system. It is this aspect of user influence that differentiates the recommender systems domain from the *machine learning* domain. The latter tries to as close as possible predict future data based on historic data, while the former uses historic data to influence future behavioral data in a more subjective way.

Evolution of Input Data

Another evolving aspect of the recommender systems domain is the input data that is used to feed recommendation algorithms. Originally, user ratings were the sole source of input for recommendation processes. User preferences expressed as numbers in a confined interval (e.g., 1-5) went in, recommendations came out. Ratings however, are hard to collect since they require explicit user interaction and may even introduce noise because of user inconsistency [29, 30]. Collecting implicit feedback in the form of user interaction and behavioral data is easier and requires no additional user interaction, and thus a considerable amount of research has focused on how implicit feedback (additionally) can be processed into valuable input data for recommender systems (e.g. [31–33]).

In general, recommender systems will always keep adapting to changing availability of input data. If new technology and trends in society cause novel interesting data to become available, recommendation algorithms will be created that exploit the data. Two examples of such data-induced recommendation trends are social and context-aware recommendation algorithms. With the advent of social networks and in particular Facebook's *Open Graph* initiative, an abundance of social network data became publicly available. Social network data provides a rich source of user preference information (e.g., user profiles, likes, etc.) along with inter-user relationships (e.g., friends). Such data has been integrated in many so-called *social* recommendation algorithms that e.g., infer and integrate trust relationships in the recommendation process.

Context-aware recommendation algorithms involve contextual data in their recommendation process. Context data such as the time of day, current weather, user location, user mood, etc. can be very useful features since users might have different preferences depending on their context. People may e.g., listen to different music or watch different movies depending on whether they are alone or in group. While such information used to be hard to collect, nowadays almost everyone carries around devices such as smartphones and smartwatches that are capable of advanced contextual data logging. Even modern web browsers are now collecting contextual data such as location information if users allow it. Smartphones furthermore sprouted the trend of *mobile* recommender systems, which usually involve light-weight algorithms adapted to the limited computational power of mobile devices.

A Darker Image of Recommender Systems

Lastly, a trend we would like to note is the evolving *image* people have about recommender systems. While people happily install restaurant recommendation *apps* on their smartphones or receive travel suggestions in their mail, usually they are less amused when they learn that their browsing behavior is used for targeted advertising. Privacy issues have always surrounded recommender systems, but lately people are becoming more aware of their online activities and resulting *digital footprint*. Initiatives as private browsing and e.g., the 'Do not track' initiative[6] are gaining momentum. Usually privacy issues can be limited if users know exactly what is being logged and have the feeling that they get something in return e.g., promotions, recommendations, improved customer experience. Additionally they could be given the option to *opt-out* or even better, to explicitly *opt-in* to benefit from recommendation services.

Another blame on the image of recommender systems is the idea of filter bubbles. In 2011, Eli Pariser caused a stir in the recommender systems domain when he introduced the concept of the *Filter Bubble*[7]. He claimed that, because of personalization, more and more users of online platforms will be trapped in their own separate, filtered *bubbles* of information. Such a bubble would virtually surround users with content tailored to their interests and therefore at the same time also keep them from thinking outside the 'bubble'. They are no longer able to learn new things, evolve and change interests. A panel in the ACM RecSys 2011 conference[8] discussed the topic, and the take-away message was that personalization is fine as long as users can be given a certain amount of control (e.g., choose to disable personalized results) and recommender systems can be made as transparent as possible (e.g., explain the origin of a recommendation).

To summarize, we note that although the domain of recommender systems is reasonably young, it has matured and evolved considerably over the last few years (and continues to do so) to better align with the ever changing trends and needs of our modern society.

[6]`http://en.wikipedia.org/wiki/Do_Not_Track`
[7]`http://www.thefilterbubble.com/ted-talk`
[8]`http://recsys.acm.org/recsys11/recsys-2011-panel`

1.2 Recommendation Algorithm Overload

As previously discussed, for almost any type of content a recommendation algorithm has been developed. Some algorithms focus on collaborative data, some on content data, some on social or contextual data. In the last few years there has even been a rise in software libraries such as MyMediaLite [34], LensKit [35], Mahout[9], LibRec[10], Python-recsys[11], Duine[12] and Lucene[13], that offer out-of-the-box implementations of recommendation algorithms all designed to tackle the information overload problem. For an overview of even more recommendation frameworks used in research and production systems we refer to [36]. This overabundance of recommendation algorithms introduces a new problem which we refer to as the '**recommendation algorithm overload problem**'. Given an information system, context and goal, how can we decide what recommendation algorithm would be best?

While recommendation algorithms used to be competing against each other, nowadays it has been generally accepted that every algorithm has its own focus, optimal use cases, advantages and disadvantages. It seems logical to combine multiple algorithms together in so-called *hybrid recommenders*, to overcome their individual drawbacks [37–39]. Although hybrid recommenders are well-accepted and have shown their merits, the procedure of actually building and tweaking a hybrid recommender system is still a tedious and time-consuming process. Therefore most hybrid recommender systems are integrating only a few algorithms and are often configured in a static way.

What if we could have a hybrid recommender system that integrated all existing recommendation algorithms and dynamically decided which (combination of) recommendation algorithms to apply for a given context or scenario? Both users and industry would benefit from the improved user experience that may result from such a self-learning hybrid recommendation platform.

[9]`https://mahout.apache.org`
[10]`https://github.com/guoguibing/librec`
[11]`https://github.com/ocelma/python-recsys`
[12]`http://www.duineframework.org`
[13]`http://lucene.apache.org`

1.3 The Research Challenge

In this work we aim to investigate approaches and strategies that enable a future-proof hybrid recommendation platform that can seamlessly integrate existing recommendation algorithms and dynamically combine them into a best-fit hybrid recommender. While most mainstream recommendation algorithms start from the notion that users in a system behave similarly [40], recent research is turning more towards the idea that every user is unique and (combinations of) different algorithms may be best for different users [39, 41, 42]. Therefore we want a recommendation strategy to be personalized in the sense that it may dynamically adapt its hybrid approach for different users. In other words, we want to investigate how to ***dynamically generate personalized hybrid recommender systems***, as reflected by the title of this work.

We now list a few general research questions that we aim to answer in this work; more specific research questions are defined at the beginning of each chapter.

- Do (all) users benefit from a personalized hybrid recommender system?
- How can a hybrid recommender system be configured automatically?
- How can such a system be evaluated?
- Can a complex hybrid recommender system be used in real-time environments?

1.4 The Pieces of the Puzzle

Designing self-learning hybrid recommender systems is a complex task which we can divide into smaller and more manageable subproblems. The chapters in this work each tackle such an individual subproblem and are structured as follows.

In Chapter 2, we illustrate how all recommender systems research starts with input data. We elaborate on different types of user feedback and discuss the shortcomings of currently available public rating datasets. We experiment with several use cases and ultimately create our own rating dataset which we also make available to the general public.

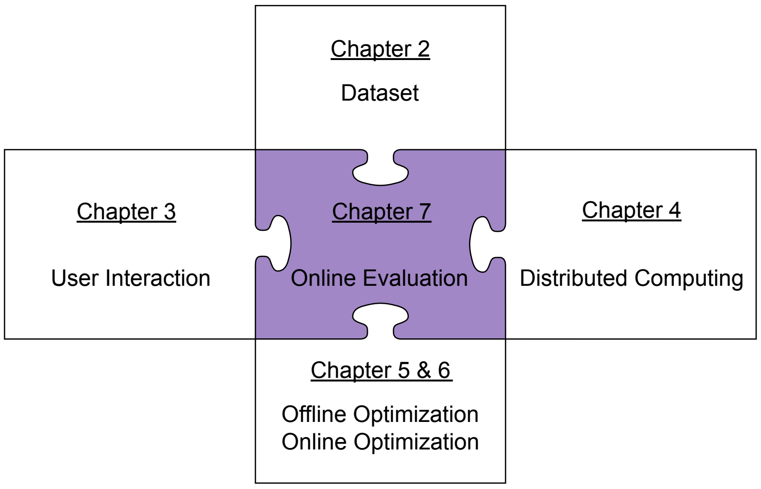


Figure 1.2: Illustration of how results of all chapters are used in our final chapter focusing on online evaluation.

Chapter 3 underlines the importance of user interfaces and their resulting human-recommender interaction processes. The chapter focuses both on feedback mechanisms and how recommendations can be presented to users while avoiding filter bubbles and introducing a sense of control and system transparency in a hybrid recommendation context.

Since hybrid recommendation calculations can be computationally very intense, we dedicate Chapter 4 to a high-performance viewpoint of the recommendation process. We investigate how recommendation algorithms can be mapped on to a high-performance computing infrastructure and benefit from distributed and parallel deployment.

These first 4 chapters provide the basic research on which the rest of this work is then founded. In addition to the recommendation algorithms themselves, a recommender system needs input data, a user interface and a decent computation infrastructure to support its recommendation strategies and properly expose its features to users. The following two chapters focus on the self-learning aspect of our research goal.

In Chapter 5, we consider self-optimizing hybrid recommender systems from an offline perspective. Using fixed datasets, we experiment with strategies that allow user-specific optimization of hybrid parameters in the recommendation process. In particular, we present results for hybrid switching and weighted hybridization scenarios.

Chapter 6 then continues with the obtained offline optimization approach and tries to get the recommender system out of the lab by assessing and improving its ability towards meeting real-world requirements in an online recommendation scenario. We evaluate our self-learning hybrid strategy in multiple dimensions both from a system perspective (e.g., scalability features) and a user perspective (e.g., transparency and control options).

In Chapter 7, we ultimately combine all the research from previous chapters and expose our self-learning hybrid recommender system to actual users in an online evaluation experiment. Instead of a typical lab-based experiment where test users are asked to use the system in a controlled environment, we opt for a true out-of-lab experience by making the system publicly available and attracting real users. We take our offline self-learning optimization system (Chapter 5), adapt it to run in an online environment (Chapter 6), deploy it on a distributed and parallel hardware infrastructure (Chapter 4), feed it our collected dataset (Chapter 2) and apply our user interaction experience (Chapter 3) in an easy-to-use and intuitive user interface. Fig. 1.2 illustrates how all the pieces (i.e., chapters) of the puzzle fit together in Chapter 7 and support both the validation and foundation of the research performed in this work.

Chapter 2

User Feedback Collection

2.1 Introduction

Recommender systems need input data to drive their decision making process and generate recommendations for users. The (perceived) quality of the recommendations does not solely rely on the recommendation algorithms, but also greatly depends on the provided input data. Without input, a recommender is just an engine without fuel and so the collection of input data is of paramount importance to any recommender system.

User input, often referred to as *feedback*, can be collected in various ways. Yu et al. [20] defines three categories: explicit input, explicit feedback and implicit feedback. The strategy of *explicit input* is to present the user with a list of questions (e.g., at registration). The answers can then be used to build a preliminary profile of the user containing general preference information. Collecting explicit input whenever a new user registers to the system, can help to alleviate the *cold-start* problem [37] (due to which users or items with too few feedback can not be recommended). *Explicit feedback* mostly translates to asking users to rate an item they have just consumed (or downloaded, viewed, purchased, etc.). Both explicit input as explicit feedback require the user to actively participate in the feedback process. *Implicit feedback* on the other hand collects its information in the background by means of logging data or monitoring user behavior. In the case of a video aggregation site, implicit feedback could for example monitor the duration a user watched a video.

Often recommender systems will combine forms of implicit feedback with explicit feedback by quantifying the implicit feedback to numeric values in the same range as the explicit feedback (i.e., ratings). Combining different forms of feedback has been shown to improve recommendation quality [33, 43] but the specific transformation method will depend on the item domain and involved use case.

In this chapter we discuss the current state of public rating datasets and their shortcomings towards supporting our research. We then outline an approach for user feedback collection for a specific use case (a cultural events website) and finally introduce a new rating dataset and paradigm for collecting both implicit and explicit user feedback from recent online rating sources.

Research Questions

- How useful are public datasets for (our) research?
- What types of feedback data can be collected?
- How can we collect relevant input data?

2.2 Public Rating Datasets

Ratings are usually considered private user data and therefore most online platforms do not make them publicly available. While academic researchers work on innovative recommendation algorithms and often lack user data for testing, industry has data and users, but needs the algorithms to improve their services. Because input data is difficult to collect without having access to an online platform with an active user base, most research on recommender systems relies on publicly available datasets i.e., rating datasets made public by companies or online platforms for the benefit of research.

Two of the most popular datasets are the MovieLens [44–46] and the Netflix [47–49] datasets, both focusing on the movie domain. The first MovieLens dataset (ML 100K) was released at the end of the 90s and integrated 100,000 ratings originating from users of the MovieLens system covering a seven-month period (September 19th, 1997 through April 22nd, 1998). The dataset quickly became very popular because of its simplicity (i.e., only explicit ratings and very basic item and user data were available) and the lack of other datasets at that time. Later in 2006, the Netflix dataset originated from the well-known Netflix prize[1],

[1]`http://www.netflixprize.com`

where 1 million dollar was promised to the first team able to improve the in-house Netflix recommendation algorithm by more than 10%. In the last few years, dozens of other datasets have become available focusing sometimes on very divergent item domains (e.g., jokes, like the Jester dataset[2]) or specific metadata availability (e.g., contextual information, like the datasets produced in the different editions of the context-aware movie recommendation – or *CAMRA* – challenge[3]).

In recommender systems literature, public datasets are used for a wide variety of reasons. Often they are used in comparative experiments where a new algorithm is compared and benchmarked against previously published results (e.g. [50]). In these situations the specific properties of the dataset do not really matter as long as the results on the dataset are generalizable. In other situations, however, datasets are used to feed a recommender system that is then deployed in user-centric experiments. The datasets in these situations boost the recommendation results by complementing the (sometimes very limited) rating data of the users engaged in the experiment. While the results of the comparative experiments are processed in a numerical way (e.g., ranking, similarity, accuracy, precision calculations), in user-centric experiments real users are actually looking at the recommendation results while providing their feedback. The effect of this is that experimental results can easily be influenced by properties of the integrated dataset. If current-day users were to be shown a recommender system that has been trained on e.g., the MovieLens 100K dataset, the resulting recommendations would necessarily be old movies – released between 1922 and 1998 – which may negatively affect the general user experience.

Currently, much scientific literature uses old datasets (like MovieLens) even for user-centric evaluation experiments. The MovieLens 100K dataset is often used to bootstrap a recommender system and bypass cold-start issues. In [51] an application called 'MovieQuiz' was developed using the MovieLens 100K dataset as *seed* data for the evaluation of a conversational collaborative filtering approach. Real users were asked to interact with the system, and evaluate the resulting recommendations. Another very recent example is the work of Said et al. [52] where a user study was performed to evaluate their K-furthest neighbor collaborative filtering recommender algorithm. In their study, they relied on movie data from the MovieLens 10M dataset, which is more recent than ML

[2]`http://shadow.ieor.berkeley.edu/humor`

[3]`http://2012.recsyschallenge.com/tracks/camra/`

100K (i.e., most recent movie is from 2008), but still lacks current-day popular and relevant movies.

Aside from being old and static, current-day public rating datasets are also often filtered to only contain users with a minimum number of ratings (e.g., 20 ratings for MovieLens). Because of this filtering, a systematic bias is introduced which may prevent experimental results to be generalizable to real-life scenarios [27]. So while public datasets like MovieLens and Netflix may still be useful for offline evaluation, online experiments with actual users may fail because of the lack of recent movies in the dataset and experimental generalizability.

2.3 A Cultural Events Website: Use Case Study

To learn more about collecting user input in a realistic context, we collaborated with a popular (>10,000 visitors per day) cultural events website[4]. This website contains details of cultural events taking place in Belgium. With a large user base of over 13,000 registered users and a collection of more than 20,000 events, it serves as a more than appropriate platform for our user feedback experiments.

For every event on the website there exists a detailed information page. This page offers basic event information as well as some *action links* to e.g., mail to a friend, print information, show more information, etc. There is also an option to rate events (i.e., indicate that a user *likes* an event) which can be categorized as explicit feedback. Monitoring how a user interacts with interesting action links, could on the other hand be considered a form of implicit feedback. Even the action itself that a user browsed to a certain event page could be used as implicit feedback towards the event.

To investigate how explicit and implicit feedback relate to one another, a user feedback experiment was set up on the cultural website. For a continuous period of 7 months, every user interaction with the website in the form of ratings, browsing to an event or clicking on preference indicating action links was registered and logged. The following list discusses the action links (on an event detail page) we considered to be indications of the user interest in the event.

- **More details**: Shows detailed information about the event (instead of just a short description).

[4]`http://www.uitinvlaanderen.be` managed by CultuurNet Vlaanderen

- **More dates**: Shows more dates on which the event takes place.
- **Mail**: Allows to mail a link of the event to a friend.
- **Print**: Prints the event information.
- **Show map**: Shows a map detailing the location of the event.
- **Train directions**: Provides train direction to get to the location of the event.
- **Bus directions**: Provides bus directions to get to the location of the event.

The website technology was based on the popular open-source content management framework *Drupal*[5] and so a Drupal module was implemented that registered the relevant feedback information. Table 2.1 shows the unique users, items and total number of actions that were logged over a 7 month period.

Feedback type	#users	#items	#entries
Explicit: ratings	6,532	5,053	7,721
Implicit: clicks	160,915	49,458	321,077
Implicit: views	572,390	78,538	1,392,402

Table 2.1: A comparison of the number of feedback actions collected for explicit and implicit feedback on a cultural events website over a 7 months period.

The table illustrates an important distinction between explicit feedback and implicit feedback: implicit feedback is more abundant than explicit feedback. Because implicit feedback requires no additional effort from the user, it is much easier to collect in large numbers. When we represent the total number of entries as a pie chart (Fig. 2.1), this difference becomes even more apparent.

The most abundantly available are the views, which is to be expected since in order to rate events or click on action links, a user first has to browse (i.e., *view* action) to the event information page. The number of collected clicks is 5 times less abundant, so on average for every 5 times an event was browsed to, only one user interaction in the form of a click on an action link was detected. Finally, most surprising is the low number of explicit feedback that was collected in comparison with the implicit

[5]`https://drupal.org`

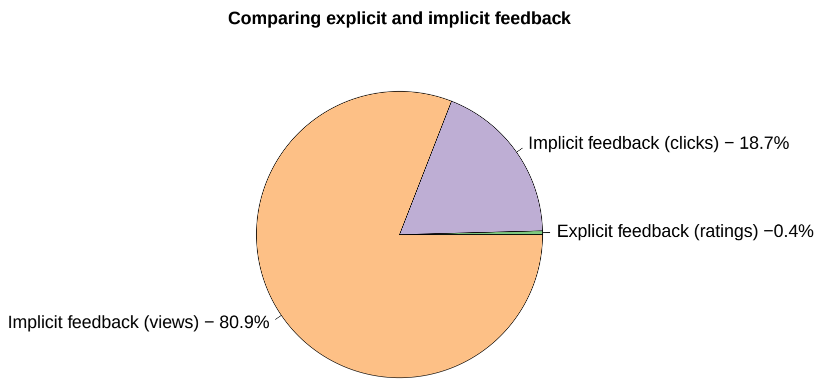


Figure 2.1: A comparison of the total number of feedback actions collected for explicit and implicit feedback on a cultural events website over a 7 months period.

feedback. For every 200 times that an event information page was visited only once the event was rated. These measurements confirm what has been observed before in recommender systems literature [33]: although explicit ratings contain more inherent information about the preference of the user towards the rated item, implicit feedback is easier to come by. To combine and use both would be the most interesting, but requires the implicit feedback to be transformed into a format comparable with the explicit feedback (i.e., ratings). We illustrate such an approach in [53].

In Table 2.2 and Fig. 2.2 we report the collected number of clicks on the action links specifically.

Action Link	#clicks
More details	274,267
More dates	32,741
Show map	6,198
Print	2,522
Mail	2,169
Bus directions	1,616
Train directions	1,564

Table 2.2: The number of clicks collected during the experiment for each monitored action link.

The *'more details'* clicks account for more than 85% of the total col-

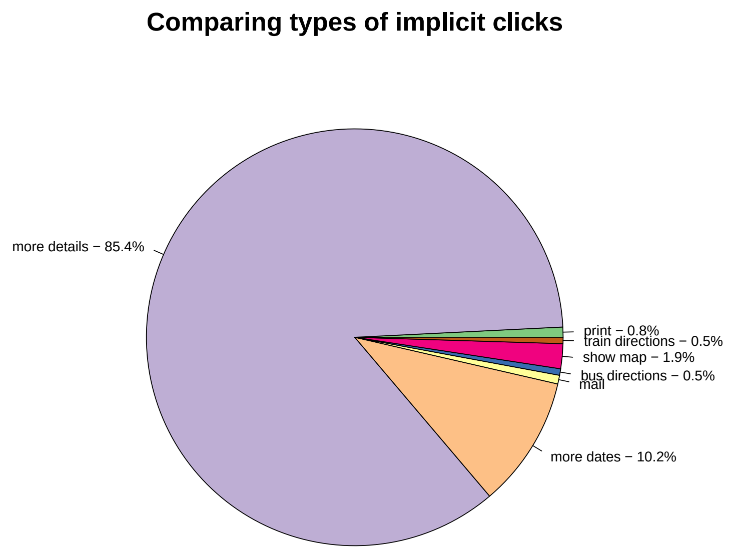
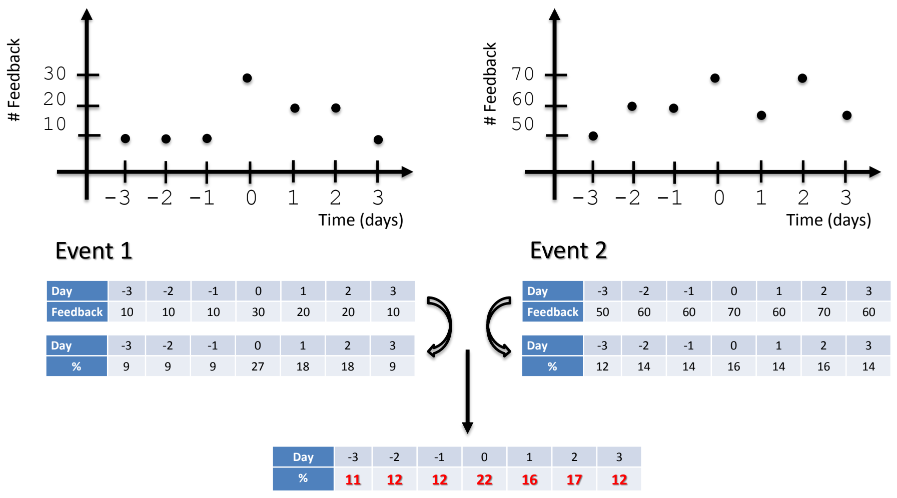


Figure 2.2: A comparison of the individual clicks on monitored action links on a cultural events website over a 7 months period.

lected number of clicks. The data shows a similar trend as for the implicit versus explicit feedback situation: the number of feedback differs greatly among the collected feedback types. While we collected many *'more details'* clicks, the far less popular *print* click may be more interesting to extract user preference information from. We could hypothesize that a user printing the event information is more likely to find the event interesting than a user merely clicking for more information. The same is true for the other action links e.g., the bus and train direction links. Users clicking those links, express a clear intend of attending the event which reflects positive interest more than just clicking the *'more details'* link. So also among the implicitly gathered feedback data we find the trend that more useful feedback types are less prominently available.

While the logging of feedback data on the cultural events website resulted in a user preferences dataset, the dataset may in fact not be very useful for general application domains. Items are *events* in this scenario and events have some very specific properties. While items in online e-commerce platforms are always available, events are transient. They take place at a certain time and place and after that, they are no longer relevant. This also impacts the collection of feedback since a user can only know for sure to like or dislike an event after having attended. We hypothesize therefore that feedback concerning events will be for the most part collected after the event has passed and can no longer be recommended to other users.

To verify the hypothesis that user feedback on events is time bound, we performed a time analysis on the collected data from our previous feedback experiment. For every event we analyzed its user feedback relative to the day of the occurrence of the event (e.g., the day of the festival). The number of feedback actions per event was expressed as percentage of the total number of feedback collected for the event to increase comparability over all events. The percentages per day could then be averaged over all events. Fig. 2.3 illustrates this time analysis procedure on two fictional events on a 7 day period.



Event 1

Day	-3	-2	-1	0	1	2	3
Feedback	10	10	10	30	20	20	10

Day	-3	-2	-1	0	1	2	3
%	9	9	9	27	18	18	9

Event 2

Day	-3	-2	-1	0	1	2	3
Feedback	50	60	60	70	60	70	60

Day	-3	-2	-1	0	1	2	3
%	12	14	14	16	14	16	14

Day	-3	-2	-1	0	1	2	3
%	**11**	**12**	**12**	**22**	**16**	**17**	**12**

Figure 2.3: The procedure for the time analysis of the collected feedback data relative to the day of the event illustrated for two events on a 7 day period. Results are rounded for simplification.

The time analysis results in a feedback pattern in function of the time relative to the day of the event and this for an *average* event in the dataset. Fig. 2.4 shows the result of the time analysis calculated for the individual feedback types: explicit ratings, implicit clicks and implicit views. A cumulative total was added to the plot to indicate the cumulative percentage of collected feedback for all types of feedback combined. The data in the plot has been limited to a time period of one year before and after the event, so the X-axis ranges from −365 to 365. The cumulative total in the plot stops at 94% (and not 100%) because 6% of the collected feedback data occurred outside that range.

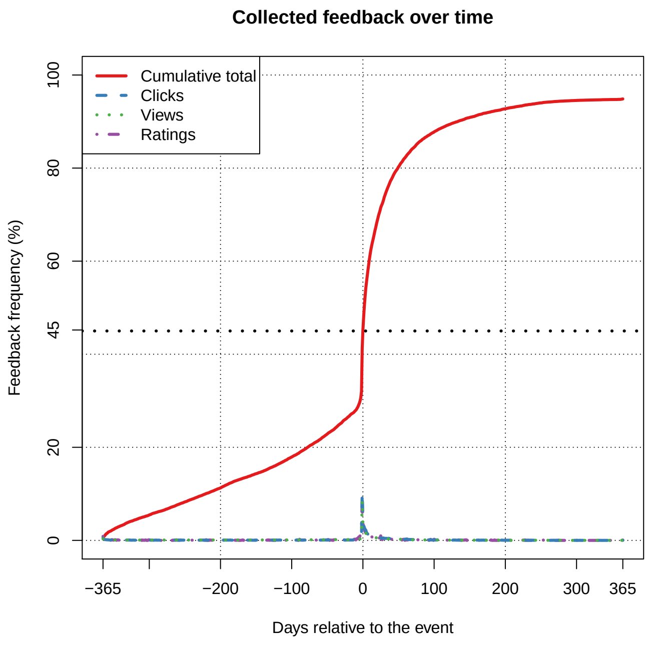


Figure 2.4: The result of a time analysis of collected feedback relative to the day of the event averaged over all events in the dataset. The pattern clearly shows how feedback is focused around the day of the event.

The time analysis figure confirms our hypothesis that the collected user feedback for events shows some time specific qualities that may prevent research conclusions from generalizing to other item domains. The time period in which events receive user feedback, is focused particularly around the date of the event (i.e., when an event is most popular). Moving farther away (in time) from the event causes the feedback rate to drop significantly until e.g., after a year close to no feedback is collected.

Item domains with non-transient items e.g., movie or music domains, might have higher feedback rates when an item is first introduced (e.g., new movie is available in Netflix) but in general, an item remains available to the user and will collect feedback over periods longer than a year before and after the release date.

In conclusion, we note that although the logging of the user feedback on the cultural events website did offer valuable insights regarding the availability of implicit versus explicit feedback, the collected feedback is very time focused which limits its usability to transient item domains. In the following sections we construct more general datasets, usable for recommender system experiments in non-transient item domains.

2.4 Movie Ratings from Twitter: The MovieTweetings Dataset

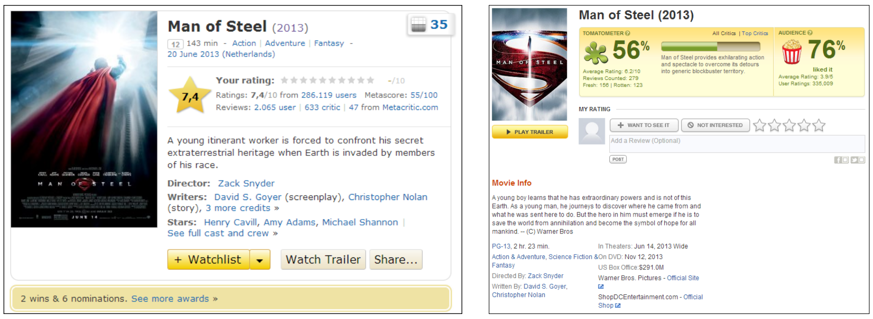
The typical MovieLens and Netflix movie rating datasets are inherently old (i.e., include old ratings from old movies) and static. A relevant movie rating dataset however should include relevant, current-day popular movies and be updated regularly to ensure the continued freshness of the ratings and recentness of the movies. For this purpose, the Internet serves as an interesting source of ratings ready to be extracted and collected. In the next sections we explain how rating information is abundantly available in various online sources and we show how it can be unambiguously extracted from IMDb users through the social media platform Twitter.

2.4.1 Unstructured Preference Information from Online Sources

Movie information is abundantly available on the Internet through a vast number of online services. Amongst the most popular ones are *IMDb*[6] and *Rotten Tomatoes*[7]. These websites aggregate and display extensive movie information such as director, cast, genre, etc. Fig. 2.5 shows the movie information pages of both services for the movie 'Man of Steel'. Aside from movie information, the figure also shows how both services are focused towards preference indicators. In both screenshots, clear cues of user opinions are visible in the form of aggregated star ratings, metascores, Oscar nominations and user reviews. While these indicators seem suitable user feedback for recommender systems, they lack general structure, which makes automated extraction difficult, apart from being aggregated metrics rather than user individual feedback, and so they are not useful to be exploited for personalization purposes.

[6]`http://www.imdb.com`

[7]`http://www.rottentomatoes.com`



Man of Steel (2013) 35

12 143 min - Action | Adventure | Fantasy - 20 June 2013 (Netherlands)

7,4 **Your rating:** -/10

Ratings: **7,4**/10 from 286.119 users Metascore: 55/100

Reviews: 2.065 user | 633 critic | 47 from Metacritic.com

A young itinerant worker is forced to confront his secret extraterrestrial heritage when Earth is invaded by members of his race.

Director: Zack Snyder

Writers: David S. Goyer (screenplay), Christopher Nolan (story), 3 more credits »

Stars: Henry Cavill, Amy Adams, Michael Shannon | See full cast and crew »

2 wins & 6 nominations. See more awards »

Man of Steel (2013)

TOMATOMETER All Critics | Top Critics

56%

Average Rating: 6.2/10
Reviews Counted: 279
Fresh: 156 | Rotten: 123

Man of Steel provides exhilarating action and spectacle to overcome its detours into generic blockbuster territory.

AUDIENCE

76%

liked it
Average Rating: 3.9/5
User Ratings: 335,009

MY RATING

Movie Info

A young boy learns that he has extraordinary powers and is not of this Earth. As a young man, he journeys to discover where he came from and what he was sent here to do. But the hero in him must emerge if he is to save the world from annihilation and become the symbol of hope for all mankind. -- (C) Warner Bros

PG-13, 2 hr. 23 min.
Action & Adventure, Science Fiction & Fantasy
Directed By: Zack Snyder
Written By: David S. Goyer, Christopher Nolan

In Theaters: Jun 14, 2013 Wide
On DVD: Nov 12, 2013
US Box Office: $291.0M
Warner Bros. Pictures - Official Site
ShopDCEntertainment.com - Official Shop

Figure 2.5: Screenshots of the detailed information pages that both IMDb (left) and Rotten Tomatoes (right) offer for any available movie (here: Man of Steel). Aside from movie information there are many indicators of user opinions as well.

Another source of movie (preference) information may be found on online social networks like Facebook[8] and Twitter[9] since such social networks are very often used to promote ideas and user opinions.

If we perform a Facebook search for our example movie 'Man of Steel', we arrive at an information page (much like the IMDb and Rotten Tomatoes websites) detailing some movie data but this time focused on the social aspect. Users can, among many other things, comment on the movie, invite friends to watch it, and click 'Like' if they liked it. Information about the number of people who like the movie can be retrieved, but again, only in terms of aggregated metrics instead of at an individual, user basis. Although Facebook is currently experimenting with star rating feedback systems[10], most of the pages only allow feedback in the form of a single positive 'Like'.

Twitter also entails some challenges for automatically mining user feedback. Specifically, the results for a search query on Twitter include all (most recent and popular) tweets that contain the search query i.e., in this context, the title of the movie. Because of this, the returned tweets are very ambiguous, since they can represent opinions about the movie, links to interesting posts, images, jokes, and very often not even involve the movie at all. Especially for movies with non-specific titles the latter is very common. Movie titles like 'Gravity' or 'Up' undoubtedly make it

[8]`https://www.facebook.com`
[9]`https://twitter.com`
[10]`http://techcrunch.com/2013/11/07/facebook-pages-star-ratings`

very difficult to filter and extract only the relevant preference information (i.e., user opinions).

So while the Internet offers a wide range of services that provide movie information and user opinions, the data is often too unstructured which makes it very hard to extract it automatically, or the data is only available in the form of aggregated numbers. Interestingly, however, while online services have their challenges in isolation, it is in combining them that we have found the key to mining structured and explicit movie ratings, as we show in the next section.

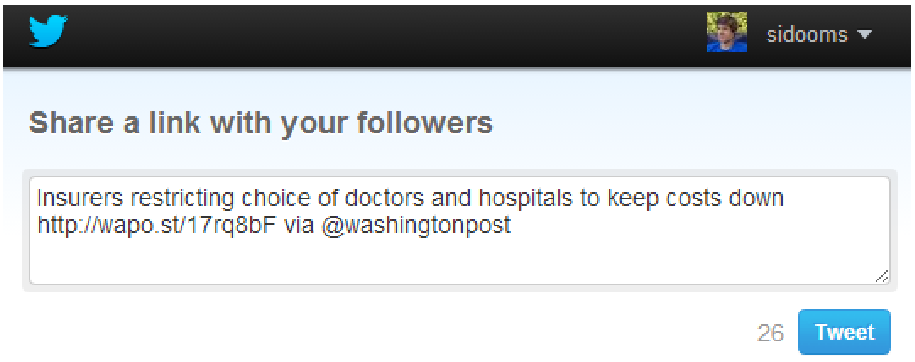
2.4.2 Structured IMDb Ratings from Twitter: The MovieTweetings Dataset

Rating datasets like MovieLens and Netflix are very popular, and the main reason for this is, among other things, their simplicity. They offer explicit 5-star rating values that a number of users provided on a number of movies. Such explicit movie rating data can directly be offered as input to a recommender system while implicit feedback data (e.g., comments, posts, views, etc.) first has to be processed and quantified [31, 32, 53]. Therefore, most of the time simple explicit rating datasets will be preferred over the unstructured preference information that can easily be found on available online services.

While searching for structured movie preference information, to our own surprise, we discovered a vast number of explicit movie ratings from IMDb being posted on Twitter by means of a *social sharing* feature. This notion of *social sharing* is increasingly becoming more popular on the Internet. Websites offer promote buttons that aid users in posting interesting content (i.e., often the page the user is currently browsing) directly to their social network. When such a button is clicked, a user may add some additional comments before finally submitting. The content provider usually already provides a suggestion (template) for what a user might post, for instance, the title of the page together with the URL and some reference to the social network account of the website itself (illustrated in Fig. 2.6).

We have found that one of the mobile apps from the IMDb platform[11], after rating a movie, offers a well-structured template to post on Twitter (Fig. 2.7). When a user rates a movie on the iOS mobile app, an option is available to ‘Share my rating’. When enabled, the user is taken to a

[11] `http://www.imdb.com/apps`



Insurers restricting choice of doctors and hospitals to keep costs down
http://wapo.st/17rq8bF via @washingtonpost

Figure 2.6: Illustration of the social sharing feature that is included in many websites (here: *washingtonpost.com*). By clicking a *share* button, the content provider provides a suggestion as to what the user might post to its social network (here: *twitter.com*).

screen that proposes to post the following text (for the movie 'Man of Steel'):

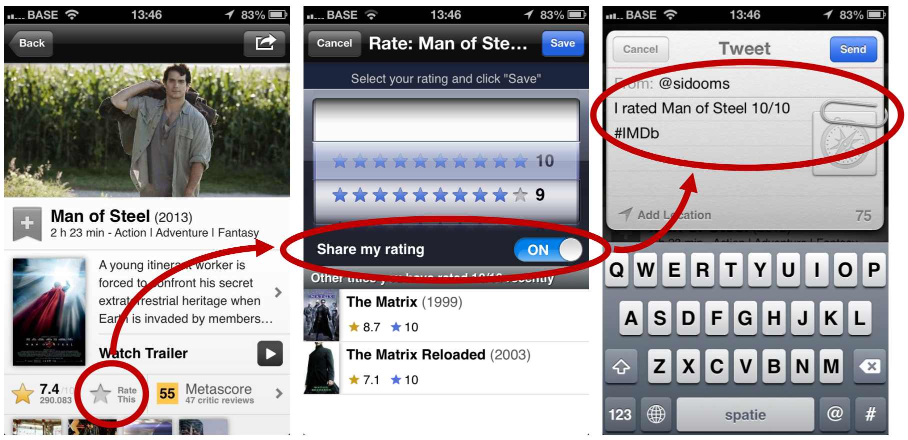
```
I rated Man of Steel 10/10 #IMDb
```

This pre-formatted tweet is well-structured and therefore apt for automated extraction. In the tweet we find the title of the movie, the rating and a website-specific hashtag. The hashtag allows for easy filtering tweets originating from IMDb. When the tweet is finally posted, a link to the IMDb page of the involved movie is inserted as follows:

```
I rated Man of Steel http://www.imdb.com/title/tt0770828
10/10 #IMDb
```

From this link, the IMDb id of the movie can be extracted which allows us to unambiguously identify the movie that is rated in the tweet (which is not always possible using only the movie title). While searching Twitter can lead to many ambiguous results, we can use the proposed fixed format of the tweet to our advantage. Instead of searching for the movie title, we use the query 'I rated #IMDb' and then apply string matching techniques to extract the relevant fields from the returned tweets. Specifically, we extract the following fields from the tweets:

- Twitter user id



A young itinerant worker is forced to confront his secret extraterrestrial heritage when Earth is invaded by members…

From: @sidooms

I rated Man of Steel 10/10 #IMDb

Figure 2.7: Screenshots from the iPhone mobile IMDb app illustrating the process of rating a movie and sharing it on Twitter.

- IMDb movie id
- Rating
- Timestamp

Furthermore, we also extract additional genre data from the IMDb page of the movie rated in the tweet. Such additional genre data can be exploited by content-based recommenders who would use movie attributes in the recommendation process [3]. Even more content attributes can be downloaded by extracting them directly from the corresponding IMDb page (whose URL can be reconstructed directly from the IMDb movie id by adding '`http://www.imdb.com/title/tt`'). Sometimes users tend to extend the default tweet (up to the limit of 140 characters imposed by Twitter), with their own opinions or additional comments, such as:

```
I rated Man of Steel http://www.imdb.com/title/tt0770828
10/10 #IMDb Great movie!
```

For our dataset, however, we do not integrate these additional comments which would require Natural Language Processing [54] and might introduce more noise than information. Instead, we focus on the available explicit feedback, i.e., the numerical rating. Starting March 7, 2013 we queried the Twitter search API on a daily basis and extracted ratings from relevant tweets into a new movie rating dataset called *Movie-*

Tweetings. The dataset itself consists of three files: *ratings.dat*, *users.dat* and *movies.dat*, which are formatted similarly as the popular MovieLens dataset to facilitate integration in existing implementations.

- **ratings.dat** contains the ratings as tuples, together with the user, movie and corresponding timestamp. The user id is an internal numerical identifier, while the movie id is the IMDb id. Ratings range from 1 to 10[12].
- **users.dat** provides a link between the internal user ids and the true Twitter id of the user, allowing for additional data enrichment.
- **movies.dat** lists the movies that were rated at least once, together with the movie year and genre data, from IMDb.

The dataset offers two repositories[13]: latest data and snapshots. Latest data always contains all the data in the dataset, including the most recently added data. This repository will therefore be subject to frequent updates. The snapshots, on the other hand, offer fixed portions of the dataset in various sizes (e.g., 10K, 100K, 150K) to stimulate and facilitate experimental reproducibility.

2.4.3 MovieTweetings Advantages

In summary, we list the advantages of the MovieTweetings dataset with respect to other existing public datasets.

- Realistic user behavior (unfiltered, raw data)
- Metadata easily expandable (linked to a well-known movie database)
- Real users (live online evaluation possible)
- Frequently updated (several snapshots available)

The MovieTweetings dataset is unfiltered and therefore a natural dataset. No users or items are excluded from the dataset for not having a sufficient amount of ratings. This enables the simulation of realistic user

[12]The dataset does include a few 0/10 ratings, but they are originating from users that manually changed the rating value to 0 in the template tweet.
[13]Available at `https://github.com/sidooms/MovieTweetings`.

behavior. A real-life recommender system will have to deal with non-active users and poorly rated items, and so the MovieTweetings dataset can offer a means to experiment with and simulate these scenarios.

The metadata contained in the dataset consists of movie genres (as available in the MovieLens dataset), but can be very easily supplemented with other movie information. Because the dataset links every movie to the unique IMDb identifier, additional metadata can be collected by using tools as the OMDb API[14] or by scraping the original IMDb website itself.

Because rating data is collected from publicly available information, the users contained in the MovieTweetings dataset did not need to be anonymized. So the user identifiers used in the dataset can be linked to the real Twitter users. Additional user data can easily be collected using the Twitter API and by analyzing online user behavior. Furthermore, the integration of real approachable users in the dataset offers an interesting user-base for researchers that may spark a new generation of low-effort online/live user-centric evaluation experiments.

The most important advantage and difference towards other datasets is the fact that the dataset is constantly updated. Instead of offering data from a fixed period in time, new ratings extracted from Twitter are added frequently. The MovieTweetings dataset will therefore always contain the most recent and popular movies, which makes it an interesting seed dataset for user-centric experiments.

2.5 Investigating Dataset Biases

The MovieTweetings dataset may be subject to biases introduced by the manner in which it is collected and in the sampling inherent to the Twitter API [55]. To understand the consequence of working with the MovieTweetings dataset, it is important to first understand these biases and study the extent of their impact. The ratings in the dataset come from users of the IMDb platform that rate movies using the mobile IMDb app for iOS and post the ratings (publicly) on Twitter. One obvious bias to consider here is the effect of the ratings being collected through Twitter and not directly from IMDb itself. Only data from Twitter users with a public profile that chose to share their IMDb rating is included. Maybe these users are sharing their high and low ratings, but consider the mid-

[14]`http://www.omdbapi.com`

range ratings – such as 6/10 – not interesting enough to be posted on their social network. Another more structural bias comes from the fact that the dataset focuses on the IMDb platform. This platform offers a broad selection of movie information, but not the movies themselves. Apart from the ability to watch the trailer of a movie, users can not watch movies on the IMDb site before rating them, something that is possible in a system like Netflix. Also the rating scale may affect the users [56], since the IMDb platform allows to rate movies using a 10-star feedback system, instead of the more common 5-star feedback system (like MovieLens). In this section, we address these issues by comparing the MovieTweetings dataset characteristics with other datasets and analyzing the biases introduced by Twitter and the IMDb platform.

2.5.1 Twitter Bias

To study the possible rating biases introduced by including only Twitter ratings, we would need to compare the ratings of the entire IMDb platform with the subset of the ratings extracted through Twitter. For this purpose a correlation study seems appropriate. The complete set of IMDb ratings (per user) is not available but the platform does provide the aggregated average movie scores. Unfortunately for us, IMDb does not disclose its exact averaging formula. On their website[15] they claim to apply various filters to eliminate the effect of fake ratings by individuals who are trying to distort the aggregated rating of a movie. Therefore in our correlation analysis we have to take at least a small amount of noise into account (i.e., perfect correlation will not be possible).

For every movie that was included in the MovieTweetings (100K snapshot) dataset, we used the OMDb API to obtain the aggregated average IMDb rating. We then compared this average value with the average of the MovieTweetings ratings for that movie in a correlation analysis. Fig. 2.8 (left) shows the resulting scatter plot, visualizing every movie with its aggregated IMDb rating (on the Y-axis) and the corresponding average MovieTweetings rating (on the X-axis). The resulting Pearson correlation value is 0.542, which indicates a positive correlation. When we inspect the scatter plot, we find this confirmed, in particular for the higher rating values (higher than 6). The figure however also shows a significant amount of noise, which we hypothesize originates from movies with a low number of ratings.

[15]`http://www.imdb.com/help/show_leaf?ratingsexplanation`

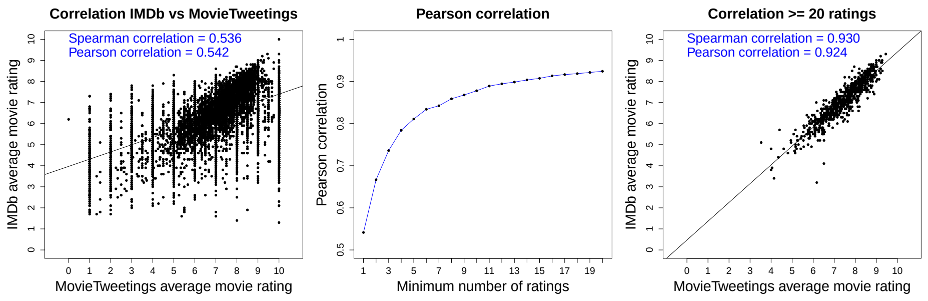


Figure 2.8: Plots illustrating the correlation of the aggregated IMDb rating and the averaged MovieTweetings rating per movie. The figure on the left shows the correlation for all movies in the MovieTweetings dataset. In the middle, the Pearson correlation values are shown for movie subsets having a minimum of 1-20 ratings. The scatter plot on the right illustrates the absence of noise for movies with a minimum of 20 ratings.

To experiment with the effect of the rating frequency of the movies, we repeated the correlation analysis for different subsets of the MovieTweetings movies: we illustrate in the middle plot of Fig. 2.8 the increasing Pearson correlation for the movie subsets having a minimum of 2, 3, ..., 20 ratings. As expected, the harder the constraint, the more the noise is reduced, which leads to a stronger linear correlation. Fig. 2.8 (right) plots the correlation values including only movies with a minimum of 20 ratings. Note that the correlation seems to converge to a value below the perfect correlation (i.e., of 1.0), this may be the result of not using the exact averaging formula of IMDb.

Complementary to this correlation analysis, we further compare the MovieTweetings dataset with the IMDb ratings by performing a popularity analysis for both datasets. A well-known IMDb popularity list is the top 250 movie list[16]. This list shows the best rated movies according to a Bayesian estimate defined in Equation 2.1:

$$weighted\ rating\ (WR) = \frac{v}{v+m} * R + \frac{m}{v+m} * C \tag{2.1}$$

where:

- R: mean rating for the movie

[16]Accessible at `http://www.imdb.com/chart/top?ref_=nv_ch_250_4`.

- v: number of ratings for the movie
- m: minimum ratings required to be listed in the Top 250 (currently 25,000)
- C: the mean rating across the whole report (currently 7.0)

So the overall popularity score takes into account the rating values, as well as the number of ratings itself to rank the movies. We adopted this as the definition of popularity and implemented the formula to rank the movies in the MovieTweetings dataset (m was set to 20). For the same set of movies we also calculated the IMDb popularity value using the IMDb average rating values. Having calculated both the IMDb popularity and the MovieTweetings popularity value, the two lists can be ranked according to each value and compared. For the comparison, we employ the Jaccard similarity index. A similar approach was used by Bellogín et al. in [57] to compare lists of popular artists for different cutoff values. Fig. 2.9 shows the Jaccard index for all movies with a minimum of 20 ratings. While the top 10 popular movies are almost identical, the two datasets tend to diverge when less popular movies are considered (with another small spike at around top 50) and slowly converge to a Jaccard value of 1 at the end.

Table 2.3 shows the list of top 10 movies for both MovieTweetings and IMDb and their resulting Jaccard index values. The lists are very similar except for some small changes in the ordering. Apart from the similarity amongst these top popular movies, data shows that more recent movies tend to obtain a higher overall popularity score in the MovieTweetings dataset. This is to be expected, considering how MovieTweetings data is collected from Twitter. Therefore the dataset contains more recent and currently popular movies. We expect however that with more data being collected regularly, the effect of the recentness of the movies will gradually decrease.

While the MovieTweetings dataset may favor recent movies, we hypothesize that, in general, similar trends can be noticed for IMDb. One of such trends is the type of movies that are popular i.e., the genre. A set of genres is available for each movie allowing for a new correlation analysis, this time comparing the similarity amongst genres of popular movies for both MovieTweetings and IMDb. For this experiment we focus on the top 250 popularity list. For the 250 most popular movies we summed up the number of times each genre occurred (that is, we compute the term frequency of each genre [58]), doing this for both datasets.

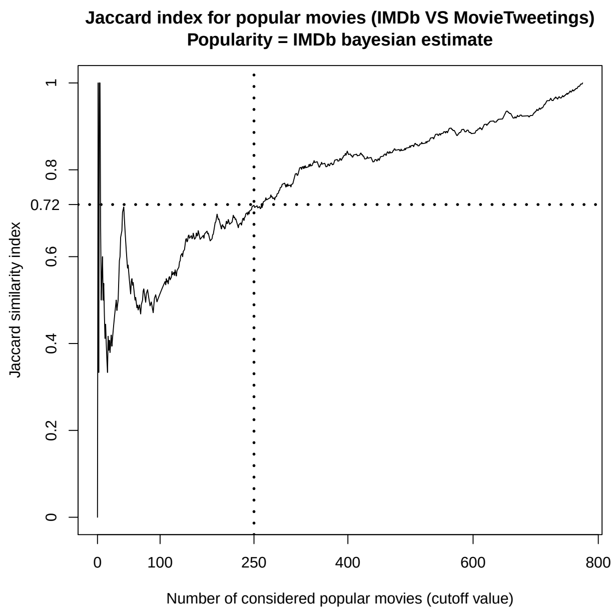


Figure 2.9: The Jaccard similarity index, indicating the similarity between the top popular IMDb and MovieTweetings movies, for different cutoff lengths. Only movies with a minimum of 20 ratings are taken into account.

Fig. 2.10 shows the correlation between the genre counts of the most popular MovieTweetings movies versus the ones of IMDb.

While the Jaccard index for the top 250 popular movies was 0.72 (see Fig. 2.9), Fig. 2.10 clearly shows how highly correlated the movie genres are for both lists (Spearman correlation is 0.99). This close-to-perfect correlation indicates that although the movie lists may not be strictly identical, movies are being replaced by similar (probably more recent) movies with similar genres. For instance, in both datasets *Drama* seems to be the most popular genre, followed by *Thriller* and *Crime*.

2.5.2 IMDb Bias

Another interesting approach towards investigating the biases in the MovieTweetings dataset is to compare with other popular datasets in the domain. This allows us to verify how inherently different the ratings originating from the IMDb platform are from other movie platforms like Netflix and MovieLens. For this experiment specifically, we compare

Rank	IMDb	MovieTweetings	Jaccard
1	The Shawshank Redemption	The Shawshank Redemption	1.00
2	The Godfather	The Dark Knight	0.33
3	The Dark Knight	The Godfather	1.00
4	Pulp Fiction	Pulp Fiction	1.00
5	The Godfather: Part II	LOTR: The Return of the King	0.67
6	The Good, the Bad and the Ugly	Terminator 2: Judgment Day	0.50
7	LOTR: The Return of the King	Schindler's List	0.56
8	Schindler's List	Forrest Gump	0.60
9	12 Angry Men	Saving Private Ryan	0.50
10	Inception	Inception	0.54

Table 2.3: List of top 10 popular movies from IMDb vs MovieTweetings

MovieTweetings with MovieLens because it is the most popular dataset in the recommender systems domain and is still publicly available[17] (the Netflix dataset is not anymore due to privacy issues).

Both MovieTweetings and MovieLens have a dataset snapshot containing 100K ratings, so these are obvious candidates for our comparative study. MovieTweetings is however a natural, unfiltered dataset, while in the MovieLens 100K dataset only users with at least 20 ratings are included. To increase the comparability of the datasets, we generated a new MovieLens dataset based on the data of MovieLens 10M. We sliced the first 100K ratings of this bigger dataset, taking into account the rating timestamps. While in MovieLens 10M all users have rated at least 20 movies, for its 100K subset, this will no longer be true, making the comparison with MovieTweetings more fair. We will refer to this dataset as the *ML 10M100K** dataset. Table 2.4 lists some of the basic characteristics for the rating datasets (i.e., number of users, items, ratings and density). The density metric is calculated according to the following equation:

[17]At `http://grouplens.org/datasets/movielens`.

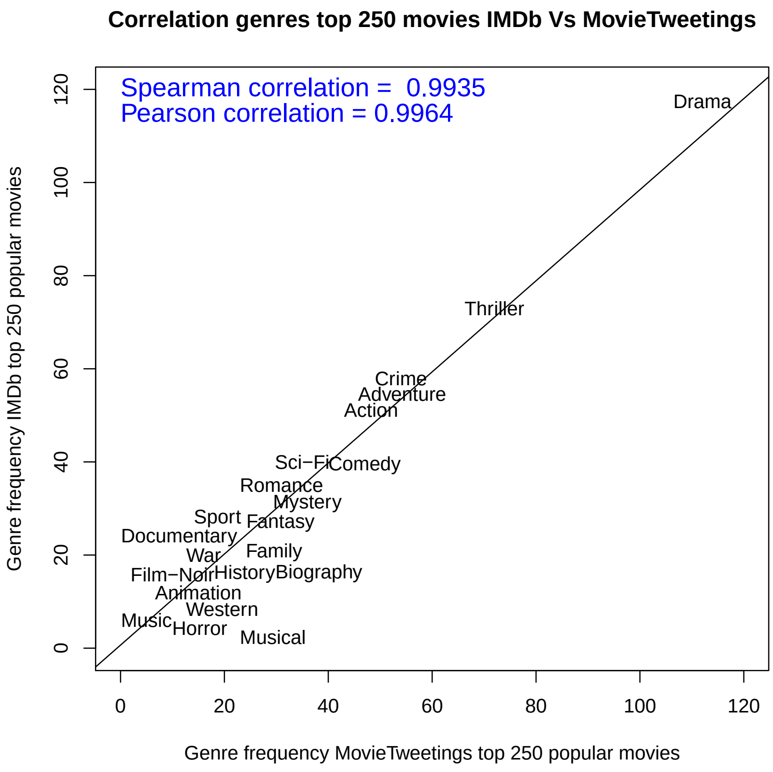


Figure 2.10: The correlation of how many times each genre occurred in the top 250 movie list for MovieTweetings and IMDb.

$$\begin{aligned} \textit{rating density} &= 100 \times \frac{\#\ \textit{available ratings}}{\#\ \textit{all possible ratings}} \\ &= 100 \times \frac{\#\ \textit{available ratings}}{(\#\ \textit{users}) \times (\#\ \textit{items})} \end{aligned} \tag{2.2}$$

These numbers clearly indicate one of the major aspects that differentiates the MovieTweetings dataset: its low density. The number of items and users is much higher (almost by a factor of 10) for MovieTweetings compared to the other datasets. While most rating datasets are restricted to users of the particular closed system (e.g., the Netflix or MovieLens system), the MovieTweetings dataset integrates data from users of the IMDb platform, which is very popular and open to anyone on the Internet. Also in terms of number of items the IMDb catalog is not restricted to movies that can be rented but rather includes almost all existing movies. Integrating the IMDb platform as rating source, therefore leads to very high numbers of distinct users and items, leading to a density value closer to 0 than for the other datasets.

	MovieTweetings	ML 100K	ML 10M100K*
#users	16,554	943	2,109
#items	10,506	1,682	655
#ratings	100,000	100,000	100,000
density	0.06%	6.3%	7.24%

Table 2.4: Basic characteristics of the datasets.

In terms of number of ratings, the three datasets are equal (i.e., 100K ratings), nonetheless it is worth comparing the distribution of their rating values to uncover different rating behavior. In Fig. 2.11 we plotted the histogram of the rating values for MovieTweetings. These values are based on the IMDb 10-star rating scale, whereas MovieLens users rated on a 5-star rating scale. The difference in rating scales makes it harder to compare the rating frequencies. Therefore, to ease the comparison with the MovieLens datasets, we paired the rating values for MovieTweetings in Fig. 2.12.

The histograms display the rating values on the X-axis and the number of times each rating value occurred in the dataset (i.e., frequency) on the Y-axis. A general trend is that the more positive rating values are more frequent in any of the three datasets. This is a well-known observation in the recommender systems research domain referred to as *not random missing data* [59]. Users mostly watch movies they assume to be interesting (based on, for instance, genre, trailer, or other movie information), and therefore most movies they rate (apart from the ones they wrongfully assumed interesting) will be rated positively, which explains why generally negative or low ratings are not present in these datasets.

Although all three datasets distributions are showing a skew towards more positive rating values, the trend is significantly stronger for MovieTweetings. Only 2% of its rating values are smaller than 5 (neutral rating), compared to the MovieLens datasets where 17% for ML 100K and 11% for ML 10M100K* are lower than 3 (the counterpart neutral value

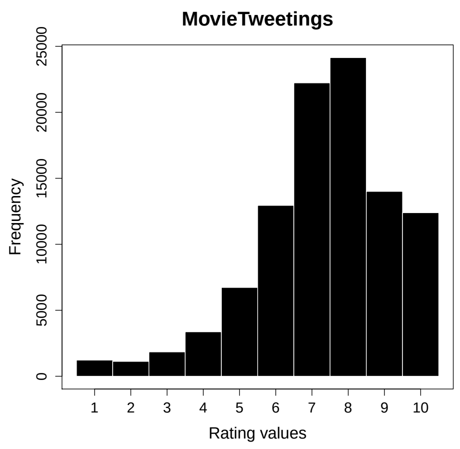


Figure 2.11: Rating value histogram for the MovieTweetings dataset. The rating values are based on the IMDb 10-star rating scale.

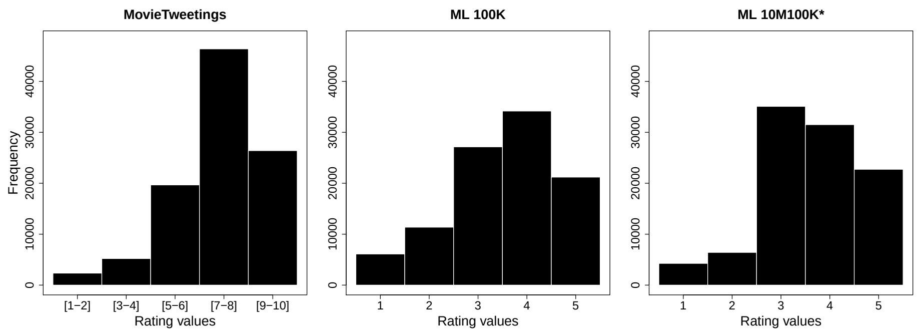


Figure 2.12: Rating value histograms for the datasets MovieTweetings, ML 100K and ML 10M100K* illustrating a similar (yet stronger for MovieTweetings) shift towards positive rating values across the datasets.

in these datasets). A possible explanation for this may be the fact that users are not explicitly asked to rate movies on the IMDb platform as is the case for the MovieLens system. IMDb only recently included (basic) recommendations on its website, hence there used to be no direct incentive to rate movies other than to contribute to the aggregated IMDb rating value for a specific movie. We hypothesize that therefore users are even more skewed towards only rating exceptionally good movies, resulting in higher rating values.

Aside from the rating value distribution, it is also important to compare the general rating distribution among the datasets. In Fig. 2.13 the distribution of the ratings across the items is shown (similar to [60] where Netflix and MovieLens were compared). Items are ordered by popularity (in terms of number of ratings) and expressed as a percentage of the total number of items. From the figure we can see that in the case of MovieTweetings, 40% of the total amount of ratings is provided on only 1% of all the items (i.e., the 1% most popular), compared to MovieLens where this is 10%. Thus, in general the MovieTweetings dataset is less dense, but 40% of the ratings are mostly concentrating on a very small number of items which greatly increases the density for these items in particular (the top 105 most popular movies are each rated on average 380 times). The ML 10M100K* rating distribution is somewhat more similar to MovieTweetings than ML 100K is, but both are in fact still significantly different from the MovieTweetings rating distribution. Hence, MovieTweetings is more biased towards popular movies.

An important aspect about the MovieTweetings dataset is its recentness. Since ratings are mined from Twitter, there is no limitation as to how old or recent a rated movie should be. Data shows however that recent movies are rated more, obviously because users tend to rate the movies they have just seen, and recent movies are more easily available (in cinemas) while being – probably – more interesting topics to share on a social network. Specifically, in Fig. 2.14 we plotted the year of every rated movie and its frequency for the three datasets. In general, similar patterns can be observed: a long tail with a peak at the end. The location of the peak indicates the most recent movies in the dataset, which is 2013 for the MovieTweetings (100K) dataset and towards the end of the 90s for the MovieLens-based datasets. The histograms here indicate how old the MovieLens datasets truly are. The ML 100K dataset includes more ratings from older movies, which may still be relevant data for some use cases (e.g., recommending classic movie titles to older users) but in most cases, however, users will prefer recommendations for modern and

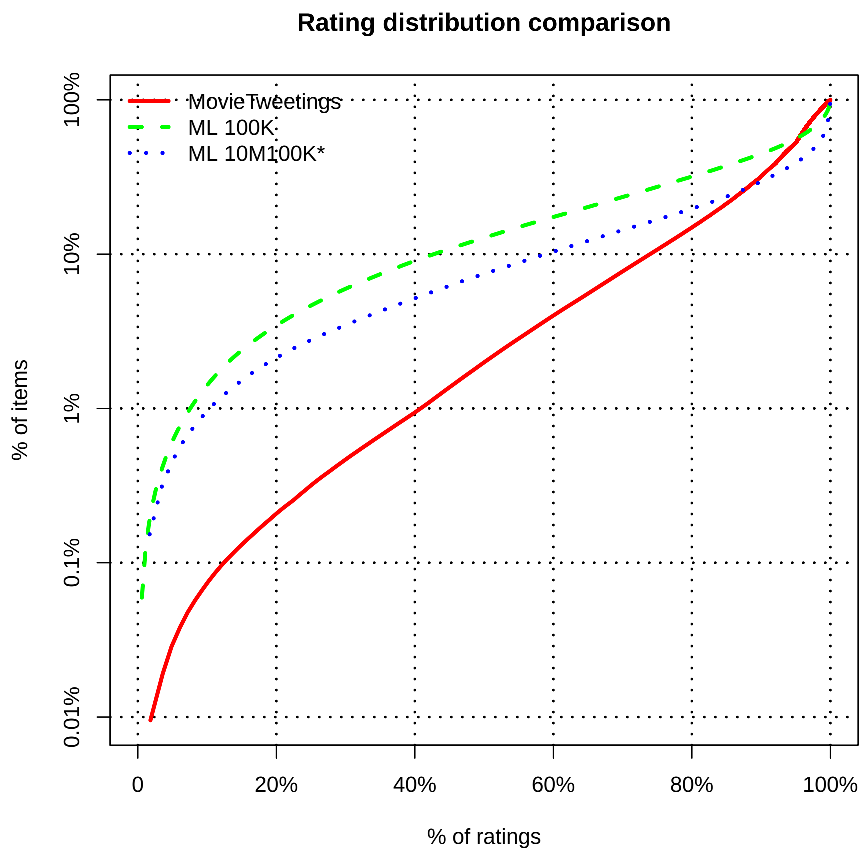


Figure 2.13: Rating distribution comparison, linking the number of ratings with the number of rated items. Items are sorted according to popularity i.e., most rated, with the most popular items at the bottom. Note that the Y-axis is a log scale.

recent movies and for those situations the MovieTweetings dataset may offer an ideal way of bootstrapping a recommender system and avoid cold-start issues for new users or unrated items.

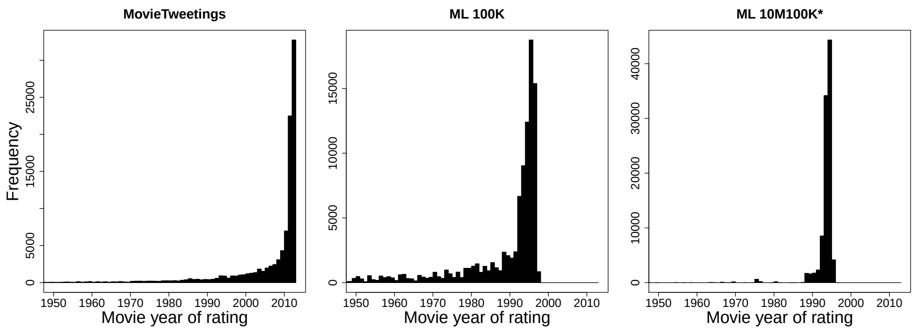


Figure 2.14: Histograms illustrating how frequent movies of a given year (on the X-axis) were rated for MovieTweetings, ML 100K, and ML 10M100K*. While MovieTweetings is updated frequently, and therefore contains very recent movies, the most recent movies in the MovieLens dataset are from the late 90s.

Similarly as we presented in the previous section, where we compared the IMDb ratings with MovieTweetings, we now compare MovieTweetings with MovieLens by means of a rating correlation analysis. To be able to easily compare the rating values we rescaled the MovieTweetings ratings to a 5-star scale. For movies in MovieLens also present in the MovieTweetings dataset, we calculated the average movie rating and correlated the results. We present the results for the ML 10M100K* dataset.

Fig. 2.15 shows the results for movies which have been rated at least 1, 2, 3, 4, 5 and 20 times. The exact Spearman and Pearson correlation values are listed in Table 2.5. These figures show trends similar to the comparison of MovieTweetings and IMDb data (see Section 2.5.1). For all movies (i.e., figure for movies with $>=$ 1 ratings) a general trend of positive correlation can be noted, while some movies show diverging rating values (i.e., dots arranged vertically on the figure). This effect decreases when we restrict the item set to movies with a minimum of 2, 3, etc. ratings, which again confirms the diverging rating values to originate from movies which have been rated only a few times. Correlation values get stronger when increasing the minimal rating threshold per movie, except in the last case (the subset of movies with at least 20 ratings), where there are too few movies (i.e., less than 30) with at least 20 ratings that occur at the same time in the MovieTweetings and the

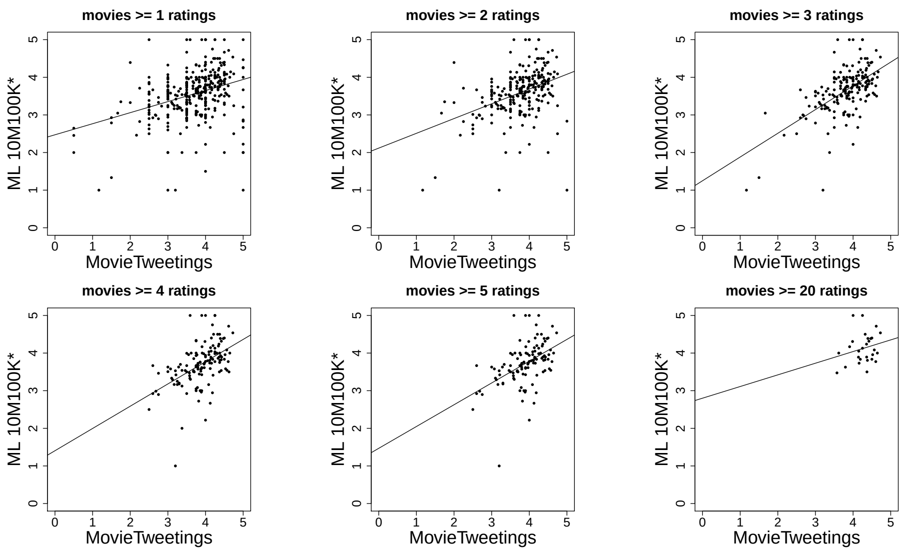


Figure 2.15: Scatter plot illustrating the correlation of the average MovieTweetings rating per movie and the ML 10M100K* dataset for different subsets of movies which have been rated at least 1, 2, 3, 4, 5, 20 times. The MovieTweetings results have been rescaled to a 5-star rating scale to make the ratings more easily comparable.

ML 10M100K* dataset to make a proper analysis.

Included movies	Spearman	Pearson
movies >= 1 ratings	0.367	0.335
movies >= 2 ratings	0.438	0.401
movies >= 3 ratings	0.543	0.593
movies >= 4 ratings	0.555	0.490
movies >= 5 ratings	0.539	0.481
movies >= 20 ratings	0.230	0.233

Table 2.5: Correlation values for average movie ratings of MovieTweetings and ML 10M100K*.

Similar results were obtained for the ML 100K dataset; more specifically, ML 100K presented a slightly larger overlap with MovieTweetings in terms of movies (i.e., 80 movies with at least 20 ratings), but the overlap between these datasets was still too limited to perform a thorough popularity comparison.

In summary, in this section we studied the biases that influence the MovieTweetings rating data. With a sufficient number of minimum ratings per movie the dataset correlates strongly with ratings found on the IMDb website, as confirmed by the popularity analysis. We therefore consider the bias introduced by using only ratings posted on Twitter (instead of all IMDb ratings) to be not significant or even non-existing. The dataset does however show a bias towards very recent and popular movies, but a similar trend could be noted for the MovieLens dataset (focused heavily on movies from the late 90s).

2.6 Benchmarking the MovieTweetings Dataset

In this section, we analyze the MovieTweetings dataset under several conditions (data splitting, performance of recommendation methods, and evaluation metrics) and compare these results with those obtained using other datasets.

2.6.1 Experimental Setup

As in Section 2.5, we use two versions of the MovieLens data (i.e., *ML 100K* and *ML 10M100K**) against which we compare the MovieTweetings (MT) 100K snapshot. We follow the evaluation methodology presented in [61], where for each user a set of non relevant items (unrated by this user in the training and test splits) is randomly selected (100 in our case), and then, for each highly relevant item in the test split (i.e., those rated as 5 in MovieLens or as 10 in MovieTweetings), a ranking is generated by predicting a score for this item and the other (not relevant) items. Then, the performance of this ranking is measured using the *trec_eval* program[18]. In this way, standard retrieval metrics such as precision, normalized Discounted Cumulative Gain (nDCG) or Mean Average Precision (MAP) could be used [58]. Additionally, and for the sake of comparison with other research, we also measured error-based metrics such as Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE), pervasive in recommender systems literature [27].

We have tested five recommendation algorithms as implemented in the MyMediaLite Recommender System library (version 3.10)[19]. All of them only use ratings as the basis of the predictions, two are non-personalized

[18]Available at `http://trec.nist.gov/trec_eval`.
[19]Available for download at `http://mymedialite.net`.

(user and item average) methods, and the rest do use information about the target user, either as memory-based collaborative filtering algorithms (user and item nearest neighbor) or as a model-based (MatrixFactorization) recommender. Table 2.6 shows a description of these approaches, along with the default parameters for MyMediaLite version 3.10 used in our experiments. Note that some of these specifications are not standard (e.g., the cosine function as similarity, or using a baseline predictor in the nearest neighbor methods), but since they are applied to all the datasets impartially, the overall conclusions should be fair and not sensitive to this aspect. In any case, we did a preliminary test with Pearson's correlation as similarity function and the general trend remained the same; the only change was that the performance of UserKNN and ItemKNN improved slightly (but uniformly in the three datasets).

Name	Description	Parameters
ItemAverage	Target item's average rating	–
UserAverage	Target user's average rating	–
ItemKNN	Item-based nearest neighbor	k=80, correlation=BinaryCosine, reg_u=15, reg_i=10, num_iter=10
UserKNN	User-based nearest neighbor	k=80, correlation=BinaryCosine, reg_u=15, reg_i=10, num_iter=10
MatrixFactorization	Factorization of rating matrix using stochastic gradient descent	num_factors=10, regularization=0.015, learn_rate=0.01, learn_rate_decay=1, num_iter=30

Table 2.6: Description of the evaluated recommendation algorithms and the values of the used parameters.

2.6.2 Experiments

We present the results obtained when comparing the algorithms listed in Table 2.6 for the three datasets introduced before (whose statistics

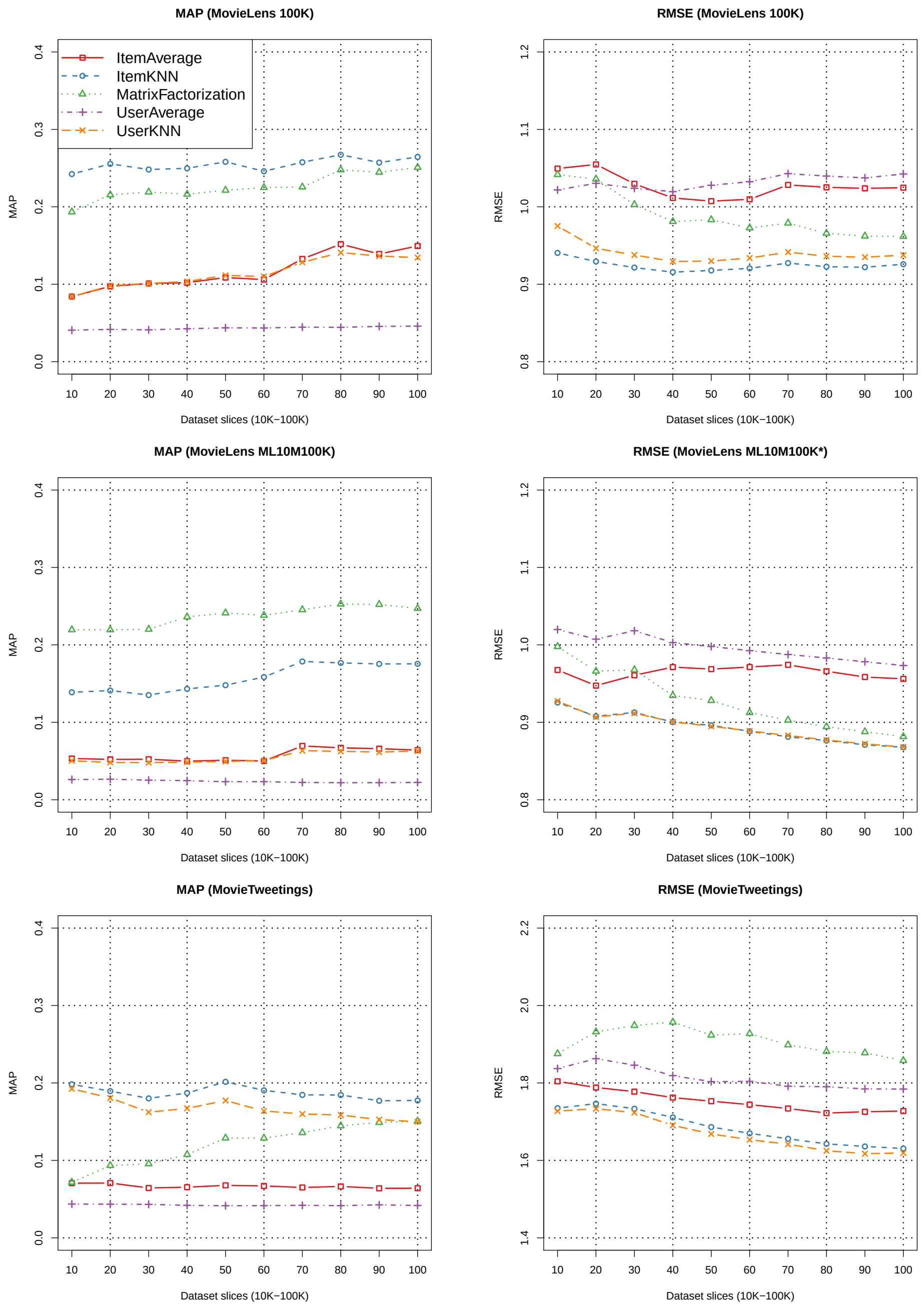


Figure 2.16: Mean Average Precision (MAP, the higher the better) and Root Mean Squared Error (RMSE, the lower the better) metrics computed using a cross-validation 5-fold splitting strategy.

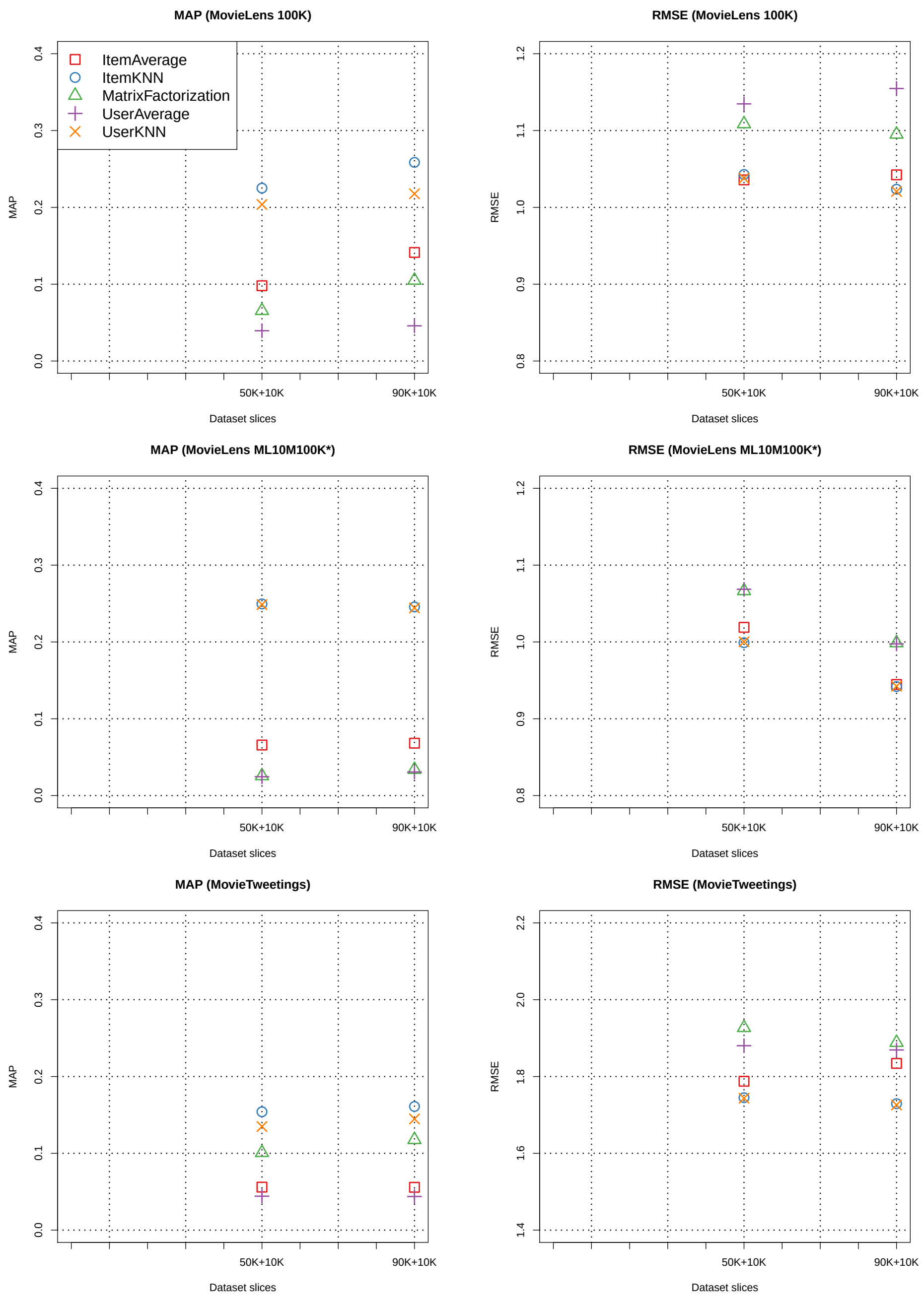


Figure 2.17: MAP and RMSE metrics computed using a temporal splitting strategy.

are summarized in Table 2.4). We analyze these results (presented in Fig. 2.16 and Fig. 2.17) according to three dimensions: evaluation metrics, data splitting, and recommendation performance. We focus on Mean Average Precision (MAP) [58] and Root Mean Squared Error (RMSE) [27] as evaluation metrics. Other ranking-based metrics like nDCG, precision, and recall produced similar results as those obtained for MAP, likewise for MAE with RMSE. For data splitting we experiment first with a standard way for randomly generating training and test splits: a cross-validation splitting strategy that generates non-overlapping subsets (to be used as training and test splits) where every (user, item, rating) tuple is evaluated once i.e., it only appears in one test split, and it is guaranteed that there is one split containing such tuple. The results have been averaged over 5 folds, but similar results were found with 10 folds. The second evaluated splitting strategy is based on temporal splitting [62], where the test split occurs after the training split. We use $10,000$ ratings for testing, and experiment with a window of the previous $50,000$ or $90,000$ ratings as training split.

From Fig. 2.16 and Fig. 2.17 we observe that the performance of the recommender algorithms is heavily influenced by the evaluation metric (RMSE[20] or MAP) used to decide which recommenders perform better. Specifically, whereas the ranking-based metric (MAP) is very stable – in terms of dataset snapshots and splitting strategies –, preserving the trend in best/worst recommenders, the error-based metric (RMSE) has more fluctuations. Furthermore, the results of the RMSE metric are not useful to discriminate which recommender is performing best because the values are very close to each other, and, in general, it is not consistent that the best method with RMSE (lowest value) achieves the best value with MAP (highest value) or vice versa[21]; in particular, the worst method according to RMSE for MovieTweetings is the MatrixFactorization algorithm, which has a medium-to-high performance in terms of MAP. We argue some of these differences between MAP and RMSE may be due to the different levels of density (as defined in Equation 2.2) presented in each of these datasets, since in such scenario it is more likely that more users or items may have no training information (or very little) after splitting the dataset, which seems to have a stronger effect on error-based metrics and leads to very similar performance values for

[20]Note that for RMSE, the range of ratings in MT is different (from 1 to 10) than the one from ML (from 1 to 5).

[21]This conclusion confirms works such as [60, 63, 64] where error-based metrics show different behaviors, usually not linked with the final experience of the user.

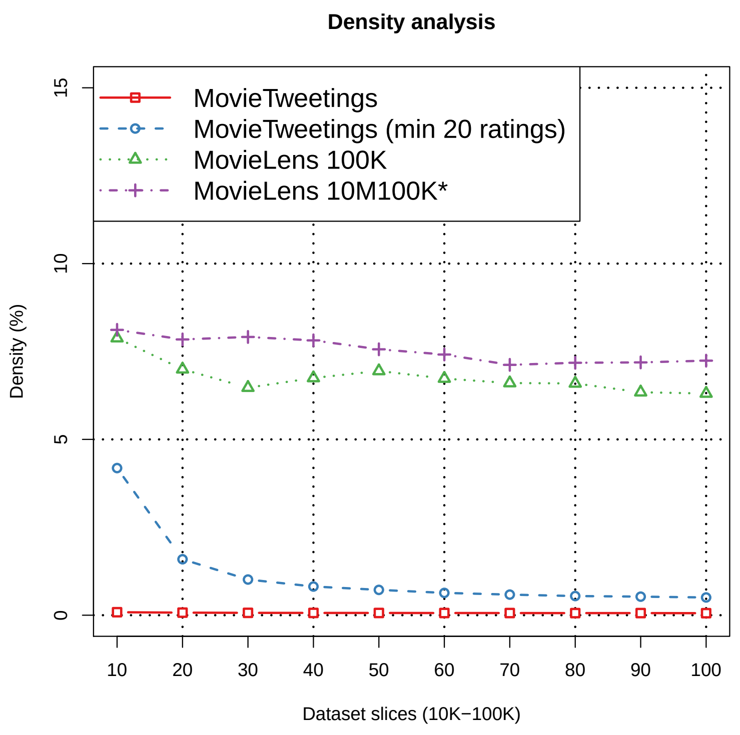


Figure 2.18: Comparison of density values for each snapshot of the datasets analyzed in the paper.

very different recommendation methods. Fig. 2.18 compares the density in these datasets as a function of the number of ratings increasing in steps of 10K; it is clear that the sparsity in the original MovieTweetings dataset is higher (i.e., density is lower), mainly because it contains a much larger number of users and items than the other datasets, but keeps the same number of ratings. This aspect of the dataset may also affect the fact that MAP is lower in MT than in ML datasets, because it is a more difficult (i.e., less dense) dataset.

To further analyze the differences in behavior when other assumptions in the dataset are considered, we generated a subset of MovieTweetings where only users with at least 20 ratings are kept. As we see in Fig. 2.18, this artificially generated dataset is less sparse than the original MT dataset, although it gets closer to the original dataset when more ratings are considered; this can be explained by looking at the dynamics of the number of users and items available at each snapshot point (Fig. 2.19).

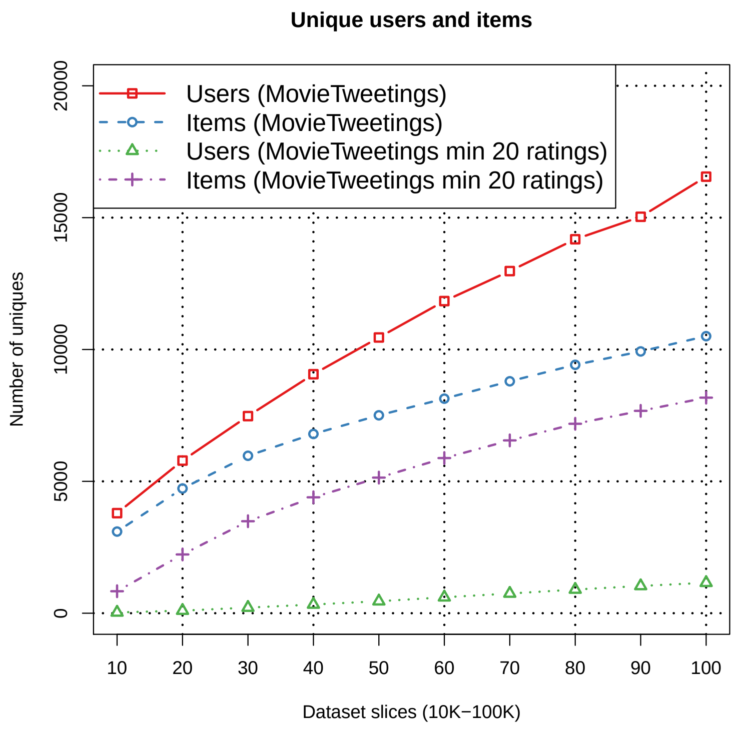


Figure 2.19: Unique users and items at each MovieTweetings snapshot.

Furthermore, the recommendation performance (illustrated in Fig. 2.20) has now changed and the relative performance between the recommendation methods changes more often than in the previous case; also the range between RMSE values is larger for some of these algorithms and, especially for MAP, there are different best methods at each point, like the MatrixFactorization method, that outperforms the other algorithms after the 40K slice.

Regarding the performance of recommendation algorithms, each dataset has a different optimal method, but in general the MatrixFactorization recommender is among the top performing recommenders, in agreement with previous research on rating-based recommendation [65, 66], and at the same time, the non-personalized recommenders (user and item average) have a very low performance. It should be noted that the user-based nearest neighbor and the item average recommenders are equivalent in the MovieLens datasets in terms of MAP (meaning that their rankings are effectively the same), whereas in MovieTweetings

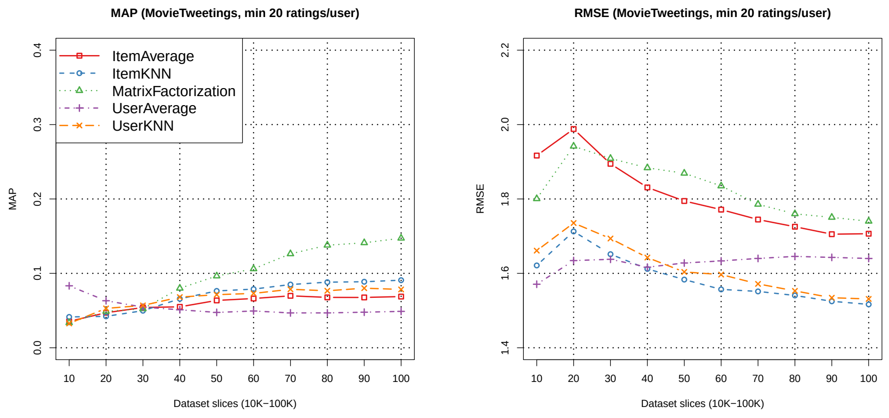


Figure 2.20: MAP and RMSE metrics computed using a cross-validation 5-fold splitting strategy for the users in the MovieTweetings dataset with at least 20 ratings.

they perform differently. Besides, the two neighbor-based recommenders (UserKNN and ItemKNN) outperform the other algorithms in Movie-Tweetings, both in terms of MAP and RMSE.

The relative ranking-based performance, in most of the cases, does not change too much from the first snapshot (10K ratings) to the last one (100K ratings). One exception to this is the MatrixFactorization algorithm in the MT dataset. As observed before in literature [67, 68], recommendation performance increases in the MovieLens datasets when more ratings are available, but this is not the case with the MovieTweetings dataset, where all the recommenders, except for the MatrixFactorization method, decrease or maintain their performance. A similar result was observed in [69], where a dataset with several new items was used, which lowered the recommendation precision. These results are completely reversed when the subset of users with more than 20 ratings (Fig. 2.20) is analyzed, since here the performance increases with more data (except for the UserAverage method). Note also that the best recommender in terms of MAP (the MatrixFactorization method) is the worst according to RMSE. A possible explanation to these effects is the amount of users and items presented in this constrained dataset (Fig. 2.19), where a smaller number of items and, especially, of users is available in the dataset, which makes it easier for the recommenders to obtain a higher performance.

Error-based metrics, as discussed before, are not very useful to decide which is the best recommender. In the same way, it is very difficult to decide in which of the snapshots the algorithms perform better. These facts make the error-based metrics not so interesting to predict which recommenders will perform better in the future, given a particular snapshot. These predictions, however, could be easily drawn from the results with the ranking-based metrics.

Finally, it is interesting to note that the results obtained in each of the data splitting techniques evaluated are very consistent for the MT dataset, whereas this is not the case for the ML datasets. Specifically, the best and worst recommenders remain the same for ML 100K, but the ones in the middle vary drastically their relative performance; even worse, for ML 10M100K* the best recommender using cross-validation (MatrixFactorization) is the worst using a temporal split. This consistency in the evaluation for the MovieTweetings dataset is a positive characteristic, since it shows a direct correspondence between the standard offline evaluation using repeatable tests (cross-validation) and the more realistic evaluation scenarios (temporal split).

In summary, in this section we have evaluated different aspects of the MovieTweetings dataset from a practical perspective. We have found that it is not very different from the two versions of the MovieLens dataset we have experimented with when using a specific splitting strategy, but it is more consistent across data splitting strategies. Nonetheless, we have observed that its very high sparsity may produce lower performance scores in general, and that the RMSE scores are not very useful to discriminate the recommendation methods, although this was also true – to some extent – for the other datasets.

2.7 Cross-Domain Datasets from Twitter

In the previous sections we illustrated how movie ratings originating from IMDb can be structurally mined from Twitter. In this section, we generalize our method to other (non-transient) item domains, and provide the tools to allow researchers to collect and build cross-domain rating datasets for themselves. Our mining method can be applied to any website that offers a *social sharing* functionality posting pre-formatted tweets – containing user feedback – on Twitter. We illustrate this on three major online platforms in very divergent item domains: Goodreads[22] (books),

[22]`http://www.goodreads.com`

Pandora[23] (music) and YouTube[24] (video clips).

2.7.1 Books - Goodreads

Goodreads is one of the largest websites for books discussion and discovery. Readers can review books and receive recommendations based on their personal taste. This website also offers to tweet about the review or rating a reader has provided. On Goodreads the following tweet template structure is used.

```
<Rating> of 5 stars to <Title> by
<Author> <Link to review on Goodreads>
```

To obtain the tweets originating from the Goodreads website, we query the Twitter API for 'of 5 stars to'. From the tweet the following information can be extracted:

- User (Twitter user id)
- Rating (5-star scale)
- Book title
- Book author
- Goodreads URL of the review

Since the URL refers to the review on the Goodreads website posted by the same user the tweet originates from, additional metadata fields (e.g. Goodreads user id, book id) can easily be extracted. Ratings provided by users on Goodreads are publicly available, so if the Goodreads user id is known, all of that user's ratings could additionally be extracted from the website to expand the rating dataset even further.

2.7.2 Music - Pandora

There are many online services for music, one of which is the online radio service Pandora. Using Pandora, users can easily stream music and receive song recommendations. The website offers a similar social share feature as we found for IMDb and Goodreads. Users can tweet about the song they are currently listening to, using the following predefined tweet format.

[23] `http://www.pandora.com`
[24] `http://www.youtube.com`

```
I'm listening to "<Title>" by <Artist> on
Pandora <Link to song on Pandora> #pandora
```

Very similar to the data available from tweets originating from the Goodreads website, the data fields available in this tweet format are:

- User (Twitter user id)
- Song title
- Song artist
- Pandora URL of the song

The difference here, is the lack of an explicit rating value. Pandora users do not rate the music, they either listen to it, or they do not. The tweets originating from the Pandora platform should therefore be considered implicit feedback. The query we use to get the Pandora tweets from the Twitter API is 'I am listening on #pandora'. The Pandora URL is available, so again additional metadata (e.g., music genre) can be extracted.

2.7.3 Video clips - YouTube

YouTube is currently the biggest provider of short video clips on the Internet and is widely famous and well-known worldwide. While YouTube used to have a 5-star rating scale, in 2009 it was replaced with a thumbs up/down system because it more closely aligned with typically observed rating behavior[25]. When users watch videos on YouTube they can rate them by clicking a like or dislike button and tweet about it. The preformatted tweet in this situation shows the following structure.

```
I liked a @YouTube video [from @uploader]
<Link to YouTube video> <Title>
```

To restrict the Twitter API to results originating from these YouTube related tweets, we employ the query 'I liked a @YouTube video'. The data fields that can be extracted from the resulting tweets are:

- User (Twitter user id)

[25] http://youtube-global.blogspot.be/2009/09/five-stars-dominate-ratings.html

- The @handle of the video owner (optional)
- YouTube URL of the video

While there is no star-rating involved in this scenario, the feedback gathered is explicit feedback (i.e., the user explicitly expressed that she liked the video). The URL contains the unique YouTube identifier for the video which can be used to request additional content data (e.g., tags) from the YouTube API.

2.7.4 Cross-Domain Mining Experiment

To validate our generalized method of mining rating datasets from Twitter, we set up an experiment to automatically build 4 rating datasets, one for each of the previously discussed online platforms.

2.7.4.1 Experimental Setup

For each of the online platforms (i.e., IMDb, Goodreads, Pandora and YouTube) we queried the Twitter API at fixed time intervals (between 5 and 30 minutes) to download all tweets containing the aforementioned preference indicators. The frequency of querying the Twitter API depended on the typical number of tweets associated with the specific website. YouTube tweets were much more numerous than tweets from IMDb, so we had to query the Twitter API more frequently (every 5 minutes) to capture all the relevant tweets, while respecting the Twitter API limitations.

The data fields as discussed in the previous section were extracted by means of a series of specific regular expressions and stored line by line in dataset files. Ratings were mined over a period of 2 weeks (from December 19, 2013 to January 2, 2014) and processed in 4 resulting datasets. The (Python) scripts used for the downloading and processing of the files, and the resulting datasets are available on the Github platform[26].

2.7.4.2 Results

Table 2.7 lists the basic characteristics for each of the 4 collected datasets. While each dataset was mined on Twitter for the exact same period of time, the number of extracted ratings is significantly different. The

[26] `https://github.com/sidooms/Twitter-ratings`

	IMDb	Goodreads	Pandora	YouTube
#ratings	9,297	43,960	1,468	2,867,182
#users	3,412	19,680	1,039	420,373
#items	2,689	27,403	425	1,112,292
avg rat./day	664	3,140	105	204,799
sparsity	0.99899	0.99992	0.99668	0.99999
density	0.101%	0.008%	0.332%	0.001%

Table 2.7: Dataset statistics for a 2 week mining period.

most ratings were collected from the YouTube platform, the fewest from Pandora (about 2000 times less). For all of the datasets the density (see Equation 2.2) turned out to be very low. This is to be expected since the collected datasets are unfiltered i.e., contain not only users with >20 ratings. The low density values indicate high numbers of users and items with only little rating information to link them, which can be a major problem for collaborative filtering recommender systems known as the *sparsity problem*[27] [70, 71]. A typical long-tailed distribution was found when we inspected the number of ratings per user i.e., many users had a low number of ratings (<5).

Fig. 2.21 illustrates the extreme difference in numbers of ratings collected from each platform during our mining period. The figure shows the daily number of ratings which varies day by day mostly depending on the day of the week (more activity in weekends).

An emerging research topic in the recommender systems area is cross-domain recommendation [72]. In such scenarios the sparsity problem is usually alleviated by integrating data from another domain e.g., recommending books based on previous movie ratings. One of the main challenges the domain faces is the lack of cross-domain rating datasets, which forces researchers to work with artificially generated datasets. With our approach we find ourselves in the unique position of linking rating data originating from the same (Twitter) user across multiple item domains (books, movies, music, etc.). For this purpose we analyzed the intersections of the 4 collected datasets, more specifically the intersection of users (i.e., Twitter user ids that have ratings in more than one dataset).

Fig. 2.22 shows the result of the intersection analysis for YouTube,

[27]*Sparsity* is the inverse of *density*, both terms are used in recommender systems literature.

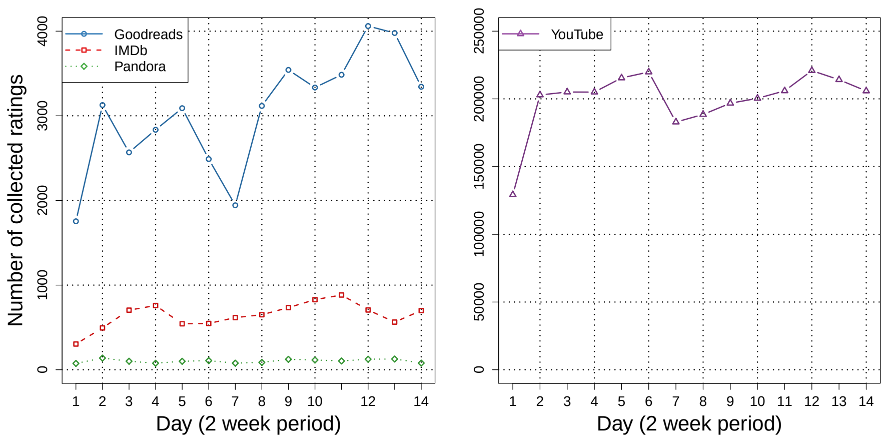


Figure 2.21: The daily number of collected ratings from Goodreads, IMDb, Pandora and YouTube. YouTube is displayed separately because of the large difference in Y-axis scale.

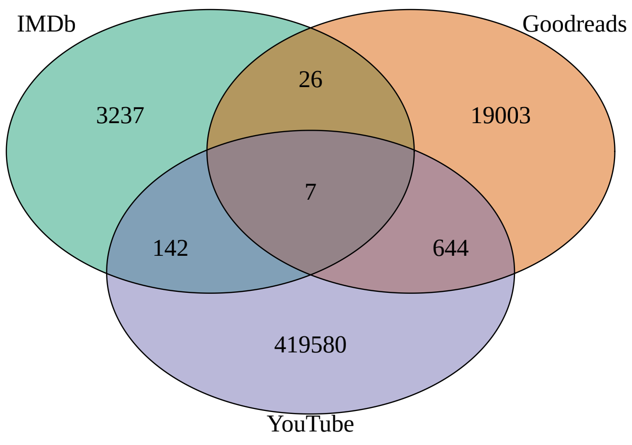


Figure 2.22: Venn diagrams indicating the numbers of unique users for the IMDb, Goodreads and YouTube datasets and their intersections.

Goodreads and IMDb (Pandora was omitted because of the low number of collected ratings). Some users turned out to actually rate across more than one domain. In total, 7 users were even found to rate across all three of the datasets. Considering the short period of data collection (2 weeks), these results look very promising for the creation of cross-domain rating datasets.

2.8 Conclusion

This chapter reflected on user feedback, which could be considered the fuel that drives recommender engines. Although user feedback is one of the main ingredients for the recommendation process, only a few public rating datasets are available for research purposes. We discussed how these datasets are often old and artificial, which limits their usability to offline comparative experiments.

We studied the collection of different types of user feedback in a realistic scenario using a cultural events website as experimental platform. Results of 7 months of data illustrated how implicit feedback was much easier to collect in large numbers compared to explicit feedback (i.e., ratings). The latter however provides more intrinsic information value regarding the true preferences of the user. Although an interesting feedback dataset was collected, a time analysis showed how the feedback was very time-bound (because of the event domain) and research performed using this dataset may not generalize very well to other item domains (e.g., non-transient domains as music, books, movies, etc.).

To assemble a more generalizable dataset useful for various research purposes, we turned to the online world and discovered an interesting connection between IMDb and Twitter. While rating movies, users of the mobile IMDb iOS app are offered to tweet their ratings in a structured format. Using the Twitter API, such tweets can be downloaded and the integrated ratings extracted. We published a new rating dataset called *MovieTweetings* based on this idea, integrating newly tweeted ratings on a daily basis.

We investigated how much the MovieTweetings ratings were biased in comparison with general ratings from IMDb (not published through Twitter) and ratings found in public rating datasets as MovieLens. The ratings seemed to correlate strongly with IMDb ratings, when a large enough number of ratings per movie was considered (>20). The dataset also showed similarities to the MovieLens dataset, but proved to incorporate much more recent and popular movies. The main differences are the fact that MovieTweetings is a natural dataset (no users are excluded), and many more unique items and users are included, which causes the density to be low (or *sparsity* to be high). The MovieTweetings dataset was benchmarked using multiple recommendation algorithms, evaluation metrics, dataset sizes and data splitting strategies. Again, similarities to the MovieLens dataset were noted, although evaluation results seemed more stable and consistent for the MovieTweetings data.

Finally, we generalized our approach of mining ratings from Twitter to other platforms implementing similar *social sharing* features. We illustrated how ratings from various domains could be collected including books (i.e., Goodreads), music (i.e., Pandora) and video clips (i.e., YouTube) by mining 3 additional datasets over a short period of 2 weeks. Since users were always originating from Twitter, the same user ids could be found in multiple datasets which opened the doors to true cross-domain rating datasets. Our experiments, above all, have shown the wealth of both explicit and implicit user feedback that can be easily extracted from public online sources these days. The datasets resulting from experiments in this chapter can be used to alleviate the shortcomings of the already available public rating datasets.

Chapter 3

Human-Recommender Interaction

3.1 Introduction

Human-recommender interaction (HRI) is a component of the recommendation process that is often overlooked, even though it contributes a great deal to the overall user experience and satisfaction of people using a recommender system. A car engine – as powerful as it may be – without a car structure, body and driver controls such as the steering wheel, pedals, etc. is just a piece of useless metal. The same applies to recommender systems. A mathematical model predicting user preferences, only becomes useful and interesting when it interacts with actual users and thus exposes its features.

Information systems expose their functionality by means of a user interface, which is most often graphically implemented (i.e., graphical user interface or *GUI*). However, with the advent of more complex technology as smartphones and tablets, also other types of interaction methods as *gesture control* and *voice control* are rapidly becoming integrated in our daily lives.

While the phrase "Design is not important, just make sure it works", is often heard in computer science courses, the impact of visual design and layout for user interfaces should not be minimized. Specifically for recommender systems it has been hypothesized[1] that the user interface may be responsible for 50% of the total user experience when recommen-

[1]`http://recsys.acm.org/2009/invited_talk_strands_martin.pdf`

dation results are shown to users.

In this chapter we acknowledge the importance of user interfaces and discuss their applications with respect to HRI. Specifically, we focus on two interaction processes relevant for typical recommender systems: providing input feedback and visualizing recommendation results. For both scenarios we discuss the current state of the art and detail our own contributions applied to actual use cases.

Research Questions

- What is the effect of visualization on the recommendation process?
- How can hybrid recommendations be visualized?
- How can the 'Filter Bubble' be pierced?

3.2 Feedback Mechanisms

Feedback mechanisms are user interaction controls designed to collect user feedback, in our case for recommender systems. While in the previous chapter we focused on the importance of the collected user feedback and its processing, here we elaborate on the design and the consequences of the interaction controls themselves.

Feedback mechanisms come in all shapes and sizes. The most common example is the 5-star rating system which allows users to express their preference for certain items on a 5-star rating scale. The 5-star rating system is incorporated in many well-known recommender platforms as MovieLens and Netflix. Another popular system is the thumbs up/down feedback system as used by YouTube. Closely related to the thumbs up/down system is Facebook's unary 'Like' rating system. IMDb implements a 10-star rating system, the Jester joke-rating system [73] a continuous rating bar which ranged between [-10,+10], the MPAA film-rating system (used to rate a movie's suitability for US audiences) rates movies in categories ranging from G (general audiences) to NC-17 (exclusively adult).

But what makes a good feedback mechanism? How does it affect users? What attracts most feedback and what results in the most interesting information from the viewpoint of the recommender system? In the following sections we elaborate on these questions.

3.2.1 Related Work

Cosley et al. [74] is one of the earliest but still relevant works that acknowledged and researched the importance of rating interfaces in recommender systems. They conducted three experiments with 536 users of the MovieLens website focusing on the topics of re-rating, different rating scales and rating manipulation. Some of their research questions where:

- How consistent are users when re-rating items?
- How do different rating scales affect users' ratings?
- Can the system make a user rate a 'bad' movie 'good'?

One of the experiments involved three different rating scales: binary (thumbs up / down), no-zero (a scale from -3 to +3, but no zero value), half-star (scale ranging from 0.5 to 5 in half star increments). For a full discussion we refer to their work, but some of the remarkable results they found where:

- Users' rating behavior seemed to be consistent. When re-rating previously rated movies a correlation of 0.7 was found, more or less in agreement with results from previous literature [75] (their correlation was 0.83).
- By showing false predictions of non-rated items, users can be manipulated to rate positively instead of negatively.
- Users preferred finer-grained scales.
- Finer-grained scales did not have an adverse effect on recommendation accuracy.

Cosley et al. showed how fine-grained rating mechanisms do not lead to a decreased accuracy of the recommender system. On the contrary, users may even be able to more accurately express their preferences which should lead to an increase in recommendation quality. But what is good for users and what users like, may not be the same thing. In more recent work, Preston and Colman [76] studied user responses on scales ranging from 2 to 101 points. Their users seemed to be happiest (and more consistent) with a 5-10 point scale while they indicated that the 101-point scale was actually better at capturing their feelings.

There may also be a trade-off to consider between user benefit and user effort. In [77], ratings are considered from an economic viewpoint.

They pose that users are only willing to provide an effort in proportion with the actual received benefit. Such benefits may be improved recommendations, unlocked features on a website, and so on. So even though a 1000-point scale may actually be best at capturing the different levels of preferences for all users, it may too negatively skew the user effort-benefit ratio. Users may find it more difficult to rate on too fine-grained scales, leading to inconsistency, user frustration and possibly overall reduced user interaction.

Some recent research has specifically compared the observed user effort for different rating systems. Sparling et al. [78] set up a user survey where 348 users where asked to rate items from two item domains: movies and Amazon reviews. Four popular online rating systems were included in the survey: the unary 'like it' scale, the binary thumbs up/down scale, the 5-star scale, and a 100-point slider scale. Various experiments measured the costs and benefits associated with the different rating scales. As expected, the more fine-grained rating mechanisms where found to require more cognitive effort from users. They also attracted fewer ratings compared to the 'easier' rating systems. Since the experiments spanned two item domains, results could be compared and some were actually found to be influenced by the respective domain: a higher satisfaction for the 5-star scale was noted for the movie domain while the thumbs scale noted a slightly lower satisfaction. The authors cautiously hypothesize that users may prefer finer-grained scales more for subjective item domains. Overall the 5-star scale was found most preferred by users (the survey asked users to rate the rating systems). The authors propose the interesting idea of dynamically adapting the rating mechanism to the user context. A recommender system may show a coarse-grained rating scale for new users at first, while after a while users may be willing to express their preferences on a more finer scale depending on their growing experience with the system and its item domain.

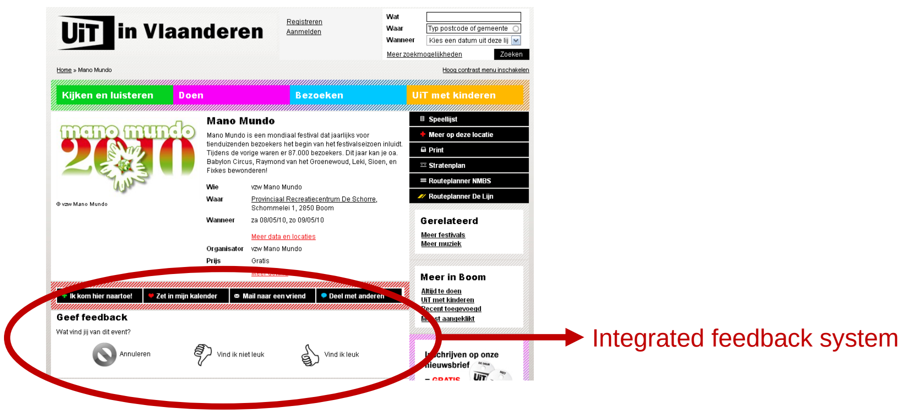
In the next section we present the results of our own rating mechanism experiment where we investigate the popularity of different rating systems and their ability to attract user feedback as observed on an actual (non lab-controlled) website.

3.2.2 A Cultural Events Website: Use Case Study

In this section we study user behavior towards 4 different explicit feedback mechanisms that are most commonly used in online systems. We monitored and analyzed the behavior of users towards these systems in a real online environment. Related work has already stated that recommender interfaces can influence users' opinions and therefore their ratings. There is however little knowledge on the influence that design of feedback mechanisms has on the willingness for users to give feedback. We wanted to capture the popularity of each system and track the specific interaction of users.

3.2.2.1 The Experiment

We collaborated with a popular Belgian cultural events website[2] and integrated some custom feedback mechanisms on the events information pages. We expanded these event detail pages with a custom built (Drupal) module that allowed users to rate the events. Graphical design was carefully attended to, to ensure optimal integration in the general look and feel of the website (Fig. 3.1).



Mano Mundo

Mano Mundo is een mondiaal festival dat jaarlijks voor tienduizenden bezoekers het begin van het festivalseizoen inluidt. Tijdens de vorige waren er 87.000 bezoekers. Dit jaar kan je oa. Babylon Circus, Raymond van het Groenewoud, Leki, Sioen, en Fixkes bewonderen!

Wie	vzw Mano Mundo
Waar	Provinciaal Recreatiecentrum De Schorre, Schommelei 1, 2850 Boom
Wanneer	za 08/05/10, zo 09/05/10
	Meer data en locaties
Organisator	vzw Mano Mundo
Prijs	Gratis

Geef feedback

Wat vind jij van dit event?

Figure 3.1: Screenshot of an event detail page which shows the integrated feedback system at the bottom.

We implemented four separate feedback systems: A 5-star rating system, a thumbs up/down rating system, and for each of them both a static and dynamic version (Fig. 3.2). The static rating systems were *HTML* form based. The user had to select a radio button associated with the desired rating and click a submit button to confirm. Doing so submitted the rating and caused a full page refresh. The dynamic systems used

[2]The website was the same cultural events website as discussed in the previous chapter Section 2.3.

JavaScript to capture onclick events and displayed a small color changing animation when hovered over the desired rating value. Clicking a value submitted the rating in the background without any portion of the page refreshing.

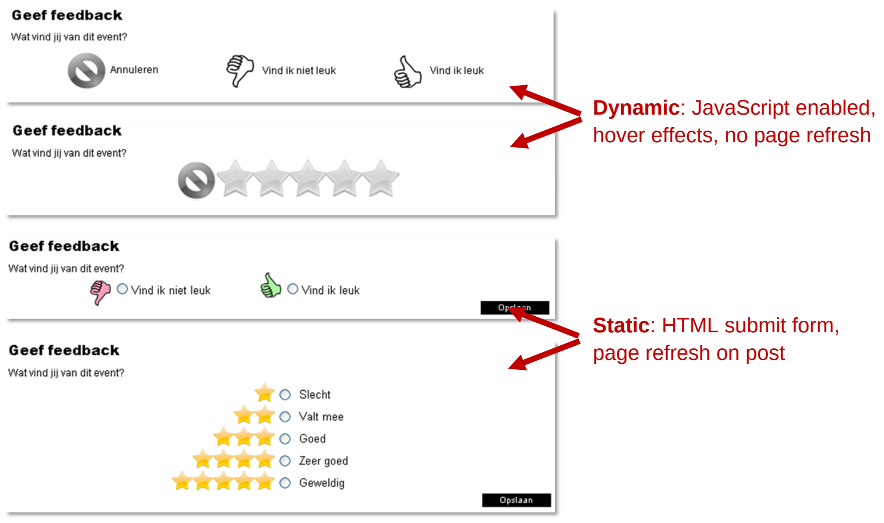

Geef feedback
Wat vind jij van dit event?
Annuleren
Vind ik niet leuk
Vind ik leuk
Dynamic: JavaScript enabled, hover effects, no page refresh
Opslaan
Slecht
Valt mee
Goed
Zeer goed
Geweldig
Static: HTML submit form, page refresh on post

Figure 3.2: Four explicit feedback systems (5-star scales and thumbs up/down) as commonly found online. User interface text in Dutch because they were integrated in a Belgian website.

For an accurate comparison of the different feedback systems, every system needed to be displayed in the exact same circumstances. We wanted to avoid any temporal effects and community influences that could render the data unreliable. Temporal effects are the effects that a different time frame could have on the experiment. Displaying every feedback system for one week at a time could favor a system that was displayed during a busier period like a holiday, rainy day, etc. The community influences we tried to avoid, include an increased user base and changed availability of content. If the feedback systems would be deployed sequentially then the last one would profit from the advantage of the largest user base, as we observed that the number of users grows on a daily bases. Since we are dealing with events as content, old events are removed from the site while new ones are added. This difference in number of events that are offered could also favor some feedback systems over others.

The standard way of dealing with these issues would be to employ an

A/B test where visitors are randomly divided in four groups each with their own feedback system. We wanted however to track individual user preferences towards all the systems and so every user had to be able to use every system. In our experiment every pageview showed a random feedback system. That way every system received an equal number of views, they all share the same settings of the experiment and users are not limited to the same feedback system.

3.2.2.2 Experimental Results

For a period of 183 days between March 2010 and September 2010 we logged all relevant data and analyzed the ratings received by the Drupal feedback module. In total 8101 explicit ratings were collected on 5446 unique events.

Fig. 3.3 shows the distribution of the rating values for the 5-star rating mechanism. The distribution shifts towards the more positive values for both the dynamic and the static versions. We monitored the same outcome for the thumbs rating system where 88% $(= \frac{3349}{3795})$ of the ratings were thumbs up values.

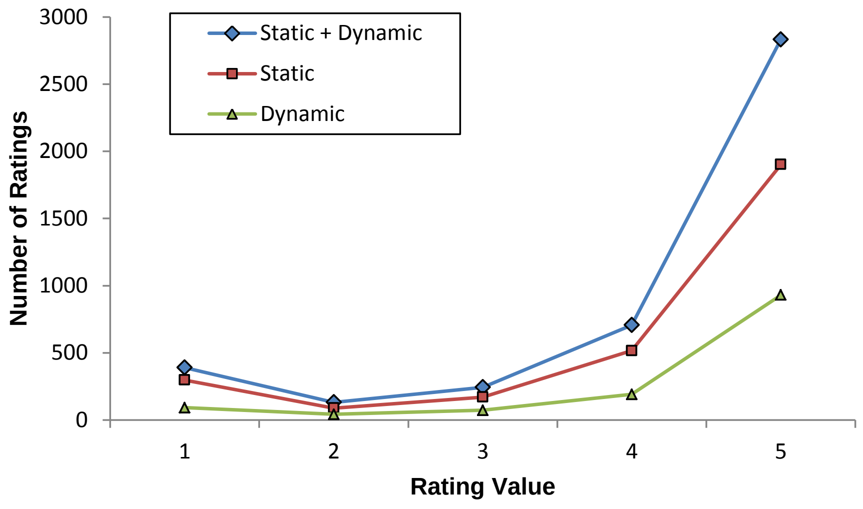


Figure 3.3: The distribution of the rating values for the 5-star rating system.

Table 3.1 depicts which explicit feedback mechanism collected the most feedback. We observe that the static 5-star rating mechanism is the most popular one, followed by the dynamic thumbs mechanism. The dynamic 5-star rating mechanism showed to be the least attractive one with less than half the ratings of its static version. The average number of ratings each system collected per day are for the dynamic 5-star, static thumbs, dynamic thumbs and static 5-star systems respectively 7, 9, 11 and 16. The differences between each of these systems are significant according to a one-tailed t-test, $p < 0.01$.

5-Star (dynamic)	Thumbs (static)	Thumbs (dynamic)	5-Star (static)
1330	1694	2101	2976
16%	21%	26%	37%

Table 3.1: The amount of ratings that each feedback system collected during the evaluation period of 183 days.

Fig. 3.4 visualizes the difference of the number of ratings collected from the static and dynamic feedback systems. We again observe that the static 5-star system processes the most ratings, whereas the static and dynamic versions of the thumbs rating system show a much smaller difference.

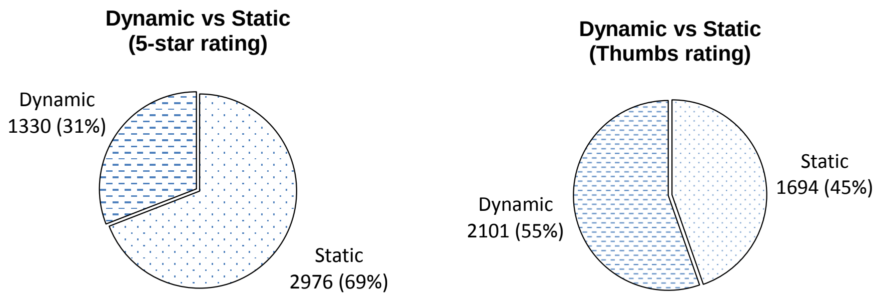


Figure 3.4: The amount of ratings that were given with either a dynamic or a static feedback system for the 5-star (left) and the thumbs up/down system (right).

In total 8101 ratings were collected and in total we logged $1,416,510$ (event detail) pageviews during the evaluation period. We can define a metric *feedback rate* as:

$$feedback\ rate = \frac{\#ratings}{\#pageviews} = \frac{8,101}{1,416,510} = 0.6\%$$

The feedback rate is an indicator of how actively a feedback system is used. While the general feedback rate of the experiment was 0.6% (i.e. 6 ratings for every thousand pageviews), the individual feedback rates for the 4 systems as shown from left to right in Table 3.1 are 0.37, 0.48, 0.59 and 0.84.

Since we allowed both anonymous users and registered users of the website to give feedback, we were also able to compare their rating behavior. In Table 3.2, a comparison is made between the feedback rates of anonymous users and logged-in users.

	Anonymous	Logged-in
Pageviews	1,395,289 (98.5%)	21,221 (1.5%)
Ratings	7,730 (95%)	371 (5%)
Feedback rate	0.55%	1.75%

Table 3.2: A comparison of the pageviews, ratings and feedback rate of anonymous users and users who were logged-in.

While we see that in absolute numbers most of the pageviews are originating from anonymous users (98.5%), we point out that in the end 5% of the ratings were still given by logged-in users. The resulting feedback rates are 1.75% for logged-in users and 0.55% for anonymous users.

To conclude we looked into a sparsity aspect of the given ratings. Between March 2010 and September 2010 there were on average approximately 30,000 events available on the website. Only 18% (=5446) of them were rated at least once. Of the 5446 different events that were rated, 23% (=1238) were rated more than once, the remaining 77% (=4208) in the tail was rated exactly once. Similar long-tailed effects were noted for the number of ratings per user: in total 7343 unique users were identified (using IP addresses), of which 94% (=6930) rated only once, 5% (=381) rated between 1 and 5 times, and only 0.4% (=32) users were found to be providing more than (or equal to) 5 ratings.

3.2.2.3 Discussion

Results showed that the static 5-star rating mechanism collected the most feedback, closely followed by the dynamic thumbs up/down system. This is somewhat unexpected because it was the oldest system and supposed to be the least attractive one. We assume this has in fact favored this system as it was easier recognizable as a feedback system (users are more familiar with this rating system).

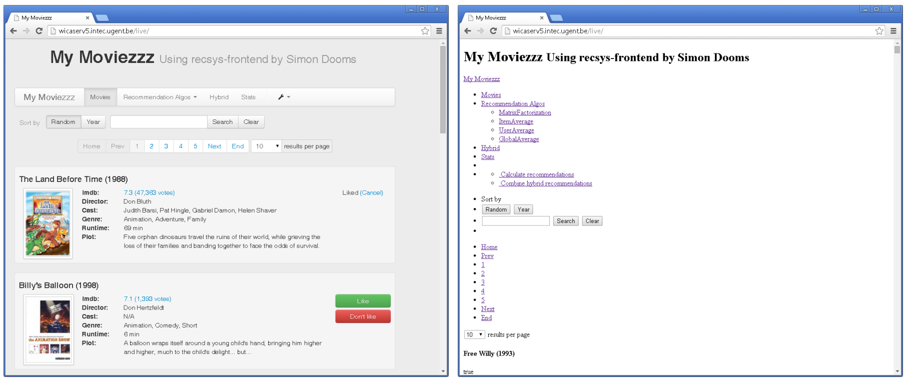
The 5-star systems failed however to produce more accurate feedback than the thumbs systems. Despite the fact that the items in our platform are events rather than movie content, we have seen that users interacted with the 5-star rating system in a similar manner as they did on the *youtube.com* site (before 2009) which is to rate using either very high or very low values. It is likely that users tend to give more positive feedback (e.g. higher rating values) because they only look at items that seemed appealing in the first place. Counterintuitive was that users do not seem to prefer the dynamic systems over the static ones.

The feedback rate of users who were logged-in was more than 3 times higher as the rate for anonymous users. Logged-in users seemed to be more actively involved and were more keen to provide explicit feedback. Still we think recommender systems should carefully consider what to do with anonymous users, as we saw that they generated 98.5% of all traffic in our experiment.

3.3 Recommendation Visualization

Visualization is everything. While often underestimated, the aspect of visually presenting results to users may even be the most important part of the recommendation process. It is the first thing users see and interact with and therefore it can be a defining factor for the user experience in total. Take for example any website and remove the layout. While the contents of the website will still be available, the lack of proper layout and design will render it useless. We illustrate this in Fig. 3.5; on the left, we show the HTML front end of what may be an online movie recommender system, on the right the same page without layout or design (i.e., without CSS files). All the same links and textual information are still available on the version without the layout, but it is clearly unusable in any practical sense.

While in the previous section we focused on the effects of the (layout



The Land Before Time (1988)

Imdb: 7.3 (47,363 votes)
Director: Don Bluth
Cast: Judith Barsi, Pat Hingle, Gabriel Damon, Helen Shaver
Genre: Animation, Adventure, Family
Runtime: 69 min
Plot: Five orphan dinosaurs travel the ruins of their world, while grieving the loss of their families and banding together to face the odds of survival.

Billy's Balloon (1998)

Imdb: 7.1 (1,393 votes)
Director: Don Hertzfeldt
Cast: N/A
Genre: Animation, Comedy, Short
Runtime: 6 min
Plot: A balloon wraps itself around a young child's hand, bringing him higher and higher, much to the child's delight... but...

Free Willy (1993)

true

Figure 3.5: Comparing a HTML front end of an online movie recommender system with (left) and without layout (right). While focus is often on content, the importance of layout and design should clearly not be underestimated.

of) feedback mechanisms, here we consider the user interaction process of presenting the actual results (i.e., recommendations) to users. We first consider some related work focusing on recommendation user interfaces and then present our own contributions regarding an in-home user interface for media content and finally our open-sourced recommendation front end framework.

3.3.1 Related Work

Most closely related to our work, is the TasteWeights system introduced by Bostandjiev et al. [79]. TasteWeights is a hybrid music recommender system with a very interactive and visual user interface. The authors performed a user study (32 participants) that compared different interactive and non-interactive hybrid strategies. The user interface of the TasteWeights system is composed out of three layers: a profile layer, context layer and recommendation layer. On each layer, users can adjust their tastes by interacting with slider weights. Using the sliders, the user preferences can be fine-tuned while the system provides dynamic recommendation feedback in real-time (i.e., the recommendations are updated). The results of the user study indicate that explanation and interaction with a visual representation of the hybrid system increases the user satisfaction (i.e., users like the system more) and relevance of the predicted content (i.e., the recommendations are better). A similar result was found by Gretarsson et al. [80] in a study of the SmallWorlds interactive graph-based interface.

It is well known that explanations increase user trust and can even increase user acceptance of a recommendation. Herlocker et al. [81] was among the first to experiment with showing users inside information about the recommendation process. They experimented (on MovieLens) with 21 different explanation types ranging from simply showing the average rating for a movie, to showing complex histograms of the calculated neighbors ratings. Their results indeed showed that explanation interfaces lead to an improved acceptance of the predicted rating. In [82] the authors elaborate on the possible goals of explanations. The system may try to convince users that a recommendation is a good recommendation (i.e., goal is promotion), or may assist them in assessing the actual quality of the recommendation (i.e., goal is satisfaction). The former can easily be implemented as a text label stating that 'Customers who bought this also bought...' while the latter requires more inside information about the recommendation process (e.g., 'Recommended item x because it contains the keywords *k1*, *k2*, *k3*'). Later, Tintarev [83] formalized even more aims for explanations including system transparency, scrutability, efficiency, etc. Each may require a different approach and implementation. She also proposes the idea that explanations may even be personalized themselves in the sense that different users may benefit from different manners of explaining the recommender system.

So user interfaces for recommendation results should preferably be interactive and provide explanations of some sort. Regarding the actual representation of the recommended items themselves, literature is mostly focusing on typical *top-n* list design. Some alternative recommendation user interfaces do exist such as the circle-based CoFeel [84] or TopicLens [85] systems, graph-based systems [80, 86, 87], or critiquing-based systems [88]. Very often however, they are specifically designed towards a specific item domain or recommendation scenario and because of their added complexity (in contrast to a simple list approach) their suitability is restricted to advanced users.

3.3.2 An In-Home Recommender System: Use Case Study

In this section we discuss the use case of the OMUS system developed in the context of the OMUS iMinds[3] research project. The Optimized

[3]`http://www.iminds.be/en/projects/2014/03/06/omus`

MUltimedia Service (or OMUS) system aimed to offer an overall and integrated information system to overcome typical information handling problems in home environments.

Media content in home environments is often scattered across multiple devices in the home network. Aside from suffering from information overload, home users are therefore often experiencing technical difficulties to connect their content with a device of their choice e.g., streaming a movie that is actually stored on a laptop to the television. To address these home-specific issues, the OMUS system was proposed which included an optimized content aggregation framework, a hybrid group-based contextual recommender system, and an overall web-based user interface making both content and recommendations available for all devices across the home network. The system was evaluated by means of focus group interviews. Different target groups were integrated in the study, including students, heavy downloaders, regular interactive digital television viewers and people with an extensive media set-up at home.

Here, we will focus on the user interface that was developed for the OMUS system and its requirements as suggested by the user study. We first overview these requirements, then present a high-level view of the recommender system and finally detail the user interface and its features. For more detailed information about the OMUS project and its other components we refer to our paper [89].

3.3.2.1 User Study: Requirements

The user study was performed in two stages, once before designing the system and once after. By means of 7 focus group interviews totaling 47 respondents, the user requirements and typical user behavior in a home environment could be analyzed. When users were asked how they find interesting (media) content, it turned out they focus mostly on content attributes. For movies e.g., the 'movie director' or 'genres' would be an indicator of their potential interest. Interestingly however, no consensus could be found as to what attributes were the most important. For some users the director was the most important movie attribute, while for others the IMDb rating was more valued. User preferences also seemed to be heavily context sensitive i.e., depending on the day, mood and whether they were consuming media in group or by themselves. For group situations, users indicated they often found it hard to find suitable content items for everyone involved i.e., reaching a group-based consensus.

Users were asked about their willingness to engage in searching and selecting content. Again very divergent results were noted. While some users appeared very willing to actively engage with an information system in search of interesting content, others just wanted to 'sit back and relax'. So a home recommender system would need to accommodate for both types of users (i.e., active and passive users).

The two major challenges revealed by the user study were the duality between active and passive users, and the context dependency. A home recommender system (and its user interface) should find an acceptable balance between accommodating both active and passive users in multiple contexts. In the following list we summarize some of the user requirements for both the in-home content recommender system and its user interface.

OMUS recommender system requirements

- Support for group recommendation.
- Contextual user preferences.
- Support for different types of user engagement: active and passive users.

OMUS user interface requirements

- Allow (optional) explicit user feedback.
- Show only basic item information, show more if requested.
- Allow to filter lists on genre.
- Allow users to control their user profiles.

3.3.2.2 The OMUS System

Here we detail the OMUS system that was built to meet the requirements as defined in the previous section. An external (out-of-home) recommendation service was developed allowing group recommendation which synchronized with a light-weight, in-home OMUS client instance. The system used the DLNA[4] standard to support cross-device media rendering support and the OMUS user interface (UI) was designed to be (HTML) web-based to be supported across the wide range of devices that may be available inside a typical home network (e.g., laptops, tablets, smartphones, smart televisions, etc.).

[4]`http://www.dlna.org`

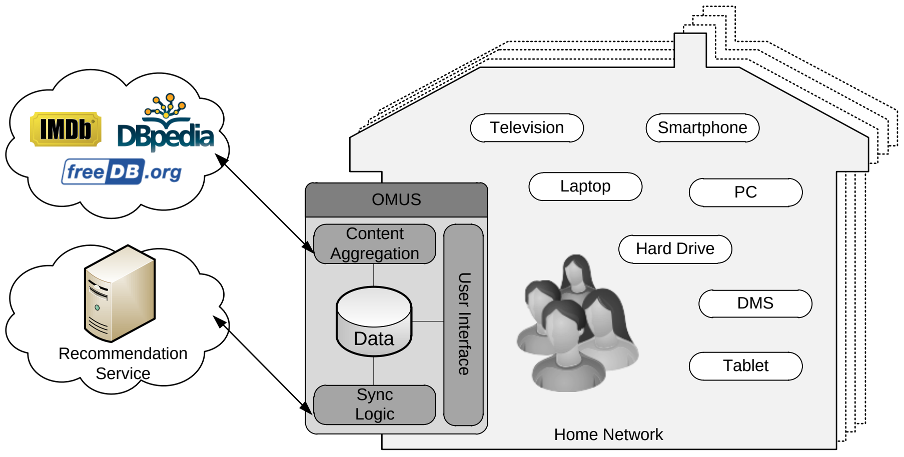


Figure 3.6: The high-level view of the OMUS system integrated in a home environment with multiple users and distinct devices.

The home network (Fig. 3.6) houses several users that each interact with a number of distinct interlinked devices. The OMUS information system, is situated at the border of the home network. The system includes components as content aggregation, a recommender system, a user interface and a central data component. The possibility of multiple houses each implemented with the OMUS system (and therefore using the same recommendation service) is graphically illustrated in the architecture by the dotted house outlines extending the central home network.

The content aggregation component centrally gathers all the information about the media content that is available to home users (in-home media), complemented with data about other consumable media (e.g., content in a friend's home, online sources). The metadata stored about an item can be used for content-based recommendation approaches.

The sync logic component is responsible for synchronization services between the home network and an external recommendation service. The external service provides the necessary recommendation functionality without imposing new (computing) hardware requirements inside the home network.

The UI actively interfaces between users, devices and the OMUS information system. All aggregated content in the home network is integrated into a single content overview list and made available through the UI.

Extra information about items (e.g., plot, genre, etc. for movies) is easily accessible, playback functionality is provided and user preferences can be specified. A recommendation list tailored specifically towards any provided context additionally assists users in their content selection process.

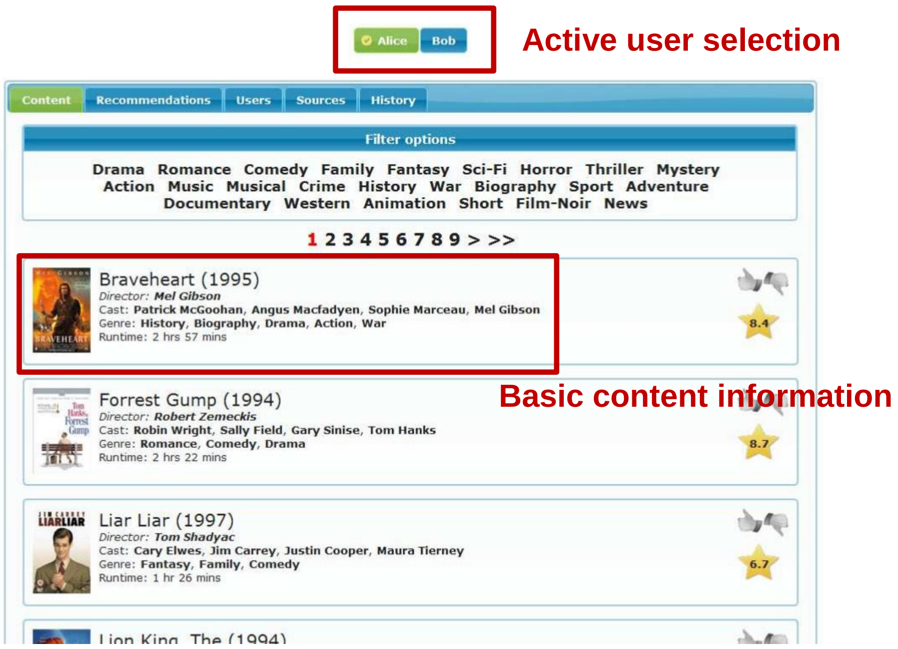
3.3.2.3 The OMUS User Interface

From the user studies, we learned that users may have very diverse requirements towards how they wish to interact with a home information system. In particular, the way in which they were willing to provide feedback (e.g., provide ratings for media) was very user dependent. The user interface was designed to handle these situations while providing the functionality to browse content, control media (i.e., play, pause, and stop media) and to tailor recommendation lists to any given context. Because the user interface must be easy to use and intuitively to work with, many of the design concepts resemble in style and behavior to that of common web applications to which users may already have an affiliation with (e.g., IMDb, YouTube, etc.). The user interface was designed to be web-based (mainly HTML and JavaScript) to make it accessible through any web-enabled device present in the home network.

Browsing and Interacting

All content available in the home network is aggregated into one overview list. For this list to become available, a set of active users must be specified to the system. Active users are users that want to participate in the same session of media consumption. Showing content only after the user selection allows for possible security policies to be enforced towards the accessibility of content (e.g., some content may not be suited for children). Content may be restricted to the minimal set of items that every user in the active user set is allowed to access.

The content overview list should contain all relevant information but at the same time also remain simple to use and easy to access. To meet these requirements we propose a two-leveled hierarchical overview. At the first level, only basic information (e.g., title, director, cast, genre and runtime) about content items is shown (Fig. 3.7). The basic information level offers a quick overview of the available content. When a specific content item is selected from the list, more detailed information is shown (Fig. 3.8), e.g., the plot in the case of a movie item type or web links to external sources (e.g., IMDb) with extra information such as reviews or trailers.



Drama Romance Comedy Family Fantasy Sci-Fi Horror Thriller Mystery
Action Music Musical Crime History War Biography Sport Adventure
Documentary Western Animation Short Film-Noir News

1 2 3 4 5 6 7 8 9 > >>

Braveheart (1995)
Director: Mel Gibson
Cast: Patrick McGoohan, Angus Macfadyen, Sophie Marceau, Mel Gibson
Genre: History, Biography, Drama, Action, War
Runtime: 2 hrs 57 mins
8.4

Forrest Gump (1994)
Director: Robert Zemeckis
Cast: Robin Wright, Sally Field, Gary Sinise, Tom Hanks
Genre: Romance, Comedy, Drama
Runtime: 2 hrs 22 mins
8.7

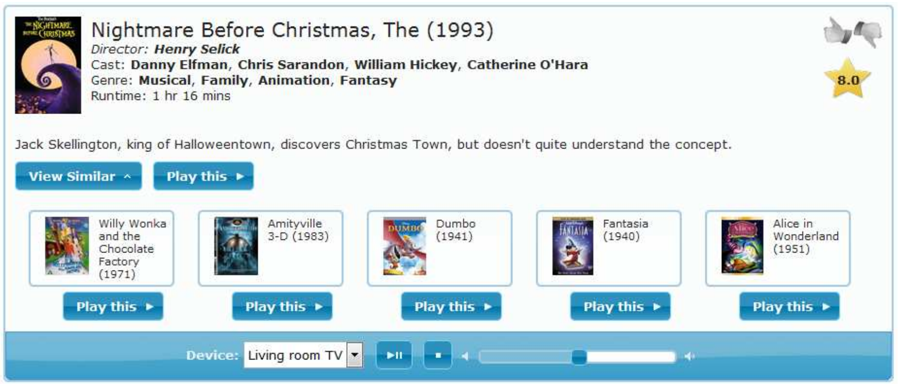
Liar Liar (1997)
Director: Tom Shadyac
Cast: Cary Elwes, Jim Carrey, Justin Cooper, Maura Tierney
Genre: Fantasy, Family, Comedy
Runtime: 1 hr 26 mins
6.7

Lion King, The (1994)

Figure 3.7: The basic content overview list. All media content discovered in the network will be listed here. At the top, active users can be specified and genre filters can be selected. Basic item information is available together with a thumbs up/down feedback system per item.

Aside from showing extra item information, the item-specific view can also be used to provide controls and tools to interact with the content. For every available item, similar items can be displayed (Fig. 3.8). These similar items are calculated by the recommender system and originate from the same content pool as shown in the content overview list. Consequently, every item displayed in the similar items list can in turn be interacted with.

The most interesting way of interacting with media content is by actually consuming it (i.e., listening to music or watching video). This is supported by the user interface by means of the media control buttons on the bottom of the item-specific information view (Fig. 3.8). Clicking the *Play this* button will trigger UPnP AV *SetAVTransportURI* and *Play* messages to be sent to the currently selected device in the device selection box. These messages cause the DLNA Digital Media Renderer



Nightmare Before Christmas, The (1993)
Director: ***Henry Selick***
Cast: **Danny Elfman, Chris Sarandon, William Hickey, Catherine O'Hara**
Genre: **Musical, Family, Animation, Fantasy**
Runtime: 1 hr 16 mins

Jack Skellington, king of Halloweentown, discovers Christmas Town, but doesn't quite understand the concept.

Willy Wonka and the Chocolate Factory (1971)
Amityville 3-D (1983)
Dumbo (1941)
Fantasia (1940)
Alice in Wonderland (1951)

Device: Living room TV

Figure 3.8: The item-specific content view. Additional information about the media item is provided (e.g., movie plot) together with similar items and some media interaction buttons.

(DMR) on the selected device to start buffering and playing the concerning multimedia content item. The device selection box is automatically populated with devices discovered in the home network that announced themselves as being a DLNA DMR. The user interface displays every available content item in the network in one interactive list in which content information can be shown, as well as interaction functionality is provided to allow the playback of every media item on every capable device in the home network.

Providing Feedback

To overcome the media overload problem, a recommender system was integrated in the user interface. To enable such recommendations, first user feedback must be collected. Through the user interface three distinct types of feedback are collected: ratings, explicit item attribute feedback and implicit media consumption behavior.

The first, is the gathering of explicit user ratings by means of the thumbs up/down widget available on the basic information display of every item (Fig. 3.7). With this widget, users can straightforwardly express their either positive or negative preference towards any specific content item in the system. The thumbs feedback system is a very intuitive and easy to use feedback system which is usable across a number of different input devices including touch devices (e.g., smartphones and tablets). The thumbs feedback system was therefore chosen over the more commonly used 5-star rating system.

The thumbs up/down feedback system allows users to express preferences on a binary scale. The user study revealed that some users are willing to put more effort into providing feedback than others and would like an increased level of control over their user profiles. To meet these requirements, an additional level of explicit feedback was introduced. Aside from liking or disliking an item in the overview list, users are also able to express their preference on a finer scale more specifically towards an item attribute. When an item attribute in the content overview list is selected, a popup window (Fig. 3.9) will allow (thumbs up/down) user feedback towards the relevant attribute. For an item of the movie type, specific user feedback can be provided towards directors, cast and genre by selecting the relevant attributes in the content overview list.

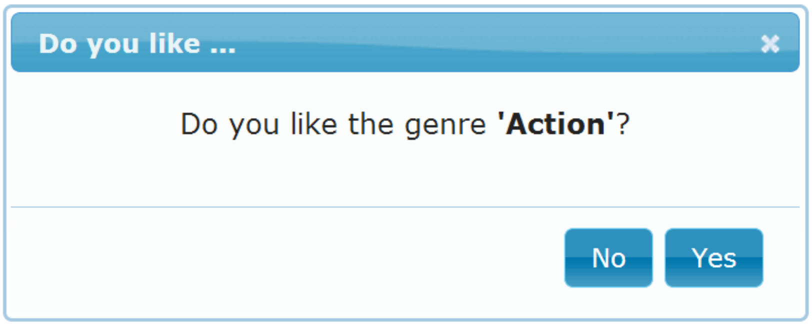

Do you like the genre **'Action'**?

Figure 3.9: An example of the popup window that is shown when an item attribute in the content overview list is clicked. Binary explicit feedback can be provided.

Users are thus able to explicitly express their preferences on either items or specific item attributes. As the user study revealed, some users are actually unwilling to provide any form of explicit feedback. They expect the system to learn without manually specifying likes and dislikes. Therefore implicit feedback is collected from user interactions via the user interface: the consumption of a media item itself is regarded as a positive preference towards that item. If a user listens to a song, or watches a movie, the system will infer a positive relationship between that user and the item. Since the user interface serves as the aggregated entry point for media control in the home network, the consumption of media and all its available properties (e.g., duration of consumption, time of consumption, etc.) are easily logged for every user of the system.

To allow users control over their system-constructed profiles, the user interface also offers a user-specific history view. The history view lists all relevant feedback that the system has collected about the user and may be used as input for the recommendation algorithms. In the history view, users can view what information is gathered and unwanted entries can easily be deleted.

In conclusion, there are three ways in which the system tries to collect information about the user. Two types of explicitly providing likes and dislikes in combination with implicitly inferring information from media consumptions provide an adequate feedback framework to support both active and passive users of the system.

Contextual Recommendation List

When all user feedback is processed and recommendations are calculated, a list with suggested items is available in the user interface. Recommended content items share the same visualization as normal content, providing the same information, similar items list, rating functionality and media control buttons. Since the user should not be overloaded with recommended items (as may be the case for the content items), the system selects the top-N (between 7 and 10) most interesting items for the set of currently active users.

When the active users change, the recommendation list instantly updates its items accordingly in real-time. That way, the system is capable of providing recommendations for single users (i.e., when only one user is indicated as active) as well as for groups of users (i.e., multiple active users). The recommended items list for groups of users aims to be a best estimation of the top-N items that will be liked by (and preferably not already consumed by) all the active users.

The user study indicated that, when people are deciding what movie to watch in group, not every member contributes equally to the final decision. Some users might be indifferent and do not really care what will be watched while others may have a really strong opinion, or are more knowledgeable about the media at hand. To be able to realistically model these situations, user weights were introduced in the system. When a new user account is created, an appropriate weight value (indicated as *importance*) can be set. Three importance weights are available in the user interface (Fig. 3.10): *low*, *medium* and *high* (indicated respectively by the symbols -, ? and +). These weights represent for each user how much its user profile should be taken into account when generating recommendations for groups of users. The availability of these user importance weights allows the system to adapt its recommended items list to very specific user situations (e.g., two parents and a child, or four friends of which one has an expert opinion on movies). By manually changing the active users and their importance weights, the recommended items list can be influenced in real-time to make good suggestions for every possible user context.

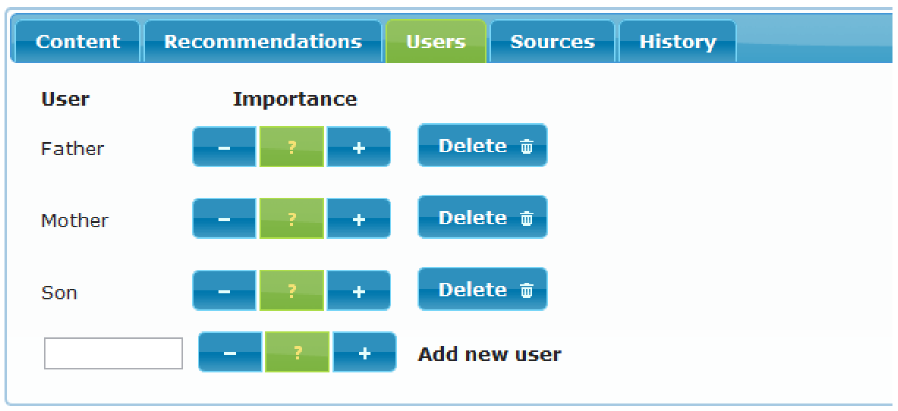


Figure 3.10: The *Users* tab of the user interface. Every user in the OMUS system is associated with an importance factor that can be changed according to the desired context.

The user study revealed that users often already have a specific genre or category in mind when searching for some media to consume. To enable this parameter in the system, genre filters were introduced. The genre filter allows to restrict the recommendation list to items of the indicated genre. Together with the ability to change the active users and set their importance weights, users are able to provide a fine grained context situation to which the recommender system can specifically tailor its suggestions. By instantly updating the list when a context parameter is changed, a feeling of real-time interactivity is achieved between the user and the system, which may boost user engagement and in the end can lead to higher-quality recommendations.

3.3.3 Recsys Front End

The recommender systems domain has matured greatly over the past 20 years. Particularly in the last 5 years a lot of work has been focusing on software libraries implementing standardized recommendation algorithms (e.g., MyMediaLite [34], LensKit [35], Apache Mahout[5], etc.) and evaluation metrics (e.g., RiVal[6]) to reduce implementation effort for researchers and increase repeatability of experiments and obtained research results in general. What the domain is currently still lacking however is open-source support for user interfaces for recommender systems. Researchers wanting to visually inspect calculated recommendation results still need to build their own user interfaces. Therefore often researchers resort to offline calculated metrics only to represent their recommendation quality rather than actually visualizing and illustrating the results (to users).

In this section we present our own 'Recsys front end', which is a generalized HTML-based front end for movie recommender systems. The framework allows to easily visualize (movie) recommendation results and integrates a wide range of well-known recommendation algorithms of which the results can also be combined into a hybrid recommendation list. The framework builds on the MyMediaLite software library and integrates our own (see previous chapter) MovieTweetings dataset for the simulation of user feedback. We have open-sourced this framework on the Github platform[7].

3.3.3.1 Architecture and Installation

The Recsys front end is designed to run on a LAMP installation (i.e, Linux, Apache, MySQL, PHP) which is a very common and easy to set up webserver configuration. After installing the necessary files from the Github project, default scripts are available to set up the database and basic configuration.

The project is composed of a web-based user interface that interfaces with a MySQL database running on the webserver. A Python interface interacts with the MyMediaLite software library to provide recommendation support. At installation time the latest version of the MovieTweetings dataset is imported to bootstrap the system with both an item catalog (i.e., all rated movies in the dataset) and user feedback of thousands of users.

[5]`https://mahout.apache.org`
[6]`https://github.com/recommenders/rival`
[7]`https://github.com/sidooms/Recsys-frontend`

3.3.3.2 Browsing and Rating

Fig. 3.11 represents the front end home screen. When the front end is loaded, the complete item catalog is shown. Similar to other information systems, users are allowed to browse content and inspect content attributes. To reduce the webserver load, the movie content attributes as Director, Cast, Genre, etc. are not stored in the database. Only the corresponding IMDb id and title are actually stored, other attributes are loaded dynamically by client-side JavaScript code which accesses the public OMDb API.

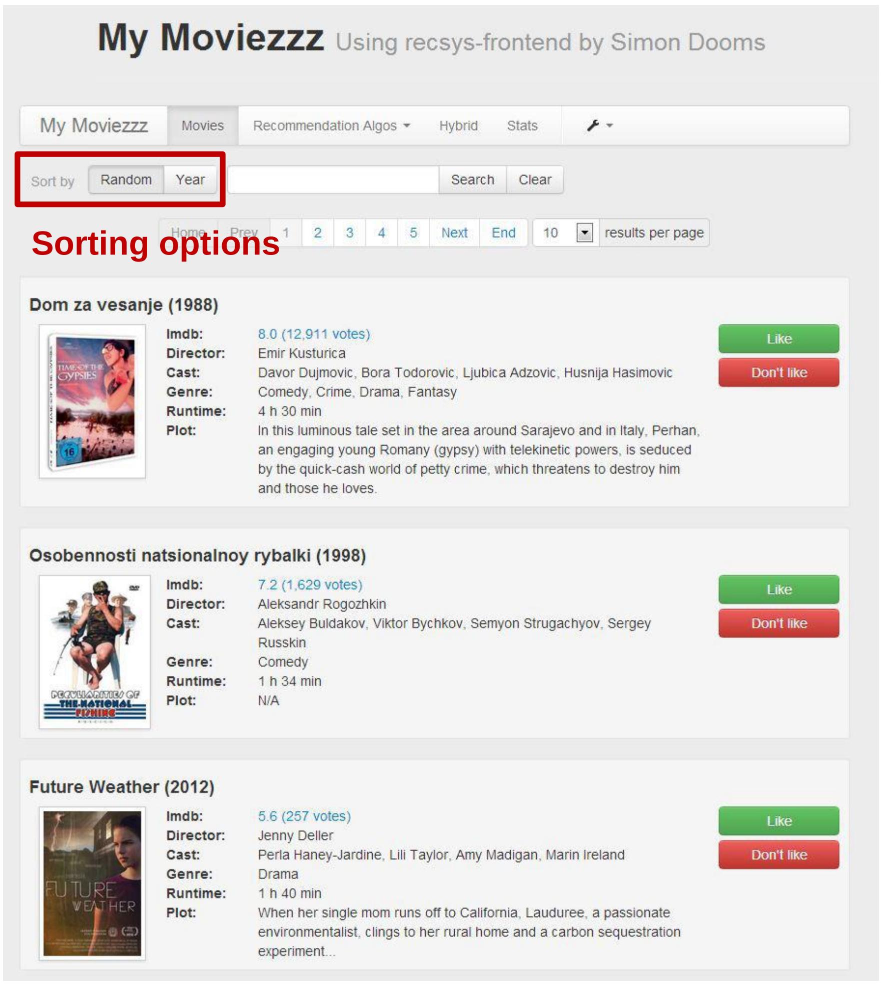
Content items are shown in a typical list-view and for each movie a *Like* and *Dislike* button is available to allow users to indicate their preferences to the (recommender) system. Items are by default ordered randomly to prevent typical popularity biases, but can also be sorted by their release date. A simple search form allows to search for particular movies using parts of the title (which is stored in the database) as search query (as illustrated in Fig. 3.12).

3.3.3.3 Calculating Recommendations

When ratings on items are provided, recommendations can be calculated. The calculation process can be triggered by clicking on the 'Calculate recommendations' link (Fig. 3.13). Doing so will trigger the webserver to start some background processes that provide the collected user feedback and item catalog to the installed MyMediaLite library and retrieve recommendation results which are then stored in the webserver database. The specific recommendation algorithm that should be executed can be set in a configuration file. If multiple algorithms are set, multiple recommendation processes will be run and their results saved in the database.

When the calculation process is finished, the user is notified and recommendation results can be inspected in the user interface. As illustrated in Fig. 3.14, the 'Recommendation Algos' tab at the top of the page allows to dynamically switch between the recommendation results of the different calculated recommendation algorithms. Such a feature is very useful for research purposes as it allows to quickly (visually) compare the results for different algorithms and approaches.

Recommended movies are visualized in the same way as other movies from the item catalog. Content attribute information is shown and users can again rate the movies which allows the system to further refine the results. The visualization of a recommended movie does integrate how-



Dom za vesanje (1988)

Imdb: 8.0 (12,911 votes)
Director: Emir Kusturica
Cast: Davor Dujmovic, Bora Todorovic, Ljubica Adzovic, Husnija Hasimovic
Genre: Comedy, Crime, Drama, Fantasy
Runtime: 4 h 30 min
Plot: In this luminous tale set in the area around Sarajevo and in Italy, Perhan, an engaging young Romany (gypsy) with telekinetic powers, is seduced by the quick-cash world of petty crime, which threatens to destroy him and those he loves.

Osobennosti natsionalnoy rybalki (1998)

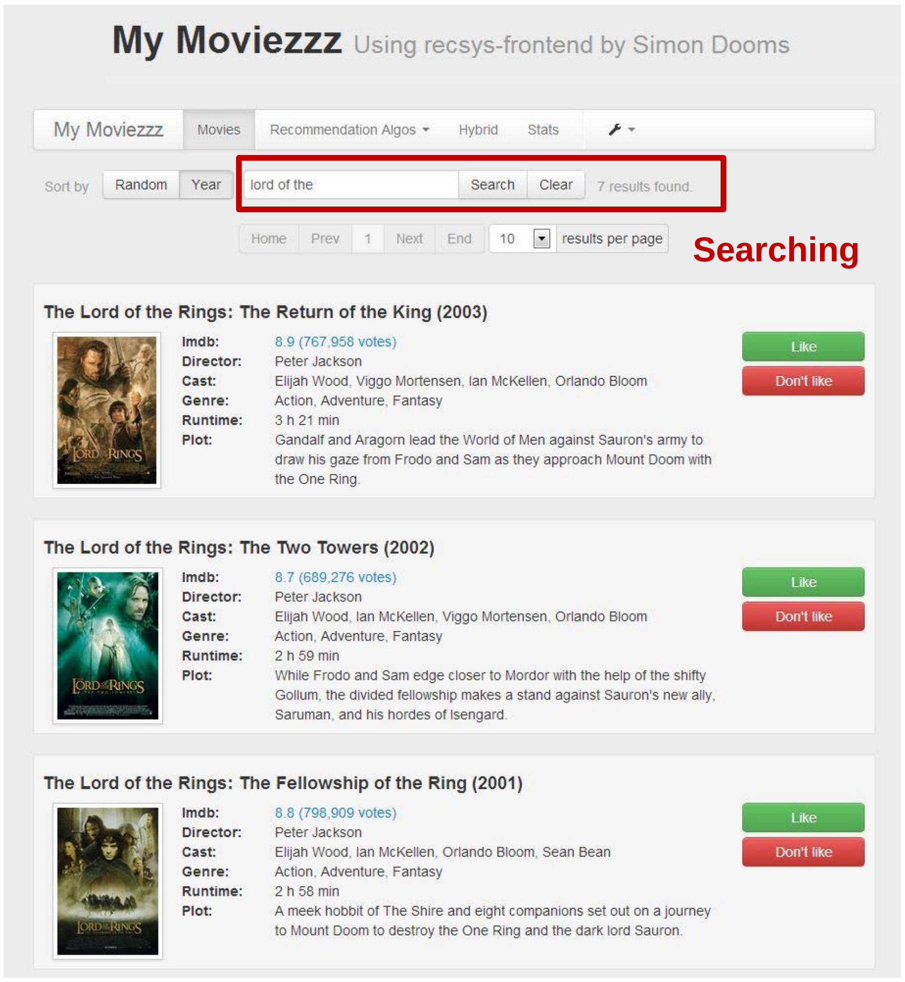
Imdb: 7.2 (1,629 votes)
Director: Aleksandr Rogozhkin
Cast: Aleksey Buldakov, Viktor Bychkov, Semyon Strugachyov, Sergey Russkin
Genre: Comedy
Runtime: 1 h 34 min
Plot: N/A

Future Weather (2012)

Imdb: 5.6 (257 votes)
Director: Jenny Deller
Cast: Perla Haney-Jardine, Lili Taylor, Amy Madigan, Marin Ireland
Genre: Drama
Runtime: 1 h 40 min
Plot: When her single mom runs off to California, Lauduree, a passionate environmentalist, clings to her rural home and a carbon sequestration experiment...

Figure 3.11: Screenshot of the home screen of the recsys front end framework. The item catalog can be browsed as a list sorted either randomly or by year. For each item (i.e., movie) thumbs up/down based feedback buttons allow users the indicate their preferences.

ever one additional feature: the predicted recommendation value. Since the integrated recommendation algorithms in the Recsys front end system are rating-based (i.e., they try to predict the rating for items based on previous ratings) every recommended movie is associated with a predicted rating. This prediction is shown together with the content attributes (Fig. 3.14) and may support debugging or serve as an explanation for the recommendation.



The Lord of the Rings: The Return of the King (2003)

Imdb:	8.9 (767,958 votes)
Director:	Peter Jackson
Cast:	Elijah Wood, Viggo Mortensen, Ian McKellen, Orlando Bloom
Genre:	Action, Adventure, Fantasy
Runtime:	3 h 21 min
Plot:	Gandalf and Aragorn lead the World of Men against Sauron's army to draw his gaze from Frodo and Sam as they approach Mount Doom with the One Ring.

The Lord of the Rings: The Two Towers (2002)

Imdb:	8.7 (689,276 votes)
Director:	Peter Jackson
Cast:	Elijah Wood, Ian McKellen, Viggo Mortensen, Orlando Bloom
Genre:	Action, Adventure, Fantasy
Runtime:	2 h 59 min
Plot:	While Frodo and Sam edge closer to Mordor with the help of the shifty Gollum, the divided fellowship makes a stand against Sauron's new ally, Saruman, and his hordes of Isengard.

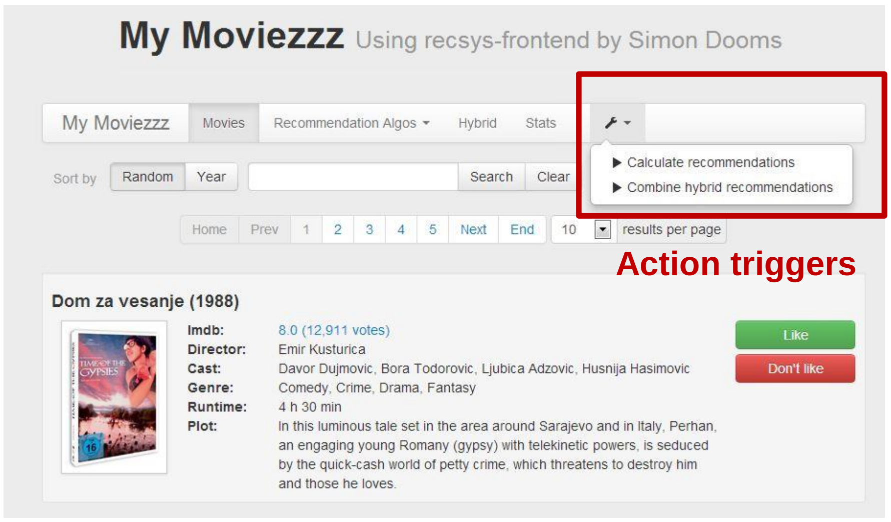
The Lord of the Rings: The Fellowship of the Ring (2001)

Imdb:	8.8 (798,909 votes)
Director:	Peter Jackson
Cast:	Elijah Wood, Ian McKellen, Orlando Bloom, Sean Bean
Genre:	Action, Adventure, Fantasy
Runtime:	2 h 58 min
Plot:	A meek hobbit of The Shire and eight companions set out on a journey to Mount Doom to destroy the One Ring and the dark lord Sauron.

Figure 3.12: Screenshot of the search form of the recsys front end framework. Movies can be searched on (parts of) their title.

3.3.3.4 Combining Hybrid Recommendations

The 'Recsys front end' was originally designed to aide in the evaluation of hybrid recommender system experiments. Therefore the system also includes a hybrid recommendation feature. After the individual algorithms are calculated, a hybrid recommendation list can be composed by the system. For the purpose of the framework a simple weighted hybrid strategy [37] was implemented, but more advanced and interesting hybridization strategies could easily be integrated in the framework.



Dom za vesanje (1988)

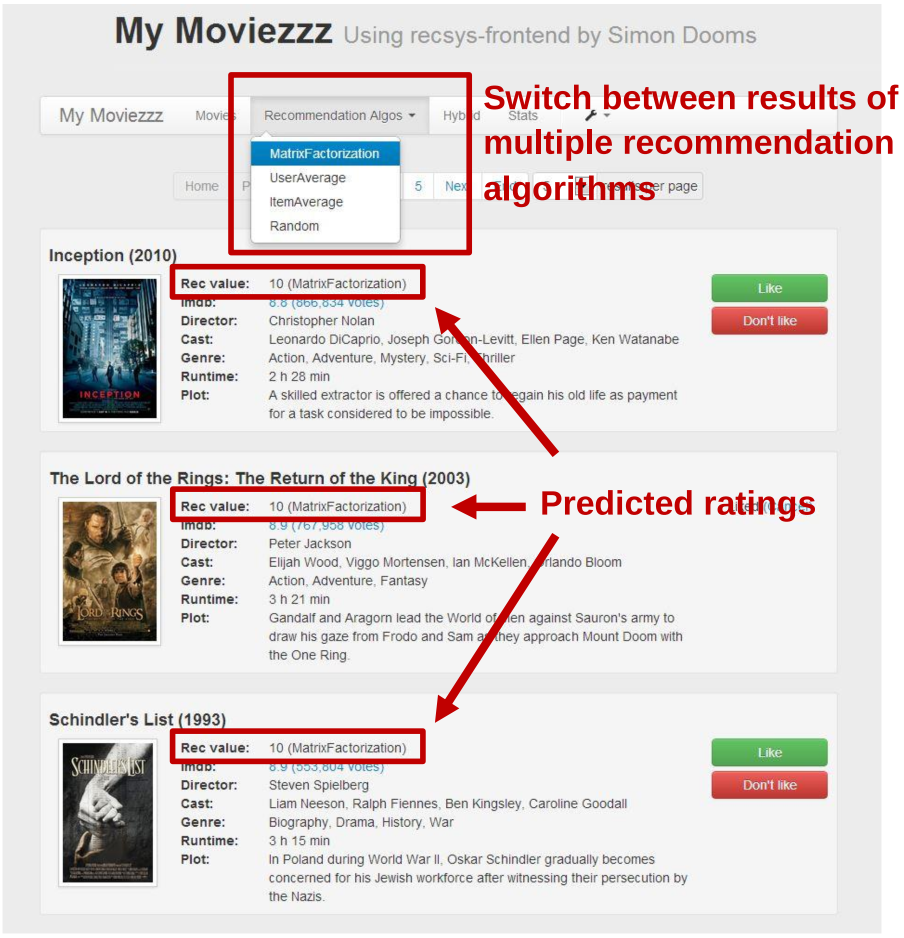
Imdb:	8.0 (12,911 votes)
Director:	Emir Kusturica
Cast:	Davor Dujmovic, Bora Todorovic, Ljubica Adzovic, Husnija Hasimovic
Genre:	Comedy, Crime, Drama, Fantasy
Runtime:	4 h 30 min
Plot:	In this luminous tale set in the area around Sarajevo and in Italy, Perhan, an engaging young Romany (gypsy) with telekinetic powers, is seduced by the quick-cash world of petty crime, which threatens to destroy him and those he loves.

Figure 3.13: Screenshot of the calculation related buttons of the recsys front end framework. The processes of calculating recommendations or combining existing results in a hybrid list can easily be triggered at the front end.

De 'Stats' tab in the user interface (Fig. 3.15) allows easy and visual interaction with the hybrid configuration settings. For every active recommendation algorithm in the system a slider is available. The sliders represent the individual contribution of the algorithms to the final hybrid recommendation result for a weighted average scenario. By increasing the slider value for a certain algorithm, the prediction score of that algorithm will be taken more into account for the final hybrid recommendation value. A pie chart representing the total contribution of each of the algorithms to the hybrid recommendation value dynamically updates when slider values are changed.

When the individual weights of the algorithms are configured, the hybrid calculation process can be triggered by clicking the 'Combine hybrid recommendations' link (Fig. 3.13). Python-based background processes will be started that combine the individual recommendation results (already available in the database) in a hybrid list using the weights as configured. The end result, a hybrid recommendation list, is saved in the database when the process is completed.

The hybrid recommendation results can be visually inspected in the 'Hybrid' tab at the top of the front end (Fig. 3.16). The hybrid recommendation results are again visualized as a browsable top-N movie



Inception (2010)

Rec value: 10 (MatrixFactorization)
Imdb: 8.8 (866,834 votes)
Director: Christopher Nolan
Cast: Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen Page, Ken Watanabe
Genre: Action, Adventure, Mystery, Sci-Fi, Thriller
Runtime: 2 h 28 min
Plot: A skilled extractor is offered a chance to regain his old life as payment for a task considered to be impossible.

The Lord of the Rings: The Return of the King (2003)

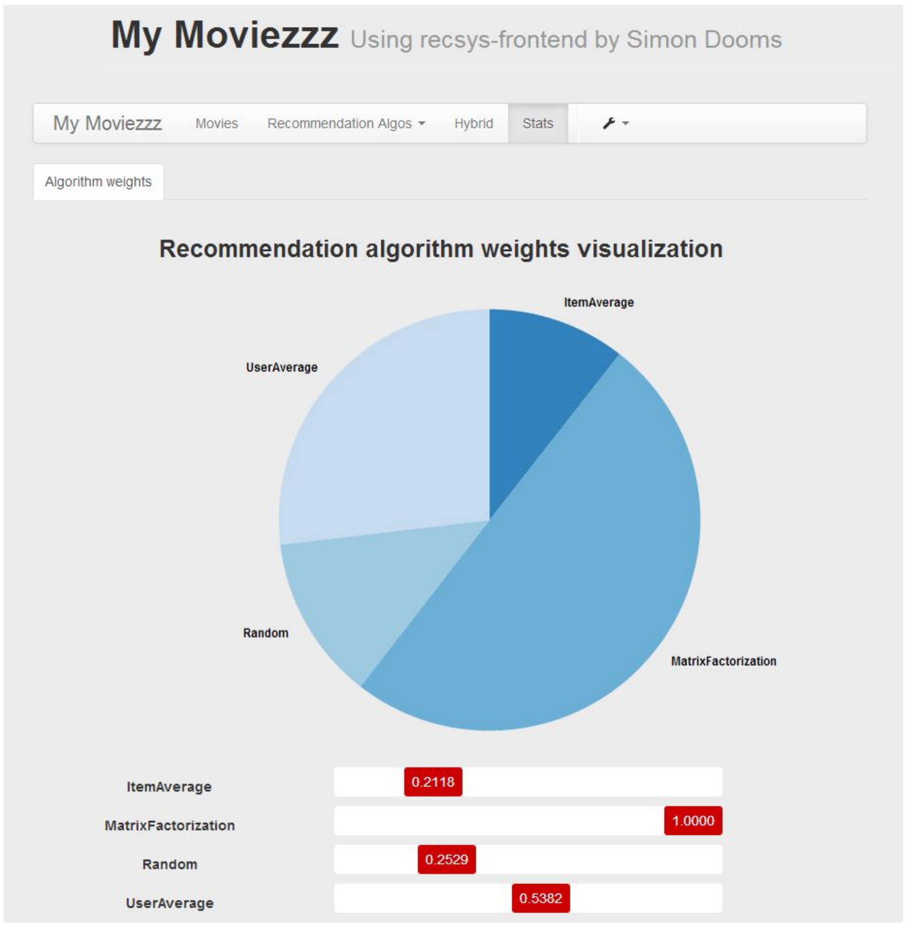
Rec value: 10 (MatrixFactorization)
Imdb: 8.9 (767,958 votes)
Director: Peter Jackson
Cast: Elijah Wood, Viggo Mortensen, Ian McKellen, Orlando Bloom
Genre: Action, Adventure, Fantasy
Runtime: 3 h 21 min
Plot: Gandalf and Aragorn lead the World of Men against Sauron's army to draw his gaze from Frodo and Sam as they approach Mount Doom with the One Ring.

Schindler's List (1993)

Rec value: 10 (MatrixFactorization)
Imdb: 8.9 (553,804 votes)
Director: Steven Spielberg
Cast: Liam Neeson, Ralph Fiennes, Ben Kingsley, Caroline Goodall
Genre: Biography, Drama, History, War
Runtime: 3 h 15 min
Plot: In Poland during World War II, Oskar Schindler gradually becomes concerned for his Jewish workforce after witnessing their persecution by the Nazis.

Figure 3.14: Screenshot of the recommendation results of the recsys front end framework. Switching between the results of different recommendation algorithms allows easy visual comparison of the results.

list. The final hybrid recommendation score (as calculated using the user-set weights at the 'Stats' tab) is available and more interestingly inspectable. For every recommended movie, next to the predicted rating value a 'Show explanation' link is available. Clicking on the link will expand a hidden information panel which reveals detailed statistics about how the hybrid score was actually calculated. This feature may again be of great value for researchers or recommender systems' administrators to debug and understand the hybrid recommendation process.

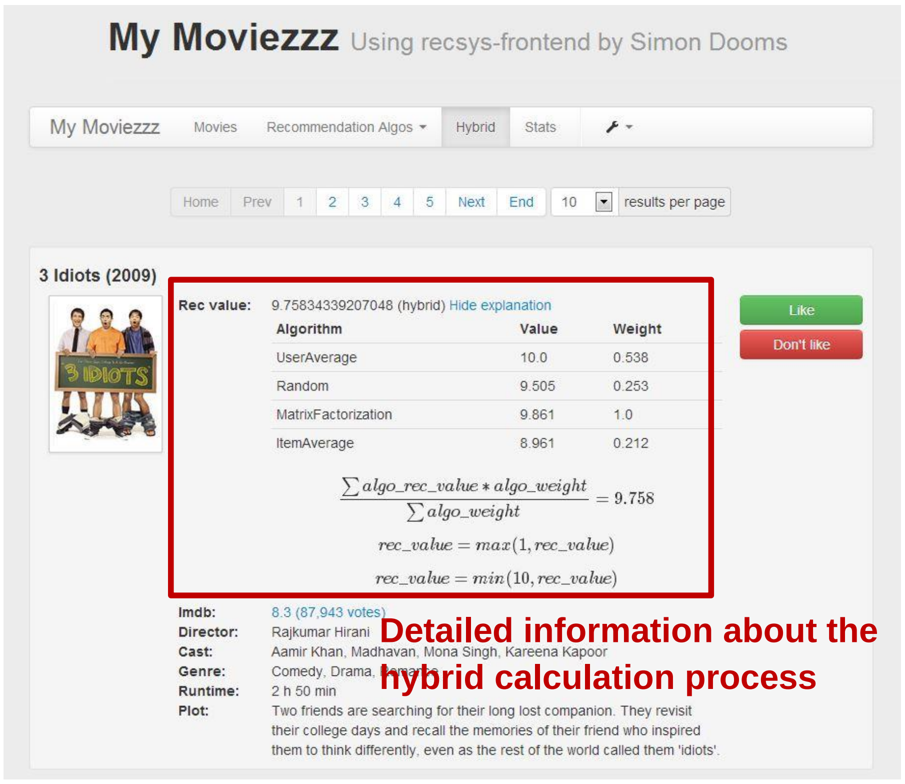


ItemAverage	0.2118
MatrixFactorization	1.0000
Random	0.2529
UserAverage	0.5382

Figure 3.15: Screenshot of the hybrid configuration page of the recsys front end framework. For every individual recommendation algorithm, interactive sliders are available representing the weighted contribution to the final hybrid recommendation.

3.3.3.5 Discussion

The recsys front end was originally designed out of a personal need for visual inspection of recommendation results of the many experiments described in this work. We hope however that by open-sourcing it, it may serve other researchers as a quick and easy to set up visual portal for recommendation results. Note that this framework is intended for academic and research-based usage, scalability and security aspects are not considered. The framework is also not suited (at least not without some


My Moviezzz Using recsys-frontend by Simon Dooms
My Moviezzz
Movies
Recommendation Algos
Hybrid
Stats
Home
Prev
1
2
3
4
5
Next
End
10
results per page
Like
Don't like
Detailed information about the hybrid calculation process

3 Idiots (2009)

Rec value: 9.75834339207048 (hybrid) Hide explanation

Algorithm	Value	Weight
UserAverage	10.0	0.538
Random	9.505	0.253
MatrixFactorization	9.861	1.0
ItemAverage	8.961	0.212

$$\frac{\sum algo\_rec\_value * algo\_weight}{\sum algo\_weight} = 9.758$$

$$rec\_value = max(1, rec\_value)$$

$$rec\_value = min(10, rec\_value)$$

Imdb:	8.3 (87,943 votes)
Director:	Rajkumar Hirani
Cast:	Aamir Khan, Madhavan, Mona Singh, Kareena Kapoor
Genre:	Comedy, Drama, Romance
Runtime:	2 h 50 min
Plot:	Two friends are searching for their long lost companion. They revisit their college days and recall the memories of their friend who inspired them to think differently, even as the rest of the world called them 'idiots'.

Figure 3.16: Screenshot of the hybrid recommendation results of the recsys front end framework. For every hybrid recommend movie a hidden information panel can be made visible to detail the actual hybrid calculation process.

modifications) to serve as the front end of a final recommender system exposed to real users. While allowing users to dynamically browse multiple recommendation lists may be an interesting feature, the interface in its current form is too technical and should be made more user friendly to better align with the reduced technical skills of an average user.

3.4 Conclusion

In this chapter we approached the typical human-recommender interaction (HRI) processes of user feedback and recommendation visualization from a user interface perspective. We have discussed state-of-the-art research and elaborated on our own contributions to the domain.

While the 5-star rating system prevailed as the overall *best* user feedback mechanism in multiple independent studies, every system and scenario will still need its own approach. For collecting user feedback, user effort, user benefit and rating accuracy should be taken into account. While the rating system should allow users to specify their preferences on a sufficiently distinct scale, research showed that the associated cognitive effort to engage finer-grained rating scales is significantly higher. Distinct users also showed very different mindsets regarding their willingness to engage with information systems. While some may be willing to invest a lot of effort in providing advanced explicit feedback, others expect recommender systems to simply 'work'. For those users implicit feedback mechanisms (i.e., extracting information from their behavioral patterns) should be considered.

For showing recommendation results, similar conclusions could be made. User interfaces should satisfy the needs of both users who prefer a clean and simple interface and users who seek more advanced information, while at the same time expose all of the recommender system's features (i.e., context options, filters, group recommendation, etc.). Explanations and interactivity turned out to greatly increase user satisfaction (and trust) for a recommender system.

The overall conclusion of this chapter is that there does not seem to be a *silver bullet*. No single rating system is the best for all situations and no single user interface serves for all users. On the contrary, different user interface aspects may be appropriate for distinct situations and users, leading towards an increased need for personalization within the HRI processes. Above all a user interface, should look good, feel natural and optimally support the very important interaction paradigm between users and recommender systems.

Chapter 4

High-Performance Recommending

4.1 Introduction

When recommenders were first introduced, they were often applied to very small datasets (e.g., the MusicFX system [90] with 25 users and 91 items) but since then datasets have massively expanded in size and nowadays a system needs to be able to process datasets like the *Movie-Lens 10M* dataset (10M ratings, 10K items and 72K users), the *Netflix* dataset (100M ratings, 17K items and 480K users) or the *Yahoo! Music* dataset[1] (300M ratings, 600K items and 1M users) and they need to do it fast because users expect responsiveness and real-time behavior.

Since sequential computers (or uniprocessors) are reaching the limits of maximum clock frequency, parallelism is often considered the solution to cope with increasing dataset sizes and limited time constraints [91]. In this chapter we consider the recommendation process from a *high-performance viewpoint* i.e., we study how typical recommendation algorithms and approaches may benefit from distributed deployment on parallel hardware.

All experiments described here (and in most other chapters as well) were run on the Ghent University high-performance computing (HPC) cluster that is freely available for university researchers. The infrastructure offers multiple clusters, each hardware-tailored for specific use cases e.g., focus on multi-core processors, high-speed networking capabilities,

[1]`http://kddcup.yahoo.com/datasets.php`

large-scale RAM memory nodes, etc. Fig. 4.1 shows the conceptual layout of such a cluster. Multiple computing nodes, each containing multiple processing cores and local storage capacity, are interconnected with an Infiniband high-speed network interface. Every node also has access to shared storage in a RAID5 configuration. For more details on the Ghent University HPC infrastructure we refer to its online documentation[2].

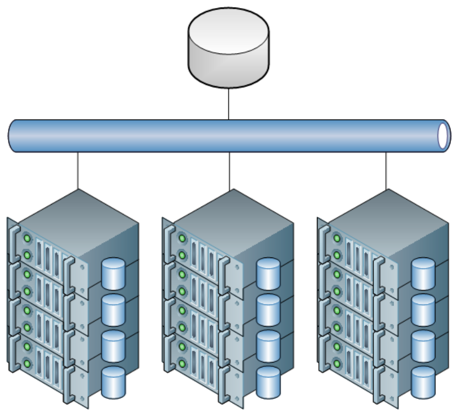


Figure 4.1: The conceptual layout of the high-performance computing infrastructure at our disposal. Every computing node disposes of a local hard disk and has access to a shared storage device in the network.

In the next sections we focus on the intersection of recommender systems and distributed computing. We first discuss relevant related work and then detail our own work focusing on file-based versus in-memory approaches and caching.

Research Questions

- How can we recommend at large scale?
- How to efficiently distribute and parallelize the recommendation calculation process?
- How can we speed up the recommendation calculation process?

[2]`https://www.ugent.be/hpc/en/infrastructure`

4.2 Related Work

4.2.1 Scalability

Related work that deals with the problem of scalable recommender systems, usually focuses on *internal scalability* of algorithms. Algorithms are tuned to focus on only the most relevant data and try to speed up the overall process by taking computational shortcuts (e.g., [92–94]). Especially for neighborhood-based methods a lot of scalability improvements exist. One of the most obvious is restricting the size of the neighborhood [44, 95]. By processing only k (instead of *all*) neighbors, the recommendation process can finish faster. The value of k can however greatly influence the accuracy of the recommendations: k set too high may bring additional noise, while k set too low may reduce recommendation quality [40]. Herlocker et al. [96] suggest that for most real-world situations k set to something between 20 and 50 seems reasonable for the MovieLens dataset.

In this chapter however, we mostly focus on *external scalability*, which does not change any internal logic of the algorithms, and thus does not affect the recommendation accuracy. External scalability considers parallelism and extra hardware as the solution to our big data problem. Instead of changing the recommendation algorithm, the calculation of the algorithm is merely restructured and distributed over multiple computing instances. The final results calculated on a single node will be identical to the results calculated in parallel on a multi-node computing architecture. In general two types of parallelism can be defined: *data parallelism* and *functional parallelism*. In the case of data parallelism, multiple processors work on different parts of the data. Usually the same code is executed on all processors and therefore this scheme is sometimes referred to as SPMD (Single Program Multiple Data) [97]. Functional parallelism involves splitting computations into subtasks that can then be executed in parallel by different processors. In this case, the processors run different code on different data which is called MPMD (Multiple Program Multiple Data) [97].

4.2.2 Distributed Recommender Systems

In [98] and [99], a distributed collaborative filtering (DCF) algorithm called DCFLA is introduced. The scalability of the system is guaranteed by distributing a user-profile management scheme using distributed hash table-based routing algorithms. The authors compared the performance of 4 CF approaches and showed how the scalability of their DCFLA approach surpasses that of the traditional CF algorithm. Performance results in terms of speedup or efficiency values however remained undiscussed.

When recommender systems are deployed in a distributed environment, research and industry often turn to MapReduce as underlying paradigm [100]. The MapReduce approach requires an algorithm to reformulate its logic in essentially two functions: *Map()* and *Reduce()*. An underlying framework (i.e., such as Hadoop) then takes care of the distribution of work across multiple computing nodes. Although work distribution is effortless, reformulating a given algorithm in the MapReduce mindset can be a tedious task and typically involves chaining multiple MapReduce phases together. Since every single phase starts and ends with data access to and from disk, chained phases introduce disk access overhead which limits overall efficiency. Additional overhead is introduced by the setup of the distributed file system e.g. the Hadoop Distributed File System (HDFS) [101], where MapReduce programs depend upon. A well-known library that implements many scalable algorithms including some recommendation algorithms on the Hadoop framework is Mahout[3].

In [102], a user-based collaborative filtering (UBCF) algorithm is implemented on Hadoop. The authors illustrated the complexity of applying the MapReduce model to UBCF and show how it can be done by splitting up the logic in 3 phases. The results showed a linearly increasing speedup, for hardware configurations up to 8 computing nodes. Schelter et al. [103] developed a MapReduce algorithm for the pairwise item comparison and top-N (i.e., recommend the best N items) recommendation problem. For the R2 - Yahoo! Music dataset they achieved a speedup value of about 4 using 20 computing machines (i.e., parallel efficiency of 20%). Jiang et al. [104] implemented an item-based collaborative filtering recommendation algorithm on the Hadoop platform. They chained 4 MapReduce phases and their parallel efficiency was about 90% using 8 computing nodes (using the MovieLens 10M dataset).

[3]`http://mahout.apache.org`

4.3 A File-Based Approach

Recommender systems working with millions of users and items, will not be able to simply read all data into RAM memory and start calculating. Efficient data storage will be required to provide fast data access with minimal delay. Classical relational database management systems (RDBMSes) are often put to the task [105, 106] although they may not always be the best option. The required data throughput needed by recommendation algorithms is very high. Massive amounts of small intermediate values (e.g., ratings, similarity values, etc.) must be stored and retrieved during execution time. If for every required value, a data connection with the database needs to be set up and closed down, the accumulated resulting data delays would make out most of the total execution time altogether. More optimized approaches may be to fetch large chunks of data at once to minimize database interaction. The most optimal being probably to prefetch as much data as can possibly fit in RAM memory. This leads to an interesting idea. If it is so important to keep interaction with the database low, then why not try to leave out the database entirely? It would free developers of the sometimes cumbersome tasks of designing efficient database structures, creating indexes and maintaining the database management software.

Many alternatives to the classical database, often referred to as NoSQL systems, have been developed and some have found their way into the recommender systems domain [107]. We want however an alternative data approach that easily maps on our HPC infrastructure (or any interconnected network of computing nodes with both local and shared storage capacity).

We believe scalability and the concept of keeping the data close to the work are the main goals to optimizing data storage for recommender systems. File systems like HDFS look promising but they often require a complete reorganization of the recommendation algorithm. Furthermore, we did not want to be bound by what technologies the infrastructure supports and so we looked into the most obvious storage approach of all: file-based data. The most straightforward way to store data on a system is by means of files. We found the file-based approach satisfying our needs for easy file system migrations, scalability and performance by carefully structuring input and output files as described in the following sections.

4.3.1 The Recommendation Workflow

In contrast to much literature about recommender systems we do not focus on the internals of the recommendation algorithm. Instead we focus on how we can provide the algorithm with the required data and structure files such that read (and write) delays are minimized, which increases overall scalability. To show our approach works with any type of recommendation algorithm we employ a hybrid recommendation algorithm integrating content-based (CB) and collaborative filtering (CF) much like described by Cornelis et al. [108]. Since this hybrid algorithm internally mixes CB and CF, the processing of the required input for both types of recommendation algorithms can be demonstrated. As stated, we will however make an abstraction of the algorithm itself and provide no further in-depth information.

We abstract the recommendation process to a three-phase workflow that requires the inputs and outputs as shown in Fig. 4.2. The process starts off with the availability of item metadata and consumptions of users. User consumptions can be anything from explicit feedback by means of star-ratings to implicit feedback like behavioral log files.

In a first phase, item similarity is calculated. The item similarity can rely on item metadata, user consumptions or both. The resulting item similarities then serve as the input of the user similarity calculation together with the user consumptions. In the final phase all three user consumptions, item similarities and user similarities are used to generate the recommendations. For a non-hybrid recommendation algorithm this three-phase workflow can be reduced to two phases by leaving out the first (if CF) or second phase (if CB) according to the input requirements for the specific algorithm.

Next, we show how the input and output of each phase can be structured to allow a file-based and scalable data handling approach.

4.3.1.1 Phase 1: Item Similarity

For many years item similarity has been a hot topic in the information retrieval domain. The problem to be solved is how similar two given items in a dataset are. There are numerous ways to go about this, the most popular ones being Cosine Similarity and Pearson Correlation [109] but many more have been investigated. Again we will make an abstraction of the problem, and assume we have an algorithm that given two items i_1 and i_2, calculates their similarity (i_1, i_2). We focus on the problem of

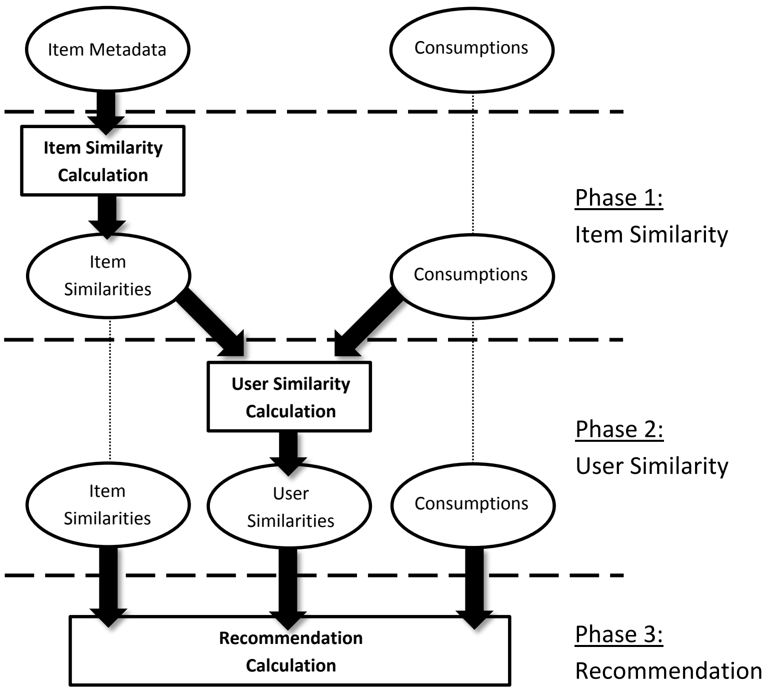


Figure 4.2: The abstracted workflow of the recommendation process focused on the ins and outs of every phase.

matching every item in the dataset and providing the necessary input to the algorithm.

Table 4.1 shows the item similarities for 5 items $i_1, i_2, ..., i_5$. Every item must be compared with every other item, but item similarity is symmetric so only the half size triangular matrix needs to be calculated. Since an item is equal to itself, the total number of comparisons that need to be done will be $\frac{n_i(n_i-1)}{2}$ with n_i the number of items.

	i_1	i_2	i_3	i_4	i_5
i_1	x	$0_?$	$1_?$	$2_?$	$3_?$
i_2	x	x	$4_?$	$5_?$	$6_?$
i_3	x	x	x	$7_?$	$8_?$
i_4	x	x	x	x	$9_?$
i_5	x	x	x	x	x

Table 4.1: The item similarities for 5 items.

A non-scalable data approach would be to read all the metadata from one big file into memory and start comparing items. More scalable is to divide the similarity comparisons amongst the available computing nodes (and their parallel processing cores). To do this, we project the calculation jobs in a one-dimensional space.

$0_?$	$1_?$	$2_?$	$3_?$	$4_?$	$5_?$	$6_?$	$7_?$	$8_?$	$9_?$

Parallelizing the item similarity calculation is now a matter of splitting the one-dimensional job list in m equally large chunks (with m the number of nodes available). For every node, jobs can be split up even further between the available cores per node. Below an example of the parallelizing of item similarity with 5 items over 5 nodes with each 2 cores.

$0_?$	$1_?$	$2_?$	$3_?$	$4_?$	$5_?$	$6_?$	$7_?$	$8_?$	$9_?$

Splitting up the calculations this way, turns the problem of item similarity into an embarrassingly parallel problem with very few dependencies between the jobs. The only thing that the jobs share is the input data. Job $0_?$ will need the metadata of items i_1 and i_2, job $1_?$ from items i_1 and i_3 and so on. If sufficient RAM is available in a node, all the metadata can be loaded. Otherwise a slicing of the metadata will be required to make sure every node is capable of loading its data.

The (file-based) output of the item similarity phase should be carefully structured so that easy and efficient accessibility is possible in the next phase. Since the output growth of item similarity is quadratic $O(n^2)$ in nature, disk usage will rapidly increase. For e.g., the similarity of 50,000 items over a billion comparisons must be calculated and stored. Two extreme approaches would be dumping all calculated similarities in one big file versus writing to a new file for every item, none of which are scalable. It is clear that a meet-in-the-middle approach will have to be devised.

We use the concept of *file buckets* to balance the similarities output. We define a file bucket as a container of individual files. A file bucket itself is in fact again a file. Instead of creating an individual similarity file for every item, we spread out the similarities over the number of file buckets available. So a file bucket contains the similarities of 1 or more items depending on the number of file buckets used.

To decide which file bucket a similarity e.g. (i_1, i_2) should be written to, we assign an internal numerical id to every available item in the system, ids ranging from 1 to n_i (the total number of items). Item similarities are then spread out over the file buckets using a modulo function. Table 4.2 shows how the similarities of our earlier example would be divided amongst 3 file buckets (ranging from 0 to 2).

#	*item simil* (x, y)	file bucket
$0_?$	$(i_{\underline{1}}, i_2)$	1 ($\underline{1}$ *mod* 3)
$1_?$	(i_1, i_3)	1
$2_?$	(i_1, i_4)	1
$3_?$	(i_1, i_5)	1
$4_?$	$(i_{\underline{2}}, i_3)$	2 ($\underline{2}$ *mod* 3)
$5_?$	(i_2, i_4)	2
$6_?$	(i_2, i_5)	2
$7_?$	$(i_{\underline{3}}, i_4)$	0 ($\underline{3}$ *mod* 3)
...	...	...

Table 4.2: The dividing of the outputs in file buckets.

We found that it was more efficient for the processing in the following phases to actually write every similarity to the file buckets instead of just the half triangle matrix. So for every calculation two values are written e.g. for $0_?$ we write both (i_1, i_2) to bucket 1 and (i_2, i_1) to bucket 2. Note that similarities will be evenly spread out over the available buckets but similarities of the same item e.g. (i_1, i_2), (i_1, i_3), (i_1, i_4) and (i_1, i_5) will always be in the same bucket. This allows for efficient loading of the similarities of an item (i.e., only one file bucket needs to be read), which will be required in the next phase.

Writing with a lot of different computing nodes (and cores) to the same file (bucket) can slow down the file system substantially. Therefore every core in every node should write to its own dedicated file buckets on its local disc. For every node the file buckets can then be merged locally first for the cores in that node, next over all nodes. Finally, the merged file buckets from all nodes and cores can then be stored on the shared storage. Fig. 4.3 visualizes this output process for 2 nodes with each 3 cores mapped on the HPC infrastructure.

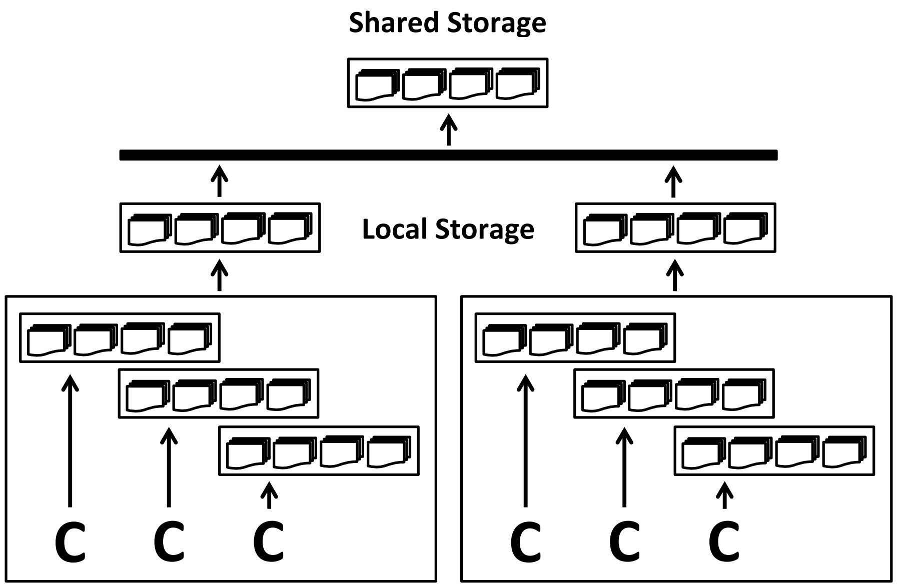


Figure 4.3: The merging file buckets strategy for two nodes with 3 cores. Every core writes to dedicated file buckets for optimal file system write efficiency. The file buckets are then merged per node and finally over all the nodes to the shared storage.

4.3.1.2 Phase 2: User Similarity

User similarity defines some degree of matching between two users. The term similarity here is somewhat misleading as we are interpreting the similarity of u_1 towards u_2 as the degree to which u_1 may provide interesting items for u_2 (as proposed by [108]). In contrast to the item similarity, this user similarity is not symmetric. The calculation of all the user similarities will therefore result in $n_u(n_u - 1)$ comparisons with n_u the number of users. Table 4.3 shows the user similarities for $n_u = 4$.

	u_1	u_2	u_3	u_4
u_1	x	$0_?$	$1_?$	$2_?$
u_2	$3_?$	x	$4_?$	$5_?$
u_3	$6_?$	$7_?$	x	$8_?$
u_4	$9_?$	$10_?$	$11_?$	x

Table 4.3: The user similarities for 4 users.

In our abstract recommendation workflow, the calculation of the user similarity (u_1, u_2) requires as input both the consumptions (i.e., ratings) of user u_1 and user u_2, and the item similarities of the items rated by u_1. If u_1 has rated for example two items i_x and i_y then the item similarities of i_x and i_y with every other item in the system must be loaded.

To load all the item similarities of a given item, we must simply load the corresponding file bucket determined by the internal id of the item and the modulo function. Because of this file buckets structure there is no need to load all the item similarities except for in the worst case scenario where the needed items are spread out over all file buckets. This can however be easily prevented by configuring an appropriate number of file buckets.

The loading of the consumptions should not pose any problems since even for big datasets like the MovieLens dataset with 10M ratings, the consumption file is only 262MB and can easily be loaded in memory.

For optimization reasons, we parallelize the user similarity calculation tasks on a different granular level than we did for the item similarity. The calculation of the user similarities (u_1, u_j) with $j = 2, 3, 4$ requires the item similarities of the items rated by u_1. It therefore makes sense to load these item similarities once and then process (u_1, u_2), (u_1, u_3) and (u_1, u_4). In the situation of 4 nodes with each 3 cores available, we would divide the calculation tasks as illustrated in Table 4.4.

$node_1$	$0_?$	$1_?$	$2_?$
$node_2$	$3_?$	$4_?$	$5_?$
$node_3$	$6_?$	$7_?$	$8_?$
$node_4$	$9_?$	$10_?$	$11_?$

Table 4.4: The division of the calculation task of the similarity values of 4 users over 4 computing nodes with each 3 cores available.

So all the similarity calculation tasks of the same user are handled by the same node; within the node the tasks can further be delegated towards the available cores.

The same output strategy as we proposed for the item similarities (Fig. 4.3) can be applied here. For efficiency reasons every core must again write to dedicated file buckets to be merged first locally on the node and then globally to the shared storage.

4.3.1.3 Phase 3: Recommendations

A recommendation is a match between an item and a user. To calculate the complete set of recommendations, such a matching between every item and user must be made as shown in Table 4.5.

	u_1	u_2	u_3	u_4
i_1	.	.	.	.
i_2	.	.	.	.
i_3	.	.	.	.
i_4	.	.	.	.
i_5	.	.	.	.

Table 4.5: The recommendations for 5 items and 4 users.

Our recommendation algorithm requires for the recommendation of an item i to a user u both the user similarities of u and the item similarities of i. This maps directly onto the file buckets structure generated by the item and user similarity phases.

To match every user with every item, every generated file bucket in the item similarity phase must be matched with every file bucket from the user similarity phase (Fig. 4.4). This approach again allows easy scaling, since the couples of item and user file buckets can be divided amongst the available nodes (and cores) in the infrastructure.

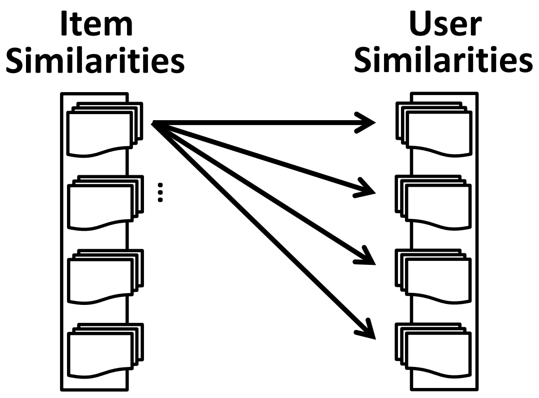


Figure 4.4: In the recommendation phase all the file buckets of the item and user similarities must be matched with each other.

A node needs only to load the item similarity file bucket and the user similarity file bucket as input for the recommendations of the items and users contained in the buckets. By increasing the number of file buckets, their file size can be reduced which may allow a node to fully load the required data into RAM memory. A trade-off between the number of jobs (couples of buckets) and the size of the job (size of the buckets) will have to be made.

4.3.2 Experimental Results

To validate our file-based recommendation approach we used our cultural events dataset collected from the *UITinVlaanderen* website. This events dataset was particularly interesting to test our recommendation algorithm because events are *one-and-only* items [110] and difficult to recommend with a non-hybrid recommender.

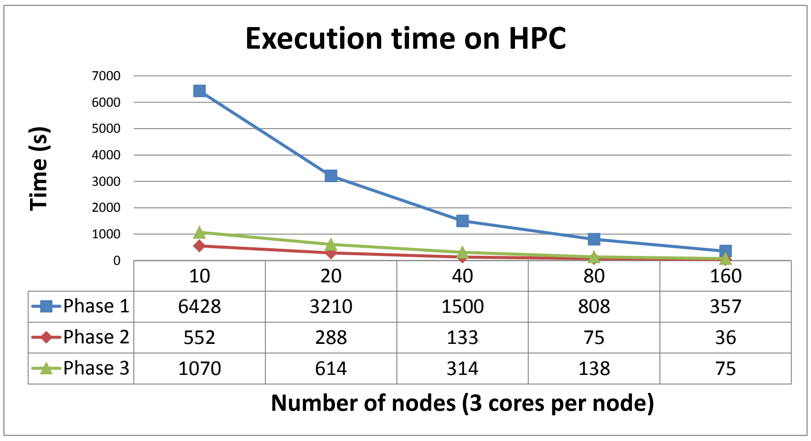
The dataset contained the metadata of 53,000 items (i.e., events) and consumptions of 1,700 users. We aggregated the 14,000 implicit and explicit user consumptions (i.e., user feedback), into 6800 consumptions by using a simple weighing scheme. In total 4700 unique events of the dataset were eventually consumed at least once.

Since the focus here is on the applicability of our file-based approach, we do not present any quality metrics about the recommendations themselves. The recommendation quality will not be influenced by our distributed calculation approach and therefore depends solely on the involved recommendation algorithm. Instead we plot the execution time of the three introduced phases of our recommendation workflow for changing hardware configurations (Fig. 4.5).

By doubling the number of used computing nodes on the HPC infrastructure we can see that the execution time of each of the phases decreases to halve the time. We repeated the experiment with a fixed amount of nodes while varying the number of cores in each node. Similar results were obtained.

The execution time of the first phase is significantly higher than that of the second and third phases. This is a result of the number of items versus the number of users available in the dataset (53,000 versus 1,700) and therefore not a consequence of the employed algorithm. Note that in the third phase the recommendation value for every item for every user in the system is calculated.

As the experimental results illustrate, using a simple file-based approach, we managed to deploy a complex hybrid recommendation algo-



	10	20	40	80	160
■ Phase 1	6428	3210	1500	808	357
◆ Phase 2	552	288	133	75	36
▲ Phase 3	1070	614	314	138	75

Figure 4.5: The execution time of the three phases executed on the HPC infrastructure by 10, 20, 40, 80 and 160 computing nodes. All nodes having three processor cores available. For phase 1, 200 file buckets where used. Only 1 file bucket was used for phase 2 considering the small number of users.

rithm on a distributed computing infrastructure by functionally dividing the work into separate subtasks (i.e., our 3 phases) which could be further decomposed into smaller chunks of work and mapped onto multiple parallel computing nodes.

4.4 In-Memory, Content-Based Recommendation

In the previous section, we focused on *functional parallelism* by dividing the recommendation process into separate phases each of which could be mapped onto multiple worker nodes. While resulting processing times scaled linearly with the applied hardware, efficiency was lost by the overhead introduced by chaining multiple phases together (each of which required reading and writing intermediate data results *from* and *to* permanent storage). In this section, we focus on *data parallelism* in order to simplify the parallel recommendation process, reduce the overhead of separate phases, and avoid load imbalance issues. We show how a content-based recommendation algorithm can be parallelized and distributed over multiple computing machines without underlying MapReduce operations or file system restrictions. While MapReduce has certainly proven its use for recommender systems [111, 112] in general, we believe the paradigm may sometimes lack the flexibility to take specific algorithm properties into account to reach high parallel efficiency values.

We show that specifically for the content-based recommendation algorithm we can divide 'work' evenly and independently such that by tweaking certain parameters of the distribution process, a given hardware configuration is optimally engaged without inter-node communication and mid-computation disk access. Because our approach executes completely in-memory (i.e., only RAM is used to store mid-computation values), disk access is reduced to a minimum and high efficiency values can be obtained.

4.4.1 Dataset Specification

We will validate our in-memory approach on the MovieLens 10M dataset. This dataset contains 10 million ratings and 95,580 tags applied to 10,681 movies by 71,567 users of the online movie recommender service MovieLens. Data is provided as three files: *ratings.dat* (252MB), *tags.dat* (3MB), and *movies.dat* (500KB). Interestingly, while every user in the MovieLens dataset has rated at least 20 movies, some users have rated considerably more. As shown in Fig. 4.6, 50% of all ratings originate from only 15% of all users. This unequal distribution of ratings can easily introduce load imbalance issues when carelessly distributing calculations across worker nodes (a challenge which we will face). For more detailed information about the dataset we refer to recommender systems literature (such as [45, 46]).

4.4.2 Parallel CB Recommender

Here, we define the recommendation algorithm that will be used to illustrate our in-memory approach. Since we focus mainly on optimally distributing and running the algorithm in parallel on a HPC infrastructure, a default out-of-the-box content-based (CB) recommendation algorithm (as described in [40]) was used as starting point. The CB algorithm calculates the (Jaccard) similarity between items based on the item metadata (i.e., MovieLens files: *movies.dat* and *tags.dat*) and recommends new items to users based on these similarities. We use the MovieLens (10M) dataset as input and deploy the algorithm to predict the user ratings for unknown (*user*, *item*) pairs. Our Python implementation can be found

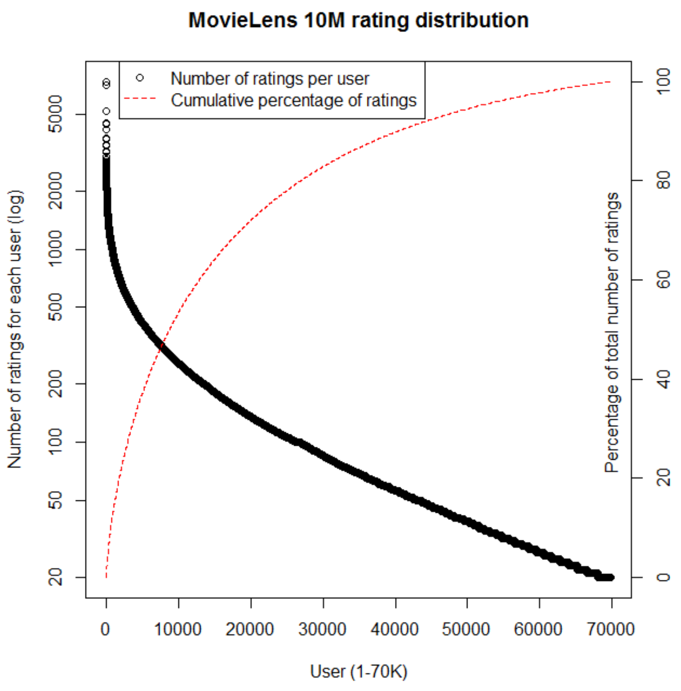


Figure 4.6: The number of ratings per user for the MovieLens 10M dataset.

in the Github platform[4].

Calculating the recommendation value for a certain user and item requires the comparison of similar items previously rated by the user. Although the number of such processed similar items is usually limited in size to reduce the computational burden and minimize possible noise [40], we do not restrict the neighborhood size to illustrate the true scalability of our method. In fact, the only optimization that is incorporated in the algorithm is the temporary storage (caching) of calculated similarity values to prevent unnecessary recalculations. However, since these values can easily become too abundant to store (with limited RAM), they are cleared with every item iteration. The pseudocode of the CB recommendation algorithm can be found in the appendix (Algorithm 2).

The calculation of $Rec(user,\ item)$ requires the rating data of the user together with the item data of the item and any other item rated by the user. We note that although the algorithm used in this work is content-

[4]`https://github.com/sidooms/DistributedCB`

based, user ratings are still important as they are used in the weighted average formula of the target user in the recommendation calculation procedure. Because we want to distribute the calculation work over multiple computing nodes, both user data (i.e., ratings) and item data will have to be considered in the distribution process. Moreover, because both data types are taken into account, our distribution paradigm can be extended to also fit collaborative filtering algorithms that focus on rating data only.

4.4.3 Parallel Strategies

We want to split the work of a complete recommendation calculation (i.e., calculating the recommendations values for all user and item pairs) into smaller pieces of work that require less input data and can be distributed over available worker nodes. In the context of a recommender system, input data consists of user data (usually ratings) and item data. The actual work consists of the calculation of the recommendation value (i.e., numeric value indicating the interest of the user) for every (*user*, *item*) pair in the system. This is usually visualized as a user-item matrix as depicted in Fig. 4.7. Every dot in the matrix represents a recommendation value to be calculated. The value of some dots may already be known, as users may have rated some items (indicated by multiple **R**s). These ratings are considered the perfect prediction of the interest of the user for that item.

The way in which the work (i.e., dots) is divided over available worker nodes will have an impact on the amount of input data needed for that node. We present three parallel strategies for dividing the work of a recommender system and their data related consequences.

4.4.3.1 Splitting in Userjobs

We could partition the user-item matrix in horizontal subsets to distribute the users that must be processed across the available worker nodes (Fig. 4.8). When these subsets are mapped onto worker nodes, every node must then calculate the recommendation value for each of these users in the subset and every item in the system.

Because the users are divided over different jobs (we refer to these as ‘userjobs’), the user input data can be split accordingly into smaller subsets. Consequently, worker nodes will be able to work with smaller datasets which can help to reduce RAM requirements. A rather technical

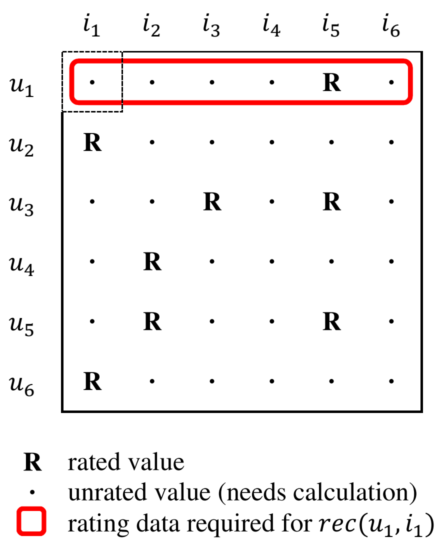

i_1 i_2 i_3 i_4 i_5 i_6
u_1 u_2 u_3 u_4 u_5 u_6
R rated value
· unrated value (needs calculation)
rating data required for $rec(u_1, i_1)$

Figure 4.7: The user-item matrix indicating the work related to a complete recommendation calculation of every (*user*, *item*) pair.

downside of this division scheme is that with a bigger number of userjobs, fewer computed intermediate item similarity values can be re-used. For a (*user*, *item*) pair, an item similarity value will be calculated between the item and all the items rated by the user. The more users are handled by a single worker node, the more of these intermediate values can be re-used. A large number of userjobs indicates a small number of users per job and so fewer recycling of intermediate similarity values.

4.4.3.2 Splitting in Itemjobs

Alternatively, we could partition the user-item matrix into vertical subsets and distribute the items across the available nodes (Fig. 4.9). Every job (i.e., 'itemjob') must now process the recommendation value for each item of the subset and every user in the system.

The obvious advantage of this method is that since every user in the system is now matched with a subset of items by every worker node, the amount of redundant item similarity computations is reduced to zero. On the other hand, because all users are processed, all user input data (i.e., rating data) needs to be loaded by every itemjob which may be too much for the RAM of a single worker node.

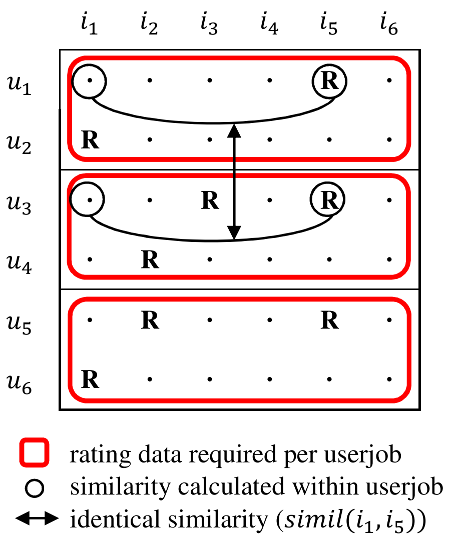

i_1 i_2 i_3 i_4 i_5 i_6
u_1 u_2 u_3 u_4 u_5 u_6
R
rating data required per userjob
similarity calculated within userjob
identical similarity ($simil(i_1, i_5)$)

Figure 4.8: The user-item matrix split into a number of userjobs (every userjob processes all items).

4.4.3.3 Hybrid Userjob, Itemjob Splitting

Since both splitting in userjobs and itemjobs have benefits and downsides, a meet-in-the-middle approach seems a good option. In this case, we split the grid of user and item pairs into disjoint subsets of users and items to be distributed across the available worker nodes. This approach allows partial recycling of similarity values while reducing the required user input data. The number of userjobs and itemjobs can be freely chosen and specifically tailored towards the available computing hardware.

The hybrid userjob, itemjob approach introduces the highest flexibility and can even be turned into one of the previous approaches by setting the value of the number of userjobs (or itemjobs) to ‘1’. Therefore, we will adopt the hybrid data parallelism strategy.

4.4.4 Load Balancing

If work is not evenly distributed among available workers then load imbalance issues may arise. These occur when synchronization points are reached by some workers earlier than others [97] and so workers display

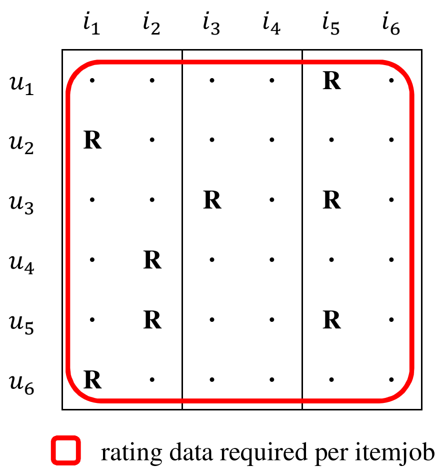

i_1 i_2 i_3 i_4 i_5 i_6
u_1 u_2 u_3 u_4 u_5 u_6
R
rating data required per itemjob

Figure 4.9: The user-item matrix split into a number of itemjobs (every itemjob processes all users).

significantly divergent walltimes (i.e., time to solution). Load imbalance greatly impacts the efficiency of algorithms because resources are underutilized while fast workers wait for slow workers to finish. If work is load balanced, it can be evenly divided over computing nodes without the need for extra inter-node communication (as would be the case in a master-slave scenario). This relaxes network constraints and allows for active engagement of all computing nodes.

For a system to be load balanced, work must be evenly distributed among its workers. Specifically in the context of recommender systems, this introduces two subproblems: How do we define work, and how can it be evenly distributed?

4.4.4.1 The Definition of Work

A straightforward way of expressing work is by computation time. Longer computation times indicate more work has been done. However, before the recommendation calculations are performed, the actual computation time is unknown. What is known, are the total number of users, items, available ratings, etc. If one of these metrics shows a positive correlation with the calculation time, it can be used as a shorthand definition of work.

We have previously introduced the notion of userjobs and itemjobs.

Because of this distinction, we want to be able to define work in terms of both user-related metrics and item-related metrics. We devised two experiments focusing on these subsets of metrics.

4.4.4.2 The Definition of Work: in Terms of Users

Intuitively, more processed users will result in longer computation times. The number of ratings that are provided by these users however may also be important. To avoid confusion, we note that when we refer to *ratings* we are referring to the user-provided ratings that are already available in the dataset (**R**s in Fig. 4.7).

The CB algorithm as discussed in Section 4.4.2 requires that to calculate *Rec(user, item)* all the ratings of that user will be taken into account. So a job processing users with a low number of ratings may finish faster than a job with many ratings per user. We define two recommendation metrics as possible candidates for a user-related work definition: *the number of users* and *the total number of ratings* provided by these users.

We measured the correlation and performed a simple regression analysis of these metrics with the actual resulting computation time for a configuration of 40 worker nodes (1 core per node). Each user was randomly assigned to one of these nodes and each node processed all of the items available in the MovieLens dataset (i.e., 40 userjobs, 1 itemjob). Since the set of items processed by each worker was the same, the influence of processed users (and therefore also ratings) on the computation time could be isolated. Fig. 4.10 shows two scatterplots indicating the computation time (of the 40 userjobs) in function of the number of users per job (left) and number of ratings (given by these users) per job (right).

While the number of users is not entirely bad at predicting the execution time ($R^2 = 0.286$), the number of ratings is an almost perfect predictor ($R^2 = 0.9251$). We learn from this that in order to achieve a load balanced system we should take the number of available ratings processed by each worker node into account rather than the number of users.

To further illustrate this concept, we compared the calculation times when evenly distributing the number of users versus distributing the users in such a way that the total amount of ratings given by these users is (as good as) equal for each node. Fig 4.11 shows the results, and as expected, the distribution of users shows a much more irregular surface pattern while the distribution of ratings results in a load balanced

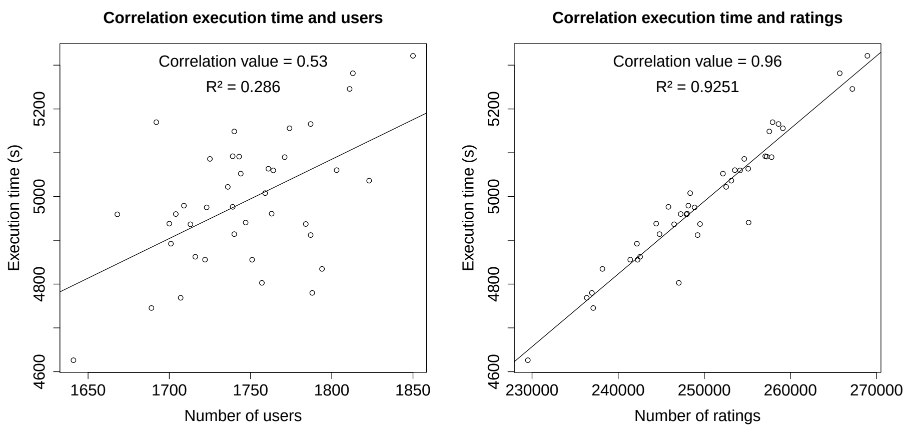


Figure 4.10: Scatterplots indicating for a configuration of 40 worker nodes (i.e., 40 userjobs, 1 itemjob) the correlation of the number of users (left) and the number of given ratings (right) with the calculation time of each job.

system.

Because work (or calculation time) is largely connected with the number of available ratings, we should strive towards an equal distribution of the number of ratings among worker nodes. Equally distributing the users is not enough because of the large divergence in number of ratings per user for our dataset.

4.4.4.3 The Definition of Work: in Terms of Items

To define work in terms of items the obviously available metric is *the number of items* that are processed by a worker node. However, with the algorithm presented in Section 4.4.2, the workload associated with an item may vary. To calculate the recommendation value $Rec(u,\ i)$ all the ratings of user u must be taken into account (**for** loop in *Rec(user, item)* procedure). We can express the true workload of item i by analyzing the number of times this **for** loop will be iterated on. This amount of iterations depends on the number of ratings of u and on the fact that u may or may not have already rated i. If u has rated i, $Rec(u,\ i)$ will not be calculated and the amount of iterations will be zero. Therefore, for a given set of users, the number of iterations for an item will be equal to the sum of the ratings of the users minus the sum of the ratings of the users that have rated i (because they will be skipped). We define a metric *item iterations* or $iter(i)$, as the total number of iterations that must be

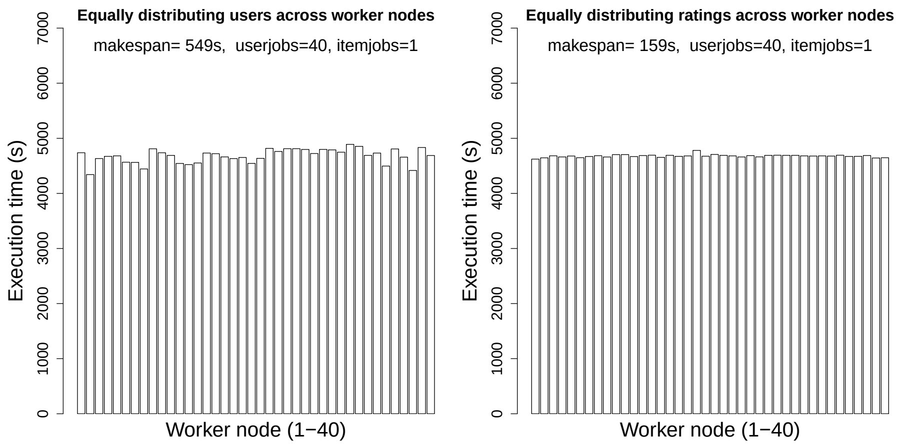


Figure 4.11: The resulting calculation times when equally distributing the number of users across worker nodes (left) versus distributing the number of users such that their total number of ratings are equal across worker nodes (right).

run for an item i to calculate the recommendation values $Rec(u,\ i)$ for all users u.

$$\begin{aligned} iter(i) &:= |number\ of\ iterations\ for\ item\ i| \\ &:= |all\ ratings| - |skipped\ ratings| \\ &:= |all\ ratings| - \sum_{(\forall u | u\ rated\ i)} |ratings\ of\ u| \end{aligned}$$

Again we set up a simple regression experiment for the two item-related metrics (*number of items* versus *number of item iterations*) with the actual computation time. The items were randomly distributed over 40 worker nodes (1 core per node) and every node processed every user (i.e., 1 userjob, 40 itemjobs). When inspecting the predictive capabilities of our metrics, both *number of items* ($R^2 = 0.9342$) and *the number of item iterations* ($R^2 = 0.943$) were found to be very good predictors for the calculation time.

When work was actually divided equally according to these two metrics and results were compared, *the number of item iterations* proved to be slightly better in terms of load balance (Fig. 4.12).

To conclude, we note that to obtain a load balanced system we should

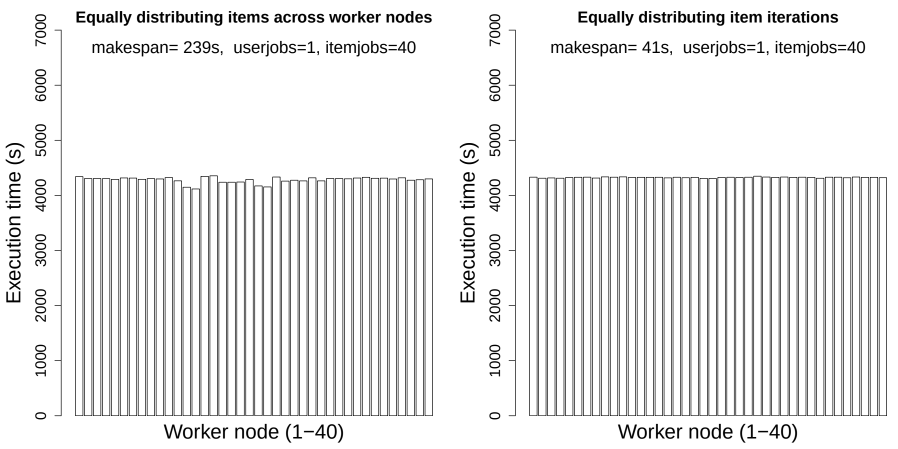


Figure 4.12: Calculation times for equally distributing the number of items (left) versus equally distributing the number of item iterations (right).

strive to an equal distribution of both *number of ratings* and *number of item iterations* over the available worker nodes.

4.4.5 Work Distribution

We now detail how we distribute work equally among worker nodes. Only if work is equally distributed, a load balanced situation can be achieved and therefore a higher parallel efficiency.

4.4.5.1 The Partition Problem

The problem of equally distributing work across homogeneous workers is not new, it is in fact a very well-studied problem in the theory of approximation algorithms [113], often referred to as 'makespan minimization'. In this problem, a number of jobs (with different estimated processing times) need to be scheduled across a number of (identical) worker nodes, such that the maximum time for any node to finish its work (i.e., the makespan) is minimized. In our context, we need to partition users and items in subsets to be processed by worker nodes such that the resulting subsets show an equal amount of ratings and item iterations. To do this, we need to solve the makespan minimization problem, but since it is considered to be *NP-complete* [113], a fully polynomial algorithmic approach might not exist. However, countless approximation schemes have been proposed to tackle this problem (e.g., [114–117]).

The problem with optimization schemes such as Monte Carlo, genetic algorithms, etc. is that they often require a large number of iterations to converge to an acceptable solution. The runtime of the partitioning of work among worker nodes will however be of crucial importance to the final performance of the system. This partitioning will have to be executed sequentially (i.e., can not be parallelized) and thus strongly limits the parallel efficiency. Moreover, we must make sure not to put more effort (i.e., time) into optimizing the partitioning than would be won by the improved load balancing. Since speed is so important, instead of applying more advanced optimization solutions we employ a simple $O(n \log n)$, greedy partitioning algorithm (Algorithm 1).

Algorithm 1 Work Distribution

```
1: Let $jobs$ be a list of jobs
2: Sort $jobs$ according to estimated workload (high to low)
3: for all $jobs$ do
4:     assign $job$ to worker node with currently lowest workload
```

The accuracy of this solution depends on the specifics of the input data and the number of desired partitions. If all users and items are equal in terms of our definition of work then the algorithm will provide an optimal solution. On the other hand, if they are extremely divergent and lots of partitions are needed, the results (in terms of 'makespan minimization') may be poor.

While we found this solution to be sufficiently accurate (i.e., leading to sufficiently load balanced systems) for most hardware configurations, some situations do require some extra attention. The accuracy of this simple greedy algorithm can be heuristically improved by our proposed *Robin Hood extension.*

4.4.5.2 Robin Hood Extension

To improve the accuracy of the simple greedy partitioning algorithm, we employ a method we refer to as the 'Robin Hood' extension. The idea is to iteratively take work from 'the rich' and give it to 'the poor' in order to balance out the wealth inequality. In this context, we define wealth as the workload of a worker node after initial partitioning. Using the definition of 'work' in Section 4.4.4.1, we can compute this workload of a worker node by summing up either *the number of ratings* (when dividing in userjobs) or *the number of item iterations* (when dividing in itemjobs) that need to be processed by the node. When the workload for every worker node is known, we select the richest (i.e., highest workload) worker node and the poorest (i.e., lowest workload) worker node. We then randomly pick a user (or item) from the rich node and add it to the users (or items) of the poor node. Doing so, we level out the maximum workload difference (i.e., makespan) associated with the worker nodes. This process can be repeated until a desired threshold of minimum makespan has been reached or a maximum number of iterations has been run.

As stated, this extension of the partition algorithm may only be needed in a few cases where the partitioning algorithm performs worse than a certain threshold of inequality. To illustrate the behavior of the Robin Hood extension, we executed it after partitioning the items of the MovieLens dataset over 40 itemjobs (and 1 userjob) with the simple greedy partitioning algorithm. We set the extension to run 500,000 iterations and we plotted the minimum makespan after every iteration (Fig. 4.13). As shown in the figure, the Robin Hood extension allows to rapidly reduce the makespan difference within only a few thousand iterations. While the makespan is reduced, work is more evenly divided over the available worker nodes and so it becomes more difficult to improve for an increasing number of iterations.

4.4.5.3 Dividing in Userjobs and Itemjobs

In the context of recommender systems, data parallelism is a very promising concept because it allows the processing of extremely large datasets on commodity computers. More interestingly, it introduces a degree of flexibility in the sense that data can be partitioned into pieces that maximize the utilization of worker resources (e.g., RAM) and therefore computational efficiency. We divide our data grid (users-items matrix) by first splitting the users into U chunks of users to be processed by an equal amount of userjobs, and then for each userjob splitting the items into another I itemjobs. The final number of jobs will consequently be the product of the number of userjobs and itemjobs.

$$total\ number\ of\ jobs\ = userjobs \times itemjobs$$

As detailed in Section 4.4.3, a worker node calculating recommendation values for (*user*, *item*) pairs must be able to hold all the ratings and item

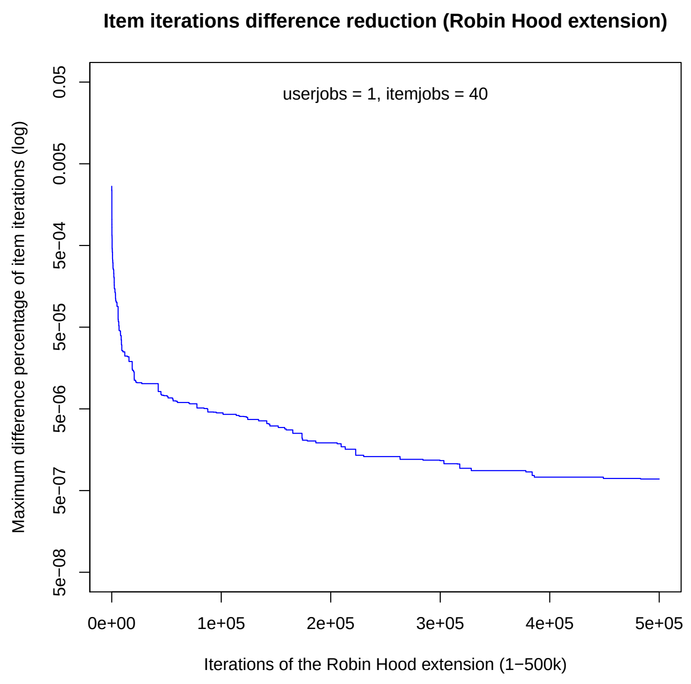


Figure 4.13: The declination of the item iterations difference between worker nodes after each iteration of the Robin Hood extension (up to 500K iterations). Note that the Y-axis is on a logarithmic scale and expressed as a percentage of the total number of item iterations.

data of these users and items in RAM. Therefore the most constraining resource requirement in the system is the amount of RAM in a computing node. Since user feedback data is usually more abundant than item data (for MovieLens 10M the ratio is 70:1) we start by dividing the user data. Data is split into equally-sized but smaller parts (i.e., userjobs). By equally-sized we mean in fact equal in terms of the metrics we defined in Section 4.4.4.1. Since we are dealing with user data, the relevant metric here is *number of ratings*. We use the partition method as described in Section 4.4.5.1 to divide the users in a number of parts such that each part contains an equal number of ratings. The number of userjobs can be freely chosen, but as mentioned, this will drastically impact the worker nodes resource requirements.

We then continue to split the input data in terms of items (i.e., itemjobs) for every userjob. As we defined *number of item iterations*

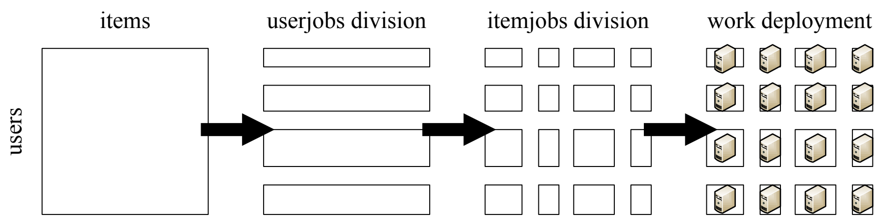


Figure 4.14: Overview of the work distribution from input data in terms of users and items to the final mapping of the data on worker nodes.

as a good predictor for workload in terms of items, we divide the items in a number of parts such that each part contains about an equal number of iterations. Since the number of iterations depends on the number of ratings that must be processed, itemjob division must be carried out after the userjob division.

Since both the userjobs and itemjobs are divided in a way that optimizes load balance, running both processes sequentially will also lead to a load balanced system. A graphical overview of the distribution of work in terms of users and items can be found in Fig. 4.14.

While the number of userjobs and itemjobs can be freely chosen, the choice does also impact the final load (im)balance state of the system. When dividing a set of numbers into x subsets of numbers with an (as much as possible) equal sum, the value of x is of great importance. It is easier to split up the numbers equally over 2 subsets, than over 4 subsets. Since the employed partition algorithm is heuristic in nature, the solution it generates depends on the difficulty of the problem. To investigate the effect of the number of userjobs and itemjobs on the final load imbalance of the system we ran some benchmarks.

We ran a number of job division processes (without Robin Hood extension) with varying userjob and itemjob parameters and compared their resulting maximum item iterations difference (Fig. 4.15). The difference in terms of *number of ratings* was negligible (because there are 10 000 times less ratings than item iterations to divide) and is therefore not displayed.

Fig. 4.15 shows the opposite effects of the number of userjobs and itemjobs on the resulting difference. A larger number of userjobs implies working with smaller subsets of ratings that consequently need to be divided further into itemjobs. Therefore, more userjobs implies a smaller item iterations difference while more itemjobs implies a bigger item iter-

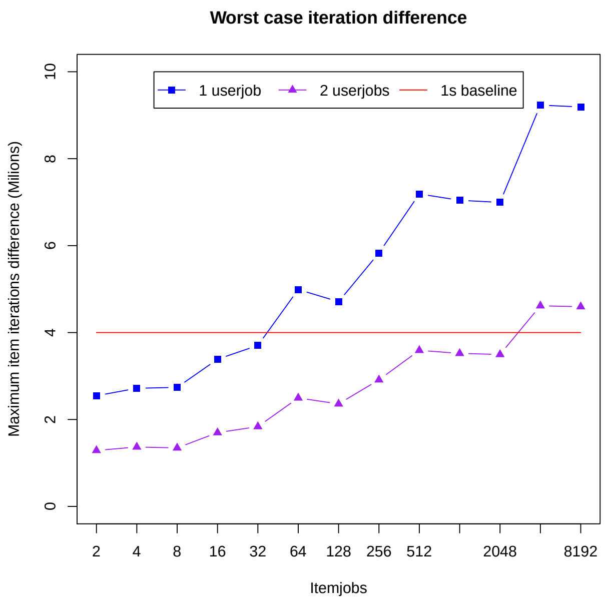


Figure 4.15: The maximum iterations difference for a work division (without Robin Hood extension) with a varying number of userjobs and itemjobs. As a baseline we indicate the number of item iterations that can be processed in 1 second.

ations difference because of the increased complexity of the problem.

A difference of 4 million item iterations (indicated on the figure) in our system corresponded with 1s difference in computation time (which was only 0.002% of the total time). It is clear that for most configurations the item difference will be negligible with respect to the total calculation time. The results and benchmarks presented here, apply specifically to the MovieLens dataset. Other datasets might introduce other ratios of ratings, users and items and so these benchmarks would have to be repeated to determine the accuracy of partitioning. If the load imbalance caused by the partition algorithm turns out too big, the Robin Hood extension can be used.

4.4.6 Performance Model

In this section we evaluate the performance of our proposed content-based in-memory recommendation algorithm. As stated earlier, we do not consider any qualitative aspects of the final recommendation output. The distribution of the calculation task across different machines does not influence in itself the quality of the recommendations (e.g., metrics such as prediction accuracy) and thus a qualitative evaluation is in this context not relevant.

The performance of a parallel algorithm is not easily measured as it can be influenced by many things. As [97] suggests, performance may be limited by load imbalance issues, the amount of serialized parts of the concurrent execution, and communication overhead that may be introduced because of the increased amount of worker nodes. We have partitioned the complete recommendation problem into any predefined number of subtasks with equal complexity. These recommender subtasks can then be mapped onto worker nodes without introducing load imbalance issues. Our algorithm and parallel strategy are devised in such a way that jobs are executed independently and so no communication overhead is introduced by increasing the number of worker nodes. Serial code fragments, on the other hand, could not be avoided. That is, some code can only be executed by one processing node and therefore limits the performance of our system.

In previous sections, we mainly focused on the recommendation calculation part of our algorithm, but for a realistic performance model, we have to consider the complete recommendation process. This process also includes processing input data and running the work division as detailed in Section 4.4.5. To gain insight into the performance of the complete recommendation process we started by measuring the execution time of every part of the process. We define these parts in function of how well they can be parallelized: *serial*, $parallel_U$, $parallel_I$, or $parallel_C$. Parts that are *serial* can not be parallelized and have to be executed sequentially. Parts that are *parallel* on the other hand can be run in parallel on a number of worker nodes. We define three types of *parallel* to make a distinction between parts that are able to run in parallel on a number of userjobs ($parallel_U$), number of itemjobs ($parallel_I$), or on all the jobs at the same time including multiple processing cores per job ($parallel_C$).

An example of what might be considered $parallel_U$ is the processing of user input data. After a certain node has read the input data from disk (which we do not take into account here), these data need to be parsed and stored into a memory data structure. Every worker node needs to do this, but only for the subset of user input data that was divided by the

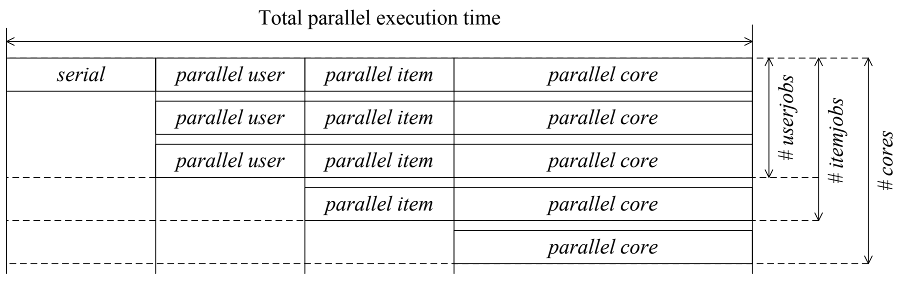


Figure 4.16: A schematic view of the total execution time of the complete recommendation process in terms of how different parts of the algorithm can be parallelized.

userjob work division. Therefore we refer to this work as parallelizable in terms of userjobs.

We ran the complete recommendation process with a single worker node and one active processing core as a performance baseline. For this, we set the number of userjobs and itemjobs to '1' (so all users and items are processed by a single job). Table 4.6 shows the resulting execution times for each part in function of the way they can be parallelized. As expected, the recommendation calculation itself, which can be fully parallelized (P_C), accounts for most of the processing time.

To gain some insights into the parallel performance of this algorithm, we calculate the speedup for a fixed-size problem (Amdahl's Law [118]). In its simplest form, speedup S_p is defined by Formula 4.3. If s is the amount of serial work and p the amount of parallel work, we define

Parallelizability	Time (s)	Time (%)
serial (s)	41	0.02339
$parallel_U$ (P_U)	29	0.01675
$parallel_I$ (P_I)	0.01	0.00001
$parallel_C$ (P_C)	174 936	99.95985
Total:	175 006	100

Table 4.6: The execution times for a complete recommendation calculation process with 1 worker node (1 userjob, 1 itemjob, 1 core).

$T(1)$ as the execution time on 1 worker node and consequently $T(N)$ the execution time on N worker nodes.

$$T(1) = s + p \tag{4.1}$$
$$T(N) = s + \frac{p}{N} \tag{4.2}$$
$$S_p = \frac{T(1)}{T(N)} \tag{4.3}$$

We adapt this formula to introduce our notion of userjobs U, itemjobs I, and processing cores C. The total number of worker nodes in our context will be equal to $U \times I$ and every one of these worker nodes may be equipped with C processing cores. So we redefine the speedup in terms of U, I, and C.

$$S_p = \frac{T(1,1,1)}{T(U,I,C)} \tag{4.4}$$

To calculate the speedup of our system, we must know the baseline time $T(1,1,1)$ (which we measured in Table 4.6) and $T(U,I,C)$, which is the time for the system to complete the calculations with U userjobs, I itemjobs and C cores (per worker node). We define $T(U,I,C)$ as:

$$T(U,I,C) = s + \frac{P_U}{U} + \frac{P_I}{I} + \frac{P_C}{U \times I \times C} \tag{4.5}$$

If we complete the baseline numbers (as percentages) from Table 4.6 in equations (4.4) and (4.5), we can express the speedup model for our algorithm in terms of userjobs, itemjobs and cores:

$$S_p(U,I,C) = \frac{100}{.02339 + \frac{.01675}{U} + \frac{.00001}{I} + \frac{99.95985}{U \times I \times C}} \tag{4.6}$$

To validate this speedup model, we compared the predicted values with empirically determined speedup values. For a number of (independent) variations of userjobs, itemjobs and cores, we ran the complete recommendation process and measured $T(U,I,C)$. Since also $T(1,1,1)$

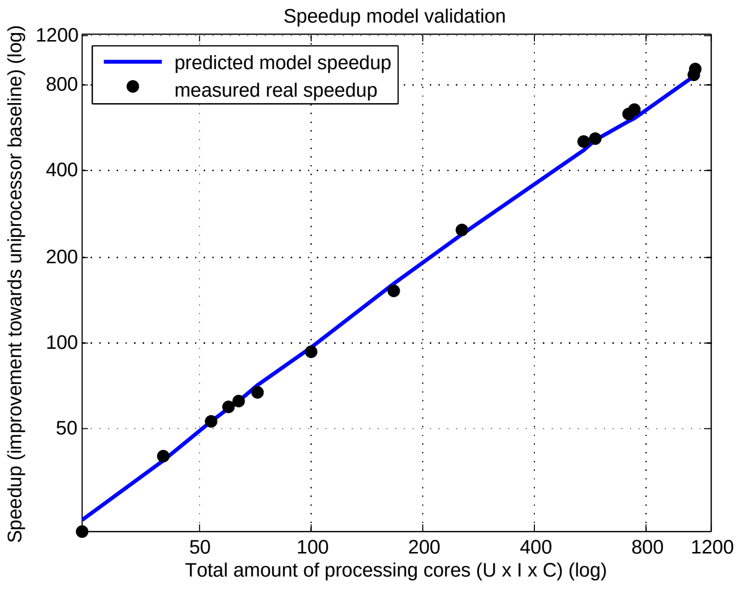


Figure 4.17: Validation of the speedup model by comparing model-predicted speedup values with empirically determined speedup values. Note that both the X-axis and the Y-axis are on a logarithmic scale.

is known, we were able to compute the actual speedup and compare it to the predictions of the model. Fig. 4.17 visualizes the model together with the actual speedup values for the different variations. When we perform a simple regression analysis, we find that our model has an $R^2 = 0.9982$. The maximum speedup error in the model was 9% (for $(U, I, C) = (2, 2, 6)$), the average error rate was 4%. We therefore consider our model to be valid with an average error rate below 5% and usable as predictor for the speedup value of our recommender system.

With this model we are now able to explore speedup values for any desired range of (U, I, C) settings. Fig. 4.18 shows the speedup values for a different number of cores with an equal amount of userjobs and itemjobs varying from 1 to 2048. It is interesting to see that the speedup value converges to a number between 4000 and 4500. This limitation is the consequence of the fraction of non-parallel code as predicted by Amdahl's Law [97, 118] which states:

$$\lim_{N\to\infty} S_p(N) = \frac{1}{s} \ . \tag{4.7}$$

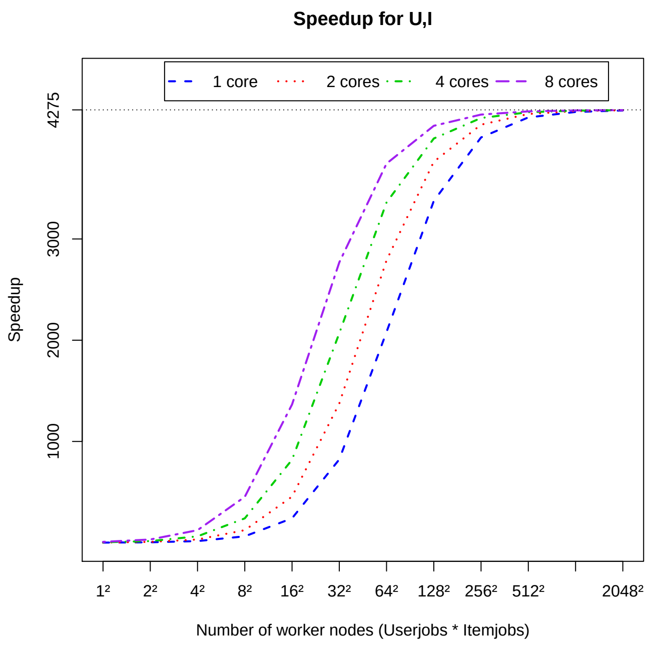


Figure 4.18: Speedup as predicted by the speedup model for various (U, I) settings.

The execution time of the parallelizable parts of the code can be made infinitesimally small (for large values of worker nodes N), so that the resulting final execution time consists almost completely of (and is therefore also limited by) the serial fraction of the code. If we calculate Formula (4.7) with the known value of the serial fraction (Table 4.6), we find our speedup limitation (Formula 4.9).

$$\lim_{U,I,C \to \infty} S_p(U, I, C) = \frac{1}{s} \tag{4.8}$$

$$\lim_{U,I,C \to \infty} S_p(U, I, C) = \frac{100}{0.02339} = 4275 \tag{4.9}$$

Although the performance of our algorithm may be limited by a speedup value of 4275, this limit is only reached for very high values of userjobs and itemjobs (>512). If both the amount of userjobs and itemjobs is 512, the resulting number of jobs will be 512^2(=262 144). If every job is mapped on a worker node, this would require an unreasonably large cluster. Therefore we conclude that the scalability of our algorithm will be limited by the availability of computing hardware before its theoretical limit of speedup is reached.

We define parallel efficiency ϵ_p for our system in terms of userjobs, itemjobs, and cores as the following.

$$\epsilon_p(U, I, C) = \frac{S_p(U, I, C)}{U \times I \times C} \tag{4.10}$$

Using the speedup model as input, we are able to explore the scalability of our algorithm in terms of efficiency for any desired range of (U, I, C) settings. If we would run our recommender system on a cluster with 200 worker nodes each with 8 processing cores (U, I, C equal to 10, 20, 8), we can expect a speedup value of 1142 which gives us a parallel efficiency of 71.4%. If we now compare our parallel efficiency with results found in similar related work, such as 90% efficiency for a Hadoop-based item-based CF recommender with 8 nodes [104], we find that our solution easily achieves higher efficiency values (and so higher speedup values) for the same configuration (U, I, C equal to 4, 2, 1), namely 99.8%.

In Fig. 4.19 we plotted both speedup and efficiency (with four processing cores). Both graphs intersect at 50% for a number of worker nodes equal to 32^2(= 1024). The targeted minimum parallel efficiency for a given scenario will ultimately depend on the availability of the hardware infrastructure and related costs.

It is clear that to come to an appropriate amount of userjobs, itemjobs and cores, a trade-off will have to be made between how fast the algorithm comes to a solution and how efficiently resources are used. Hager et al. [97] suggest the construction of a cost model and minimizing the product of walltime and infrastructure cost as a sensible balance. As mentioned before, the available RAM of worker nodes may also be limited so that a certain minimum number of userjobs will be needed to meet hardware requirements. More userjobs implicates more fragmentation of the user input data and therefore reduced RAM requirements per worker node.

For general purposes, we suggest to set the number of used cores (C) to the number available in a single computing node and the number of userjobs (U) so that the associated RAM requirements match the

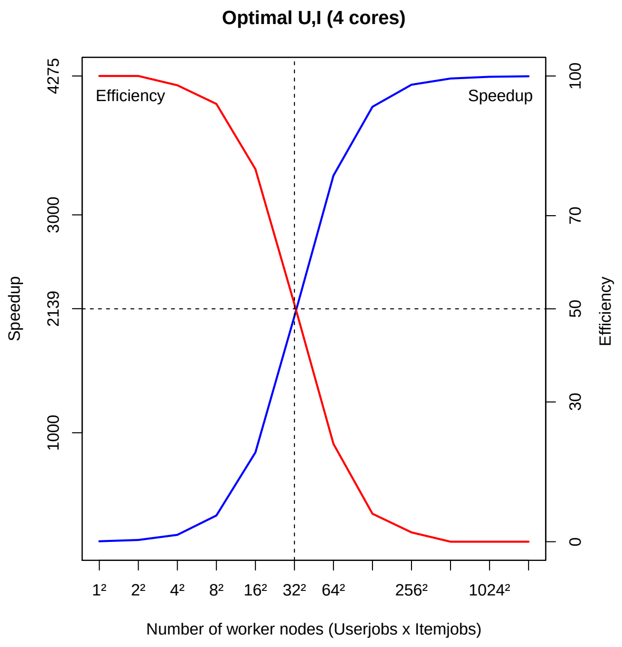


Figure 4.19: Modeled speedup and efficiency for worker nodes with 4 cores and various (U, I) settings.

available RAM. The number of itemjobs I can then be set such that the total number of jobs is a factor of the total number of computing nodes.

Our proposed in-memory algorithm offers complete flexibility as to the number of subtasks the recommendation problem should be partitioned in, and can therefore easily be optimized for any given set of cost constraints or for any given hardware configuration.

4.4.7 The Performance on another Dataset

The true performance of the system will in the end be very dependent on the available hardware infrastructure, applied (U, I, C) settings and the dataset at hand. To gain more insight into the generalizability of our results, we did a small experiment with varying (U, I, C) settings on another dataset. We used a dataset borrowed from our previous experiments on a cultural events website. This dataset contained 40,000 ratings, from 10,000 users on 40,000 events, which is considerably smaller than the MovieLens 10M dataset.

The resulting speedup and efficiency values (Table 4.7) show that the efficiency of the performance of the recommender system on this dataset is slightly less than for the MovieLens dataset. For the calculation with 200 nodes, each having 8 cores, the efficiency drops from 71.4% (for MovieLens) to 52%. This is to be expected since the cultural dataset is much smaller (about 50 times). A smaller dataset indicates less work, while the typical overhead of reading data and executing the job division stays about the same, thus decreasing the overall efficiency.

These results however indicate an interesting transition from an offline recommendation scenario to a possible real-time recommender, without the need for a customized incremental recommendation algorithm. The full recommendation calculation on a single machine takes about two hours, which would force the recommender system to calculate recommendations in an offline batch scenario. However, if a computing infrastructure with 200 worker nodes (8 cores per node) is available then the calculation takes only 8 seconds which paves the way towards real-time (content-based) recommendation for an online recommender. So while our distributed recommender will show higher efficiency values for bigger datasets, even for small datasets the merits of increasing the recommendation calculation performance are clear.

U	I	C	Time (s)	Speedup	Efficiency (%)
1	1	1	6691	1	100
2	2	1	2020	3.3	83
4	4	1	587	11.4	71
8	8	1	164	40.8	64
16	16	1	43	155.6	61
32	32	1	12	581.7	57
2	2	4	525	12.7	80
4	4	4	157	42.5	66
8	8	4	42	159.3	62
16	16	4	12	557.5	54
10	20	8	8	836.2	52

Table 4.7: Performance results of the system with a smaller dataset.

4.5 Caching for In-Memory Neighborhood-Based Models

A fast execution time can be the differentiating factor between real-time incorporating user feedback and daily batch processing recommender systems. In this section we try to reduce execution time and allow flexible memory usage by introducing caching principles for in-memory neighborhood-based recommendation algorithms.

A cache enables rapid access to popular or frequently accessed data [119]. Its use can significantly speed up data throughput and therefore also greatly impacts the total execution time of algorithms that are largely data dependent, such as recommender systems.

We focus on collaborative filtering systems, more specifically nearest neighbor (KNN) algorithms because they are most likely to benefit from cache enhancement. The general idea behind collaborative filtering algorithms is that community knowledge can be exploited to generate more accurate recommendations for individual users [40]. KNN algorithms try to harvest this community information to identify similar users or items in the system. Similarity can be computed in many ways, but most often involves comparing the ratings either given by users or received by items and the calculation of some kind of similarity metric. The rating behavior of such similar neighbors can then be used to extrapolate ratings for new users or items. These types of algorithms will have to determine the pair-wise similarity values for all users or items in the system, values which will then be re-used many times during the recommendation calculations. Such similarity values seem therefore very well suited for caching.

Research literature is rather limited on the topic of caching and recommender systems. To our knowledge, two main contributions exist that report on enhancing recommendation response times through the use of caches. The work of Qasim et al. has introduced the concept of partial-order based active cache [120] in which a caching system is constructed that allows to estimate nearest neighbor type queries from other queries in the cache. The usefulness of the cache is thereby extended beyond exactly matching previously-cached entries. The caching structure is intended to prevent recommendation lists to be calculated by estimating them based on previous answers in the cache.

Another cache-enhanced recommender system was the genSpace recommender system presented by Seth et al. as a prefetching cache that prefetches all recommendations in order to prevent slow on-demand recommendation calculation [121]. Their approach differs from ours since their cache rather prevents unnecessary recommendation recalculation while we attempt to speed up the calculation of the complete recommendation process (rating prediction for all users and items) in order to reduce overall calculation time without compromising recommendation accuracy.

In the next sections we will define a user-based collaborative filtering algorithm and show why and how this algorithm may benefit from caching internal values.

4.5.1 The UBCF Algorithm

To be able to experiment with different caching modes for recommendation algorithms, we implemented the well known user-based collaborative filtering algorithm (UBCF) [40]. This algorithm is widely accepted and commonly used in recommender systems in various domains. Here we apply it to predict a rating for all user-item pairs, based on the ratings of similar users (i.e., neighbors). The high-level algorithm pseudocode can be found in the appendix (Algorithm 3).

Our implementation employs a simple weighted average scheme to come up with the final recommendation value. Pearson correlation was used as similarity metric and the neighborhood size was restricted to the top 20 neighbors. A straightforward linear transformation was employed to rescale the Pearson correlation value from [-1,+1] to [0,1] to simplify the calculations. Note that no intermediate storage is used, all calculations are performed in-memory.

4.5.2 Similarities Usage Frequency

To calculate the predicted rating of a user for an item, neighboring users will have to be determined. That in turn requires the pair-wise similarity of that user with every other user in the system that rated the item. The most important line of code in the UBCF algorithm pseudocode is line 19:

$$neighbor\_similarity \leftarrow Pearson(user, neighbor)$$

Using a code profiler, we analyzed the runtime of our recommender system and found that 78% of the total execution time was spent calculating these similarity values. Therefore, we want to construct a caching system fit for the storage of these values in order to reduce recalculation overload on the one hand and memory (RAM) requirements on the other hand.

We start off with the hypothesis that not all calculated similarity values are equally useful. So when the recommendation values for all user-item pairs are calculated, in the end some user similarity values will have been used more than others.

To gain insight into the distribution of the usage of user-user similarity values, we set up an experiment using the MovieLens (100K) dataset. We calculated the recommendation value for each user-item pair in the dataset (without caching) and counted how frequent every user-user similarity was used. User similarity (with Pearson correlation) is a symmetric relationship and so the total number of calculated user-user couples can be defined as $\frac{943*942}{2}$ (self-similarities are not taken into account). Fig. 4.20 shows every one of these couples on the X-axis and their corresponding usage frequency on the Y-axis.

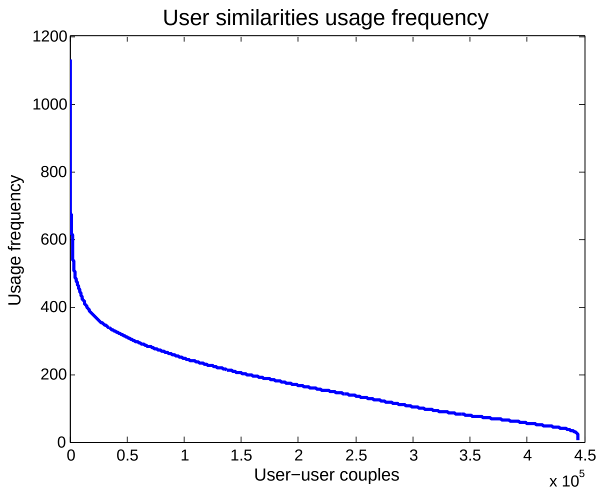


Figure 4.20: Every user-user couple in the system with its corresponding similarity usage frequency after having calculated all recommendation values for the MovieLens 100K dataset. Couples are ordered by descending frequency.

Fig. 4.20 does not show a horizontal flat line but rather a long tail curve and so we find our hypothesis confirmed: some user similarities are more frequently used than others in the recommendation calculation process. The question now remains how we can use this knowledge to our advantage. In previous sections the problem of limited RAM was already mentioned. Because of this limitation it may be impossible to simply compute all user-user similarities and keep them available in memory during the recommendation calculation. So if we can store only a limited amount of pre-calculated similarity values in memory, it may be a good idea to make sure the most interesting values are stored. If we define most interesting as 'most used' then we need a way of predicting how much a given user-user similarity value will be used throughout the recommendation process.

The usage frequency of user similarity values will be largely depending on the number of ratings provided by each user. Users with a large number of ratings may show overlap (rated the same items) with more users and their similarity will consequentially be needed more often. To examine the correlation of the number of ratings of users with their similarity usage throughout the recommendation process, we set up an experiment with three simple usage prediction formulas: *min, max*, and *sum*. For each user-user similarity value we tried to predict its usage by applying each of these aggregation operators on the number of ratings of both users.

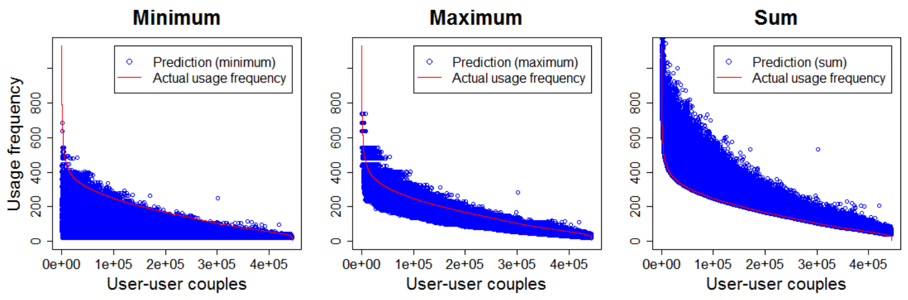


Figure 4.21: Prediction of the usage frequency of user-user similarity values by means of three aggregation operators together with the actual empirically measured usage frequency.

Fig. 4.21 plots the prediction operators applied to all user-user similarity pairs in the system together with the empirically measured actual usage value. As expected, we can see a correlation between the total number of ratings of a user-user pair and its usage frequency. The *maximum* operator seems to be the best estimation operator but has still a far from perfect accuracy.

To improve this accuracy, we looked into deterministically computing the usage frequency instead of predicting it in a heuristic way. We considered the abstract user similarity pair $(u_x,\ u_y)$. This similarity may be used while calculating the recommendation pair $(u_x,\ i)$ with i an item that u_x has not rated. To determine the recommendation value of $(u_x,\ i)$, user similarities of u_x are needed for every user that has rated item i (see UBCF algorithm in Section 4.5.1). So the similarity pair $(u_x,\ u_y)$ will be calculated a number of times equal to the number of items that u_y has rated, but that are not rated by u_x. Since here the user similarity has symmetric properties, the user similarity of $(u_x,\ u_y)$ and $(u_y,\ u_x)$ will be the same, and both should be taken into account. To conclude we can state that the number of times a user similarity pair $(u_x,\ u_y)$ will be used during the UBCF recommendation process, will be equal to the sum of the number of items rated by u_y but not by u_x and the number of items rated by u_x but not by u_y. We can reformulate this as the cardinality of the inverse intersection (which is mathematically referred to as the *symmetric difference*) of the sets of rated items by u_x and u_y as shown in Fig. 4.22.

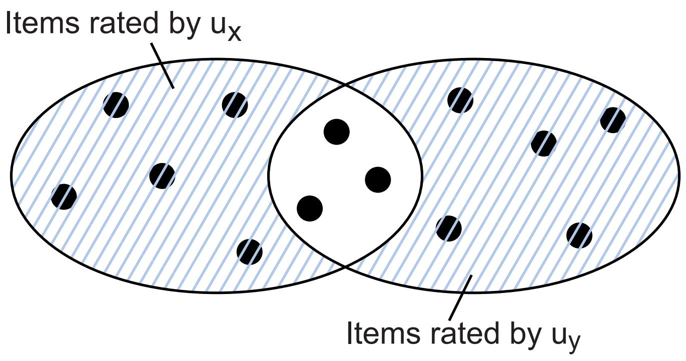


Figure 4.22: Venn diagrams indicating the items rated by users u_x and u_y. The cardinality of the inverse intersection (or *symmetric difference*) corresponds to the usage frequency of the similarity value $(u_x,\ u_y)$.

We can now experiment with a caching system that incorporates this usage frequency knowledge and measure how it affects overall performance.

4.5.3 Caching Algorithms

To measure the impact of caching on the performance of the UBCF algorithm, we compare two different caching strategies. On the one hand we work with the existing LRU (Least Recently Used) caching principle and on the other hand we present our self-designed 'SMART' cache.

The LRU caching principle is a commonly used caching system where entries that have been used the least recently will be overwritten when the cache is full. This approach follows the temporal locality principle that recently requested data has a high probability to be requested again in the near future [119]. A disadvantage of LRU is that only the time of the data access is considered and not the frequency. Therefore we devised a 'SMART' cache which takes this data frequency information into account.

The SMART cache is a type of priority cache that incorporates information about how much an entry will be accessed throughout the program life cycle. Every cache entry is associated with a priority that reflects the number of times the entry will be accessed. When the cache is full, a new entry with a larger priority (i.e., predicted number of accesses) will overwrite an existing one with the lowest priority.

4.5.4 Experimental Results

The first thing to measure is the performance of the UBCF algorithm when using either the LRU or SMART caching strategy. As baseline we considered the UBCF algorithm implementation without a cache. In that case no user similarities are stored and they have to be recalculated every time they are used throughout the recommendation calculation. We compared the execution time of this baseline algorithm to the UBCF algorithm with an LRU cache or SMART cache for storing the user similarities during execution. Recommendation values (predicted ratings) were calculated for all user-item pairs in the MovieLens (100K) dataset. The experiment was repeated for cache sizes of 20%, 40%, 60%, 80% and 100%. The cache size is expressed as a percentage of the total number of user-user similarity values (which is $\frac{n*(n-1)}{2}$ for n the number of users in the system). A cache size of 100% therefore indicates that all user-user similarity values can be stored in memory and no values need to be recalculated. Fig. 4.23 shows the results in terms of speedup (i.e., execution time reduction compared to the baseline) for the different settings.

The simple LRU caching principle performed considerably better than the SMART system for the low cache sizes (20% and 40%). When the cache size grows, this difference decreases, and with cache sizes nearing 100%, the SMART approach overtakes LRU in terms of speedup towards

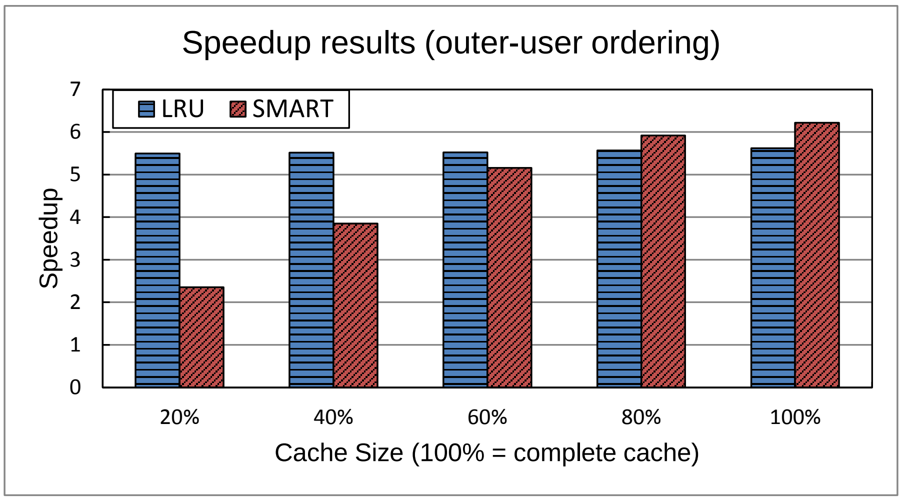


Figure 4.23: Speedup results for the LRU and SMART caching approaches towards the no-caching baseline for different cache sizes.

the baseline. Interestingly, the performance of the LRU system remains stable between speedup values of 5 and 6 (in comparison to the SMART system) for the varying cache sizes. This seems to indicate that the LRU caching system has an optimal cache size (where performance gain saturates) below the 20% limit.

We repeated the experiment using only the LRU cache and tested with smaller cache stepsizes. Fig. 4.24 shows the results of an interesting range of cache sizes between 0.1% and 0.6%. Zoomed in on that range, the saturation effect of the speedup towards the baseline is clearly visible. While the SMART caching approach seemed to linearly improve with an increasing cache size, the LRU reaches a saturated maximum speedup value at a cache size between 0.2% and 0.3%. This range seems to correspond with the amount of user similarities associated with one user. For our dataset, there are 943 users and so 942 $(n-1)$ user similarities per user. To store 942 user similarities in the cache, a cache size of 0.21% would be needed. This 0.21% corresponds to the optimal cache size in the range 0.1% and 0.6%.

The performance of a cache depends partly on the order in which new entries are provided, therefore it is interesting to see how the 2 caching systems perform under altered job execution orderings. The order in which the user-user similarity values will be inserted in the cache depends greatly on how the UBCF algorithm loops over the different

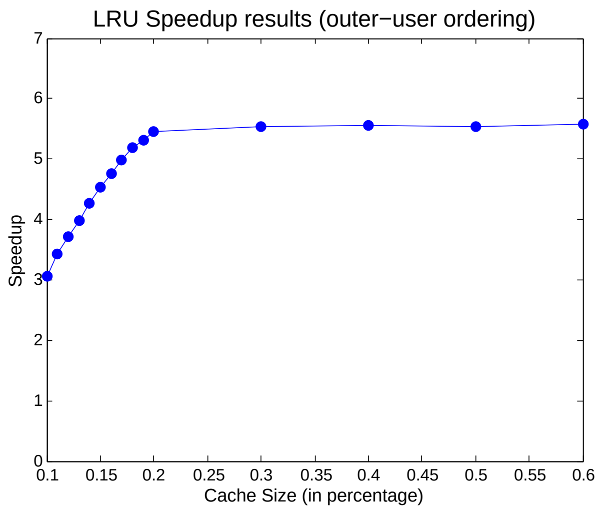


Figure 4.24: Speedup results for the LRU cache with smaller cache sizes to reveal the performance gain saturation point.

user-item pairs. In our reference implementation we calculated the recommendation value for every user for every item in that specific order. So the user-item pairs will be sequentially $(u_1,\ i_1)$, $(u_1,\ i_2)$, ... $(u_1,\ i_{n_i})$, $(u_2,\ i_1)$, $(u_2,\ i_2)$, ... $(u_2,\ i_{n_i})$, ... $(u_{n_u},\ i_{n_i})$ for n_u and n_i being respectively the total number of users and items. Since every user is handled sequentially, many similarities will be re-used in short intervals. This behavior is exactly what the LRU caching approach takes advantage of and explains its success compared to the SMART cache.

To study the impact of the job order on the caching performance, the experiment was re-run with an altered job ordering. To change the order in which similarities would be inserted in the cache, we simply switched the first two lines of the pseudocode fragment. Because of the switch, user-item pairs are processed iterating over the items first. We refer to this approach as the 'outer-item' order (as opposed to the 'outer-user' order, which we started off with). The corresponding user-item pairs will now be sequentially $(u_1,\ i_1)$, $(u_2,\ i_1)$, ... $(u_{n_u},\ i_1)$, $(u_1,\ i_2)$, $(u_2,\ i_2)$, ... $(u_{u_n},\ i_2)$, ... (u_{n_u}, i_{n_i}). Fig. 4.25 shows the speedup results for this altered scenario.

As expected, the performance of the LRU cache is greatly reduced and for every cache size the SMART cache shows faster speedup values. Although the SMART cache is faster than the LRU method, it is still

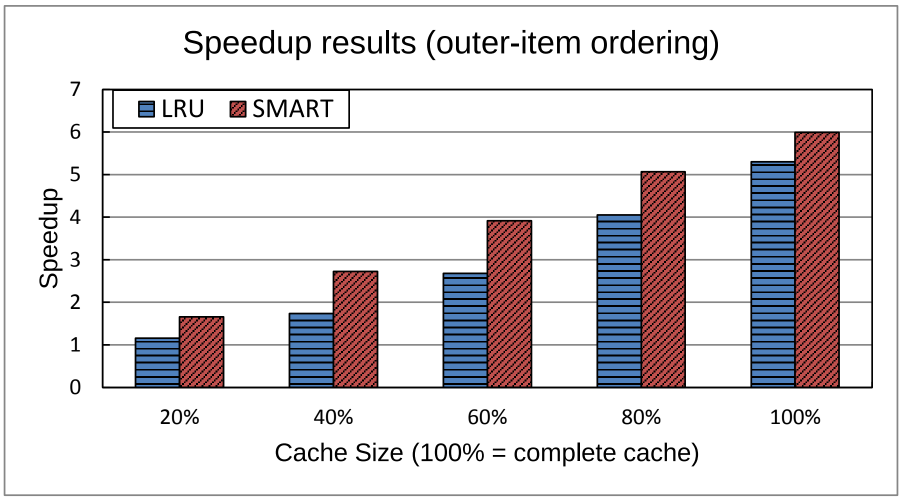


Figure 4.25: Speedup results for the LRU and SMART caching approaches (for different cache sizes) with a reversed user-item pair handling execution ordering.

somewhat slower than the speedup values of the SMART cache in the outer-user ordering situation. In general the speedup values are below those of the outer-user ordering, but the SMART cache seems less affected (than LRU) by the change in ordering. The LRU cache shows no indication of saturation as was the case for the outer-user situation.

We repeated the experiment with a third alternate job ordering situation where all user-item pairs were processed in random order. The aggregated results of multiple runs turned out to be almost identical to the outer-item results.

Our experimental results indicate how the order in which user-item recommendation values are calculated, can dramatically impact the LRU cache performance and therefore also the total execution time. Optimal results were obtained when calculating the recommendation values of each user for every item sequentially before moving on to the next user (outer-user strategy). The LRU-enhanced UBCF algorithm performed between 5 and 6 times better in that situation than the no-cache baseline and required a cache size of only 0.2% (vs. the SMART approach which required a cache size of 60% to obtain similar results). For a random job (and outer-item) execution ordering on the other hand, the SMART approach proved best in terms of stability and performance.

Although this work focused mainly on in-memory algorithms, these results may as well be useful for other situations where caching strategies can be applied to user-based collaborative filtering algorithms e.g., caching similarity values to reduce database access.

4.6 Conclusion

In this chapter we focused on the recommendation calculation process from a high-performance viewpoint. We approached the problem of distributing the recommendation process on a HPC infrastructure in two ways. First, we illustrated *functional parallelism* by splitting up the recommendation process in different functional parts and showed how each part could be distributed and computed independently using a file-based data approach. The resulting execution time scaled linearly with the applied computing hardware. Because the functional parts were however depending on the output of previous phases, the performance of our functional parallelism approach was inherently limited. Execution processes needed to wait for each other (causing load imbalance issues), and slow disk access was required at the start and end of every phase.

We then presented a second approach towards distributed recommendation calculation, applying an in-memory strategy. A content-based recommendation algorithm was used for illustration purposes, but we showed how the approach could be generalized to other recommendation algorithms as well. Here we applied the concept of *data parallelism* by splitting up the required data in blocks such that every chunk of work could be processed independently leading to embarrassingly parallel problems which can easily be distributed across worker nodes. We optimized the work to be as load balanced as possible to increase overall parallel efficiency. We showed how the in-memory approach obtained higher parallel efficiency values than state-of-the-art methods and we presented a model for predicting the efficiency and speedup given a specific hardware configuration.

Finally, we investigated caching mechanisms for in-memory neighborhood-based recommendation models which are prone to frequent reusing mid-computation values such as similarity values. By carefully structuring and ordering the recommendation calculation process, execution time speedup values up to 6 were obtained using a simple caching strategy (i.e., LRU) and low cache sizes (i.e., $O(n)$ with n the number of users).

Overall we have shown that the problem of increasing dataset sizes for recommender systems can be tackled by applying various high-performance techniques such as parallelism, distributed computing and caching strategies. By carefully analyzing the calculation process and its data requirements, we were able to significantly increase the performance of such complex data focused algorithms without the use of restraining technologies as MapReduce.

Chapter 5

Offline Optimization of Personalized Hybrid Recommender Systems

5.1 Introduction

Years of research contributions by hundreds of researchers, have led to an abundance of recommendation algorithms that can be used to combat information overload. Recommendation algorithms come in all sorts and sizes, from really simple ones as *SlopeOne* [122] to extensive mathematical-based ones as *SVD++* [61]. Algorithms can be based on collaborative filtering principles, content-based, knowledge-based, demographic-based, etc. More recently also context-based [123, 124] and social-based [125] algorithms are starting to show up. Each and every one of these algorithms have their own advantages, downsides and optimal use cases and scenarios. As stated earlier, also the availability of recommendation frameworks (or platforms) that offer out-of-the-box recommendation solutions is on the rise. It seems that, by eagerly tackling the information overload problem with new methods, the recommender domain in itself is becoming overloaded with available algorithms which makes it more difficult (for recommender system administrators) to select the right algorithm for the job.

Instead of using one algorithm, sometimes multiple algorithms are combined into a so-called *hybrid* recommender system. Hybrid recommender systems have long been popular, and are widely used in many real-world applications because of their obvious advantages over individual recommendation algorithms. By combining and integrating different types of recommendation algorithms, hybrid recommenders are able to overcome the drawbacks associated with each of them individually [38, 39, 126]. A problem that most hybrid recommenders nowadays are facing, is that they are inherent static in nature. Very often they are trained or manually tweaked before deployment, but at runtime their configuration remains the same. Their static nature prevents them from being deployed in other scenarios, with other algorithms, or for other (types of) users.

Because hybrid systems are cumbersome to configure (often done manually), the number of incorporated individual algorithms is usually rather limited to two or three algorithms at most. Since hybrid systems inherit the properties of their individual components, it seems more interesting however to have hybrid recommenders composed of more algorithms than just a few. Ideally, a hybrid recommender system would include all existing recommendation algorithms and be capable of intelligently deciding what algorithm (or what combination of algorithms) generates the most interesting results for any given user in a system.

In this chapter, we strive towards the ideal hybrid recommender system which automatically fine-tunes itself based on given individual recommendation algorithms and user input data. We focus specifically on two hybridization techniques i.e., hybrid switching and weighted hybridization and include up to 10 individual algorithms in our experiments.

Research Questions

- How can hybrid recommender system configurations be automatically generated?
- How can we optimize hybrid configurations?
- What are the optimization limitations?

5.2 Related Work

Burke et al. [37] was one of the first to categorize hybrid recommender systems in function of their combining strategies: *weighted, switching, mixed, feature combination, cascade, feature augmentation,* and *meta-level.* Every combining strategy comes with its own specific properties and consequences. Two of the most commonly combined recommendation algorithms are the content-based (CB) and the collaborative filtering (CF) approach [127–129], because they tend to complement each other in various ways. Cornelis et al. [108] for example combined elements of CB and CF to recommend time-specific items (i.e., events), which get rated only after users have consumed (i.e., visited) the items. Since in these types of hybrids, specific properties of the individual algorithms are exploited, they lack extendibility and easy integration of more (and other types of) algorithms. Very often only two algorithms are combined using a simple combining strategy, e.g. [130] where the predictions of two types of collaborative filtering systems are combined linearly (*weighted*) using the following formula.

$$P = \alpha \times P_{algo_1} + (1 - \alpha) \times P_{algo_2}$$

Hybrid solutions that integrate more than 2 individual recommendation algorithms often apply *ensemble learning* techniques. The general idea of ensemble learning is to combine multiple (smaller) individual models to obtain a global model which performs better than the individual ones. For a thorough introduction and analysis of ensemble-based systems in decision making, we refer to the excellent introduction by Polikar [131].

A recent hybridization technique is feature-weighted linear stacking (FWLS) [132], which continues on the original concept of stacking [133], where multiple recommendation models are stacked (or *blended*) together. The advantage of the stacking technique is that individual components are loosely coupled, which allows easy integration of new algorithms and tuning of the end results by adjusting the individual coefficients (i.e., weights) of the models. These coefficients are usually determined by taking into account so-called *meta-features*, which are metrics describing some specific properties of the dataset at hand. In FWLS, the coefficients associated with the models are parametrized as linear functions of the meta-features. The STREAM (Stacking Recommendation Engines with Additional Meta-Features) system [134] experimented with eight different meta-features, and ultimately found the number of *user ratings* and *item ratings* the most interesting. These metrics can be used to differentiate the usability of individual recommendation algorithms for specific recommendation scenarios. However, for the STREAM approach to be successful, expert knowledge about the properties of the integrated recommendation algorithms and its influencing factors is required (i.e., no black box algorithms). The most famous application of stacking is the winning entry of the Netflix Prize[1], where the *BellKor's Pragmatic Chaos* team stacked over hundred different models together into one blend (in fact even a blend of blends) [47–49]. Their optimization target however was overall RMSE, and so the weights for the individual algorithms were the same for every user (i.e., static hybrid).

Ensemble systems often try to optimize general recommendation metrics such as RMSE. Recent research however, is moving away from generalizing users, and towards a *per user* focus. Ekstrand et al. [41] showed that recommenders fail (and succeed) on different items and users (although their focus was on switching hybrids). Hybrid recommender systems would obviously benefit from being able to predict which recommendation algorithm works best for which user. The importance of individuality of users was also noted by the work of Kille et al. [42], in which they tried to model the difficulty of generating recommendations for individual users.

The AdaRec system [126] proposed a dynamic hybrid strategy that modified its prediction strategy at runtime to cope with unique domains, trends and user interests. Focus however, was on a switching strategy that selected the most suitable recommendation algorithm rather than composing a dynamic weighted hybrid that incorporates input from all algorithms. In [135], the authors describe their system *Semantic-Movie* which integrates multiple recommendation approaches (*recommender agents* in their words) into a single agent ensemble. Combining weights are generated by default, but can be manually adjusted per user. Automatically adjusting weights (through a *learning component*) according to user feedback was however deferred to future work.

The topic of automatically adjusting user-specific weights for dynamic ensembles has been touched by Bellogín [39, 136]. From an information retrieval perspective, he proposed adaptations of query performance techniques to define performance predictors in recommender systems. Using these predictors, recommendation strategies can then be dynamically fine-tuned. The selection of predictors and individual recommenders to be used in the ensemble is limited by specific constraints e.g., the predictors should correlate positively with the performance of not all but some recommenders.

Our goal is to concentrate on the most relevant ongoing topics of hybrid

[1]`http://www.netflixprize.com`

recommender research. We focus on dynamically optimizing user-specific hybrid systems using machine learning approaches. We compare the popular hybrid switching approach with the weighted hybrid strategy and show how both can be optimized, yielding a performance boost in comparison to individual recommendation algorithms or static hybrid systems. Individual recommendation algorithms are considered *black boxes* and therefore any type can be used.

5.3 General Architecture

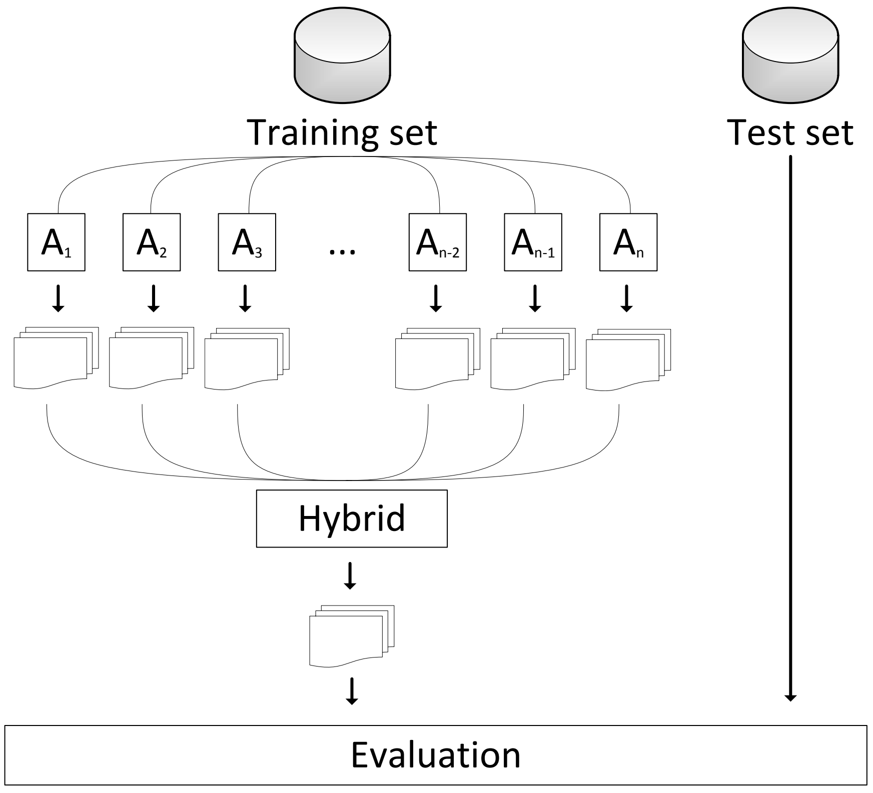


Figure 5.1: The high-level architecture of the hybrid system. Rating data is split in a training and test set. Multiple recommendation algorithms then generate recommendations, which are combined into one set of recommendations and finally evaluated based on the test set.

Our general architecture corresponds to a common hybrid recommender system layout as depicted in Fig. 5.1. A rating dataset is divided into a training and test set, which are then respectively used as input for the hybrid recommender and for the final evaluation of the system. The hybrid recommender system consists of multiple recommendation algorithms which run in parallel on the provided data and are finally aggregated into hybrid recommendation results.

For our experiments we will be using the MovieLens 100K dataset which has the advantage that every user in the system will have at least rated 20 items. Since we are tackling the hybridization problem on a user-specific level, we also want to evaluate user-specific and so training and test sets will contain all users but only a subset of their ratings. For each user we adopt a split of 60% (train) and 40% (test) ratings. Because every user has rated at least 20 items, every user will have a minimum of 8 ratings in the test set.

To serve as individual recommendation algorithms in the hybrid system, we selected the following 10 algorithms from the rating predictors available in the MyMediaLite framework. For each of these algorithms, default settings were used as set in MyMediaLite version 3.09.

- BiasedMatrixFactorization
- MatrixFactorization
- FactorWiseMatrixFactorization
- SigmoidSVDPlusPlus
- BiPolarSlopeOne
- SlopeOne
- UserItemBaseline
- UserKNN
- Constant1
- Constant5

The output of each of these single algorithms is directed to the 'Hybrid' module, where results are collected and aggregated. We specifically focus (and compare) two common hybridization strategies: a *switching* approach, where only the best algorithm is selected, and a *weighted* approach where all algorithms contribute to the final recommendations according to a specific weight.

In the end, the system is evaluated according to a k-fold evaluation procedure. The training dataset is used throughout the system to calculate the individual and finally the hybrid recommendation values. Subsequently, the results are compared with the test set. This process is repeated 10 times (i.e., 10 folds).

5.4 Offline Optimization for Hybrid Recommenders

We are attempting to optimize hybrid recommender systems in an offline setting. To be able to handle the problem of selecting or composing an optimal hybrid recommender, we consider our problem as an optimization task.

$$\underset{x}{minimize}\ f(x)$$

Here, the goal is to minimize a defined objective function $f(x)$ by providing it with an optimal input x. For the hybrid recommendation use case, the objective function could be an evaluation metric which we want to optimize (e.g., accuracy, diversity, serendipity, etc.) and the input x the hybrid recommender. The better the recommender, the better the values of the objective function.

Since we are working in an offline setting, we need to select an objective function that can be evaluated offline, or in other words, without requiring additional user input. For the purpose of demonstrating the offline optimization procedure, we optimize the accuracy metric: *Root Mean Squared Error* (RMSE).

$$RMSE = \sqrt{\frac{1}{|\tau|} \sum_{(u,i) \in \tau} (\hat{r}_{ui} - r_{ui})^2}$$

In RMSE, system-predicted ratings $\hat{r}_{ui}$ are compared with true ratings r_{ui} contained in a certain test set τ of user-item pairs (u, i) [137]. We adopt RMSE because it is one of the most popularly used evaluation metrics in the recommender systems domain and is easily computed in an offline context. Recent research [53, 63, 138, 139] shows that although recommendation accuracy is principal to achieve user satisfaction, it is not the only important metric and often metrics as diversity, transparency or trust should be considered as well. We note however, that our offline optimization strategy is not limited to RMSE, and can incorporate any desired metric as long as it can be calculated offline on a given set of ratings.

5.4.1 Evaluating Optimization

Aside from defining the objective function (i.e., RMSE), we also need data to calculate $f(x)$ given a certain x. Two available datasets are the training set and the test set. The test set is clearly unusable since we want to avoid *tuning to the test set*; the test set should only be used in the final evaluation of the complete system and not in intermediate optimizing iterations, to guarantee evaluation fairness. We therefore focus on using the training set as input for our optimization task. In other words, we intend to optimize our hybrid system in terms of RMSE on the training set. We then hypothesize that, a decrease of RMSE (lower RMSE is better) on the training set will result in a similar decrease on the test set. This may not be the case if the optimization that was *learned* from this procedure is not generalizable i.e., too specific to the training set.

To prevent overfitting the training set, we do not train our optimization on the full training set, but rather on 10 distinct subsets of this training set. Such a subset divides the training set into a smaller subtraining set and subtest set, much like the general 10-fold evaluation procedure of the system, with a user-specific ratio of 60/40 (Fig. 5.2). The objective function results are then averaged (arithmetic mean) over these 10 subdatasets. A downside of this optimization procedure, is that it requires to run the individual algorithms on 11 different datasets: 10 subfolds of the training set (to optimize the hybrid), and once on the full training set (for the final recommendations).

With a defined objective function and data to train (and evaluate) on, we can now proceed to the optimization of our hybrid recommender system. We explore both the *switching* and *weighted* hybrid strategies.

5.5 Hybrid Switching Strategy

The *hybrid switching* technique entails the switching between different recommendation algorithms by means of a switching strategy [37]. It is a very easy and straightforward method since in the end only one algorithm will contribute to the final recommendations. The selection of the 'best' recommendation algorithm for a given scenario often depends on objective metrics such as metadata availability (e.g., content-based when item features are available), the number of total ratings (e.g., collaborative filtering when a large number of items have been rated), etc.

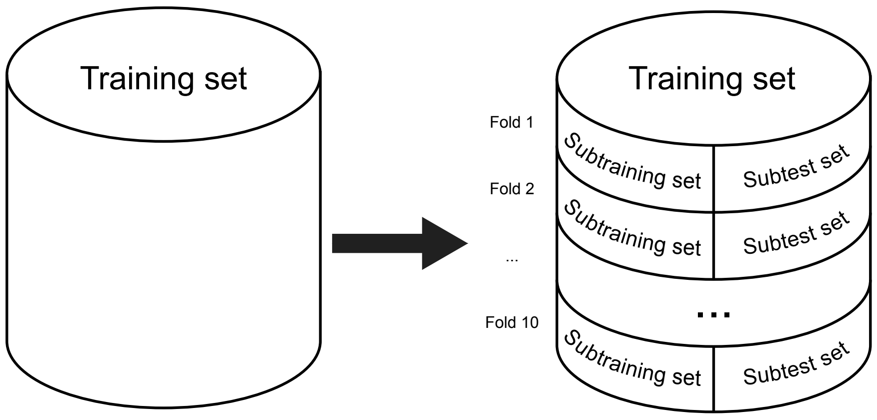


Figure 5.2: The training set is split into 10-folds with a 60/40 subtraining set and subtest set, which are then used to optimize the hybrid configuration.

However, since we are optimizing hybrid recommender systems at user-level, the switching strategy should also be implemented at user-level so that for every user, a single best algorithm can be selected.

Our user-specific switching selection strategy, starts with the determination of a default algorithm. This default algorithm serves as a fallback option when no clear 'best' algorithm could be detected for a user. The default algorithm is determined by evaluating RMSE values for all algorithms over all users on the subtest datasets, and so will be the same for every user.

Next, for every user a 'best' algorithm is detected among the available individual recommendation algorithms. Best in this case, is defined as providing the best (averaged out) RMSE values on the 10 subtest datasets. Aside from best RMSE, also the variance among the 10 (i.e., one for every subtest dataset) RMSE values is taken into account. This variance serves as a confidence value indicating the stability of the RMSE values among the different subtest datasets. A high variance indicates divergent RMSE values and so the average may not be a good prediction of the RMSE value the algorithm will finally deliver. Our experiments have shown that a reasonable method of coping with this situation, is imposing a cutoff variance threshold that, when reached, automatically discards the algorithm from the selection process (for the current user). This way only algorithms which show good and little varying RMSE values across the subtest datasets will be compared and ranked. When all, or all but one, algorithm is discarded, the default algorithm is selected for the current user. The pseudocode of this best switching selection strategy can be found in the appendix (Algorithm 4). For the experiments, we used a cutoff variance threshold value of 0.2, which was empirically determined.

5.6 Weighted Hybrid Strategy

Another hybridization technique, is the *weighted hybrid* recommendation technique. With this method the scores of individual recommendation algorithms are combined together into a single hybrid recommendation score [37]. Individual algorithms can be associated with different weights to allow fine-grained control over the contribution of the individual algorithms to the final score. Weights can be set in such a way that this weighted technique produces the same results as hybrid switching (i.e., if only one algorithm contributes to the final score). The real power of the weighted technique however, lies in the ability to join multiple algorithms together and form a new hybrid algorithm. We therefore hypothesize that this technique may yield better or at least equal results as the hybrid switching technique.

An additional advantage of the weighted hybrid strategy towards other strategies like Meta-Level or Feature Augmentation is its black box approach. Individual algorithms are considered *black boxes*, which are served input data and produce output data without revealing any internal processing information. Since the final score takes only the output of the algorithms into account, new algorithms can easily be added to the system without the need for structural changes. We intend to optimize a hybrid system built on many (i.e., 10) different algorithms, and so the black box approach seems a valuable advantage.

The core challenge of the weighted hybrid technique is to find appropriate weights for the individual algorithms. We define this problem in the form of an optimization task. We adopt the notation from related work [136] where a *dynamic ensemble recommender* was defined as follows.

$$g(u,i) = \gamma_{a_1} * g_{a_1}(u,i) + \gamma_{a_2} * g_{a_2}(u,i) + ... + \gamma_{a_n} * g_{a_n}(u,i)$$

Here γ is the weighting factor for the individual algorithms a that weighs the recommendation values $g_a(u,i)$ for a user u and item i. Since our approach is a user-specific one, we want to optimize the weights of the algorithms specifically for every user so that every user may benefit from a personalized hybrid recommender system. User-specific weights can be added to the optimization task in the following way.

$$g(u,i) = \gamma_{a_1}(u) * g_{a_1}(u,i) + \gamma_{a_2}(u) * g_{a_2}(u,i) + ... + \gamma_{a_n}(u) * g_{a_n}(u,i)$$

We can now denote the objective function as

$$f(\boldsymbol{\gamma}(u))$$

where

$$\boldsymbol{\gamma}(u) = (\gamma_{a_1}(u), \gamma_{a_2}(u), ..., \gamma_{a_n}(u))$$

with n the total number of recommendation algorithms. Through an optimization process we seek to minimize the objective function (i.e., RMSE). This metric can again be measured offline by evaluating the subtest sets and averaging out the values as was done to optimize the hybrid switching strategy.

We constrain possible weight values to the interval [0,1] so that values can be easily interpreted and compared. A final recommendation score for a user u and item i, given the weight vector and individual recommendation scores, can then be calculated by means of an average weighted formula.

$$g(u,i) = \frac{\gamma_{a_1}(u) * g_{a_1}(u,i) + \gamma_{a_2}(u) * g_{a_2}(u,i) + ... + \gamma_{a_n}(u) * g_{a_n}(u,i)}{\gamma_{a_1}(u) + \gamma_{a_2}(u) + ... + \gamma_{a_n}(u)}$$

In our weighted hybrid optimization procedure, $\boldsymbol{\gamma}(u)$ will be optimized such that $g(u,i)$ minimizes RMSE on the subtest datasets. In this procedure we iteratively try to improve the individual weights of the weight vector. To reduce the number of iterations required for this optimization, we start the procedure by selecting the *best* start vector out of a number of randomly generated weight vectors. The *best* vector being the one that results in the lowest RMSE value on the subtest datasets. Together with the random vectors, we compare all the *individual weight vectors* for every single recommendation algorithm in the system. We define an individual weight vector as a vector that allows only a single algorithm to contribute to the final score e.g, $\boldsymbol{\gamma}(u) = (1,0,0,0,0,0,0,0,0,0)$. Doing so, forces the system to take also individual algorithms into account.

After the selection of the start vector, this weight vector can be improved one weight at a time. Since this optimization procedure must run for every user in the system, we want it to produce qualitative results in a very short time frame (almost real-time). For this, we have implemented a standard binary search procedure where improved weight values will be searched in iteratively diminishing intervals both upwards and downwards as Fig. 5.3 illustrates. We have experimented with other optimization procedures such as a custom genetic algorithm and publicly available optimization tools (such as SciPy[2]), but found our standard binary search procedure to produce the best or equal results in less time. To reduce the risk of ending up in locally-optimal RMSE solutions, instead of selecting only one start vector (from the randomly generated ones), multiple vectors could be optimized to find the overall best vector. We experimented with different settings and even implemented a back-tracking mechanism to allow the optimization procedure to escape locally-optimal solutions. We found that this approach did not yield significantly improved results and therefore we did not include this in our algorithm presented below. We hypothesize that the randomization of the start vector in itself already prevents ending up in the most obvious local RMSE solutions.

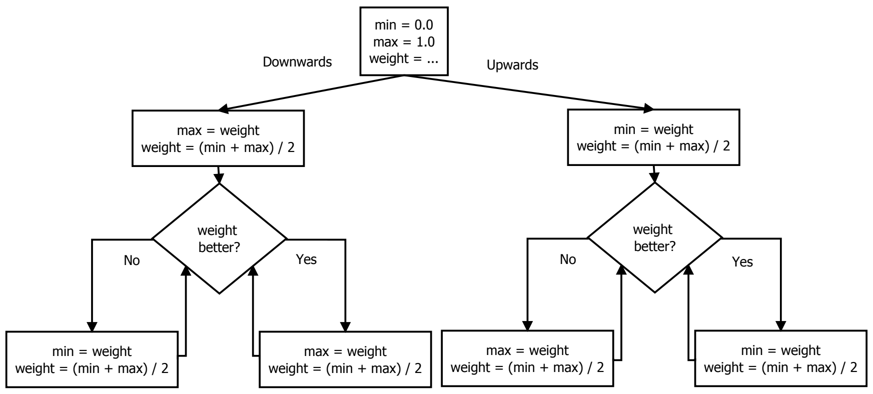


Figure 5.3: The binary search procedure to improve the weights in the weight vector. The fitness of a weight vector (i.e., is a weight better?) is defined in terms of RMSE on the subtest datasets.

We note that although the procedures searches for an improvement one weight at a time, the evaluation (i.e., is the new weight value better?) will be performed on the weight vector

[2]`http://docs.scipy.org/doc/scipy/reference/optimize.html`

as a whole. If for example, the start weight vector is $\gamma(u) = (\mathbf{0.5}, 0.1, 0.8, 0.3, 0.4, 0.4, 0, 0.6, 0.8, 0.1)$ and we seek an improvement of the first weight in upwards direction, we will evaluate if the weight vector $\gamma(u) = (\mathbf{0.75}, 0.1, 0.8, 0.3, 0.4, 0.4, 0, 0.6, 0.8, 0.1)$ yields improvement. Improvement, as stated earlier, is defined as yielding a reduced RMSE value on the subtest datasets. We again take the variance of the generated RMSE values into account to model for confidence. When a cutoff threshold for variance is reached, the evaluation discards the weight vector and the weight suggestion is considered 'not better'.

The binary search procedure is repeated for every weight in two directions until no more improvements can be found or a fixed number of iterations have passed. The pseudocode of this weighted average optimization strategy can be found in the appendix (Algorithm 5).

For the experiments, we used the same variance cutoff threshold value as for the hybrid switching strategy (i.e., 0.2). The maximum number of iterations was set to 500 and 1000 random weight vectors were used to boost the start weight vector. Adjusting the number of iterations allows to fine-tune the trade-off between optimization time and quality.

5.7 Offline Optimization Results

We evaluate the performance of the offline optimization of the two hybridization techniques as previously discussed. We first evaluate the performance (in terms of rating prediction accuracy) of the 10 individual recommendation algorithms that were selected from the MyMediaLite framework. We then continue to evaluate and compare the hybrid switching and weighted hybrid strategies. While the optimization procedures operated on the subtest datasets, all the results in this section have been calculated on the true test set.

5.7.1 Individual Algorithms

We evaluated the RMSE values of the 10 individual algorithms, averaged over all users (and 10 folds). Default parameter settings where applied as set in MyMediaLite version 3.09. Table 5.1 shows the results of the RMSE evaluation.

As the table shows, many of the individual algorithms show a similar performance. Exceptions to this are the *SigmoidSVDPlusPlus*, *Constant5* and *Constant1* algorithms, which perform considerably worse.

Closer inspection of the results of *SigmoidSVDPlusPlus*, revealed high variance in results so that for some users the algorithm performed considerably worse than for others (a result we noted also in previous experiments [53]). This result interestingly underlines the need for a user-specific approach towards recommendation. We hypothesize the *SigmoidSVDPlusPlus* algorithm needs more data to increase its relevancy or needs at least better fine-tuned parameters. For the *Constant* algorithm (which predicts a constant value of 1 or 5), poor RMSE results were to be expected.

To observe the relationships of the algorithms among each other, we plotted the RMSE values for the 7 most similar (in terms of RMSE) algorithms showing their respective 95% confidence intervals in Fig. 5.4. From the figure it now more clearly shows that *UserKNN*, *UserItemBaseline* and *SlopeOne* can be considered to have equal performance as well as *BiPolarSlopeOne*, *MatrixFactorization* and *FactorWiseMatrixFactorization*. The *BiasedMatrixFactorization* resides somewhere in between. These results are confirmed as we calculate the statistical significant differences among the algorithms with a Wilcoxon Signed-Rank Test (Table 5.2).

Method	RMSE
UserKNN	0.9458
UserItemBaseline	0.9473
SlopeOne	0.9476
BiasedMatrixFactorization	0.9576
BiPolarSlopeOne	0.9759
MatrixFactorization	0.9767
FactorWiseMatrixFactorization	0.9817
SigmoidSVDPlusPlus	1.2808
Constant5	1.7420
Constant1	2.7912

Table 5.1: The RMSE values for the individual algorithms averaged out over all users according to a 10-fold evaluation on the test set. Results are sorted from low to high RMSE.

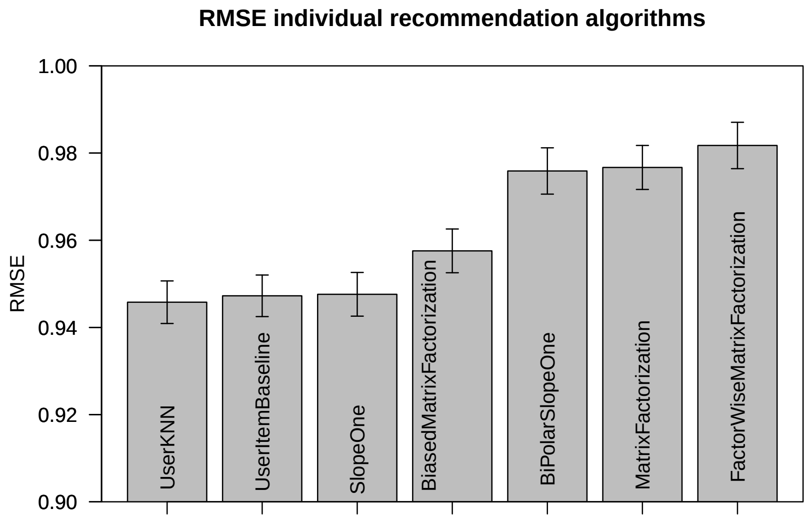


Figure 5.4: The RMSE values for the 7 most similar individual recommendation algorithms, averaged out over all users according to a 10-fold evaluation on the test set.

5.7.2 Hybrid Switching Approach

For the evaluation of the hybrid switching strategy, we performed multiple experiments, each time increasing the number of individual algorithms that were used in the hybrid recommender. The algorithms are added in order of their individual performance, from good (low RMSE) to bad (high RMSE). We define the following hybrid systems in the experiment:

$$\begin{aligned}
BS2 &:= UserKNN + UserItemBaseline\\
BS3 &:= BS2 + SlopeOne\\
BS4 &:= BS3 + BiasedMatrixFactorization\\
BS5 &:= BS4 + BiPolarSlopeOne\\
BS6 &:= BS5 + MatrixFactorization\\
BS7 &:= BS6 + FactorWiseMatrixFactorization\\
BS8 &:= BS7 + SigmoidSVD++\\
BS9 &:= BS8 + Constant5\\
BS10 &:= BS9 + Constant1
\end{aligned}$$

	1	2	3	4	5	6	7	8	9	10
1: UserKNN	-	.302	.864	**	**	**	**	**	**	**
2: UserItemB...		-	.396	**	**	**	**	**	**	**
3: SlopeOne			-	**	**	**	**	**	**	**
4: BiasedMatr...				-	**	**	**	**	**	**
5: BiPolarSl...					-	.589	.255	**	**	**
6: MatrixFact...						-	.551	**	**	**
7: FactorWise...							-	**	**	**
8: Sigmoid...								-	**	**
9: Constant5									-	**
10: Constant1				(**: $p < .05$)						-

Table 5.2: Pair-wise p-values of the null hypothesis, that the two systems have an equal performance, as computed by a Wilcoxon Signed-Rank Test.

BS stands for 'Best Switching' and the index refers to the number of individual algorithms participating in the hybrid recommender. For every one of these setups, we optimize the hybrid to select the best individual algorithm for every user individually. The results, as calculated on the test set, are shown in Fig. 5.5.

The result of the best individual algorithm (i.e., *UserKNN*) was added to the plot as a baseline for comparison. The figure demonstrates how each of the hybrid recommenders in the experiment achieves better results (i.e., lower RMSE values) in comparison with the best individual algorithm. Adding individual algorithms improves the result of the hybrid up to a number of 4 algorithms (i.e., *BS4*), results then slightly deteriorate and finally stabilize in the end.

From the results it is clear that our user-specific hybrid switching strategy performs better than simply selecting the best single algorithm and using that for every user. It is however interesting that the results do not continue to improve when additional algorithms are added. The reason for this, is that the systems *BS5*, *BS6* and *BS7* add variations of already included algorithms (i.e, *SlopeOne* and *BiasedMatrixFactorization*) rather than adding completely new algorithms as is the case for *BS2*, *BS3* and *BS4*. These variations perform considerably worse than their original algorithm, and so adding them only brings noise into the system i.e., their results on the subtest datasets do not generalize to the true test set. The performance of the last systems *BS8*, *BS9* and *BS10* remains stable because for these hybrid recommenders the performance of the added algorithms (i.e., *SigmoidSVD++*, *Constant1*, and *Constant5*) is so bad (or their variance so high) that they will almost never be selected as 'best' algorithm for a user. When we inspect, for *BS10*, the number of times each algorithm was chosen (Fig. 5.6), we find our argument confirmed.

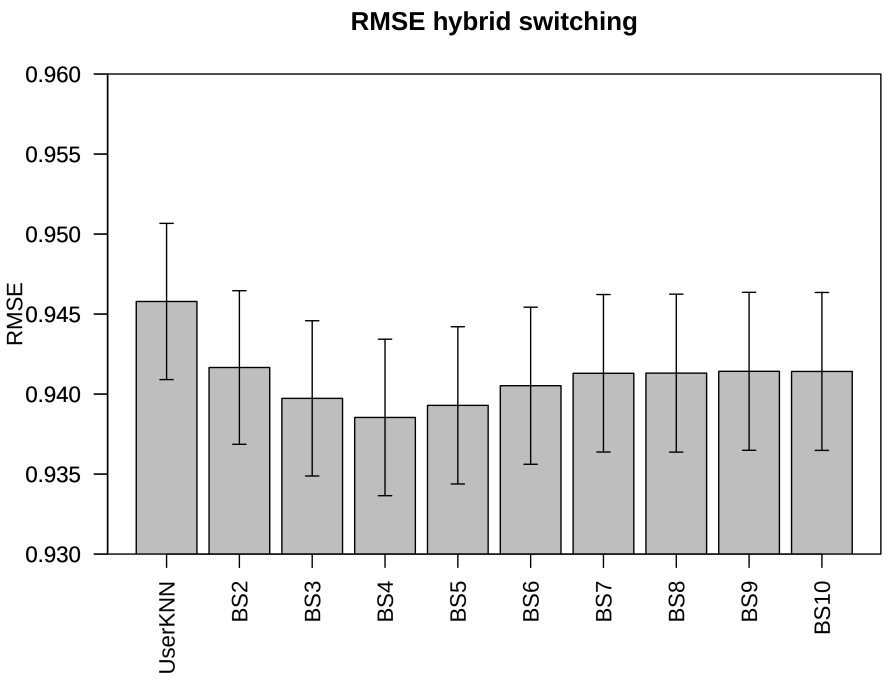


Figure 5.5: The RMSE results for multiple experiments with a hybrid switching setup. The index in the X-axis labels refers to the number of individual algorithms participating in the hybrid recommender.

When comparing the RMSE results for the hybrid switching systems, only two systems were found to have a statistical significant difference with $p < 0.05$: the *UserKNN* algorithm and the *BS4* hybrid system, although on average all hybrid systems seem better than *UserKNN*. In conclusion we state that, the best performing hybrid switching strategy (i.e., *BS4*) yields a significant improvement towards the individual algorithms approach, but the participating algorithms in the hybrid must be carefully chosen in order to obtain good results.

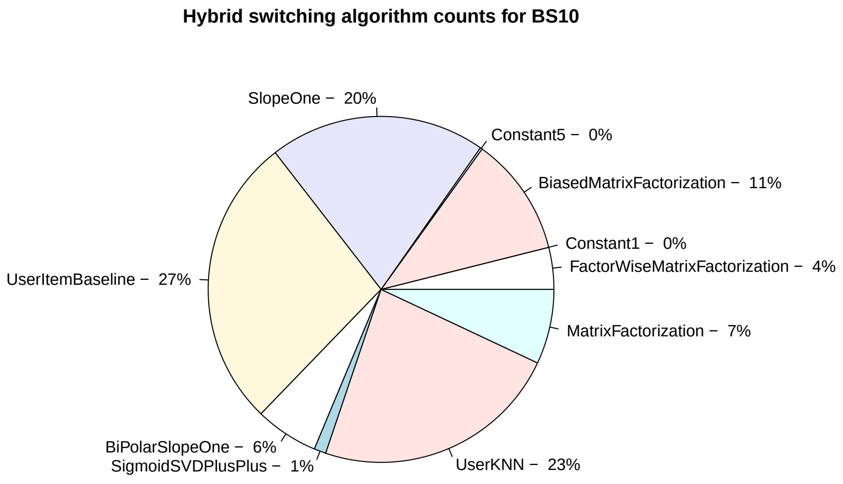


Figure 5.6: The number of times each algorithm was selected as 'best' algorithm, counted over all users for hybrid setup *BS10*.

5.7.3 Weighted Hybrid Approach

For the evaluation of the weighted hybrid strategy, experiments were iteratively performed with an increasing number of individual recommendation algorithms participating in the hybrid system. Every hybrid now optimizes the weight vectors indicating the importance of the individual algorithms on a user-specific basis. The final prediction scores for every user are compared against the true ratings in the test set.

	BS4	W2	W3	W4	W5	W6	W7	W8	W9	W10
BS4	-	.327	.781	.105	.114	**	**	**	**	.622
W2		-	.207	**	**	**	**	**	**	.140
W3			-	.177	.192	**	**	**	**	.831
W4				-	.967	.331	.150	.122	.222	.257
W5					-	.311	.138	.112	.207	.277
W6						-	.636	.562	.799	**
W7							-	.913	.829	**
W8								-	.746	**
W9									-	**
W10					(**: $p < .05$)					-

Table 5.3: Pair-wise p-values of the null hypothesis, that the two systems have an equal performance, as computed by a Wilcoxon Signed-Rank Test.

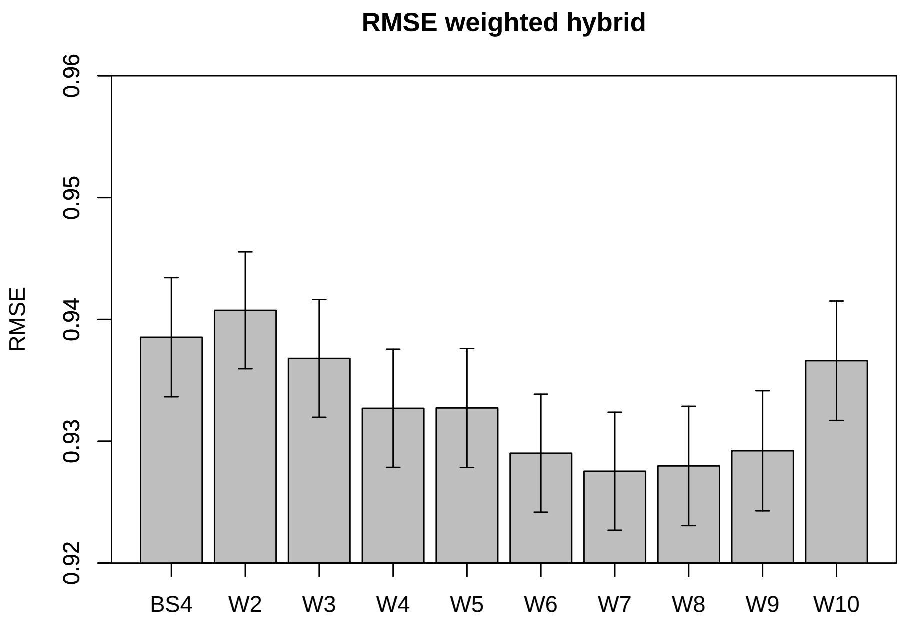


Figure 5.7: The RMSE results for multiple experiments with a weighted hybrid setup. The index in the X-axis labels refers to the number of individual algorithms participating in the hybrid.

Fig. 5.7 shows the RMSE results for multiple hybrid configurations with a varying number of participating individual algorithms. Table 5.3 depicts the statistical significance of the difference between the algorithms. Algorithms are added in the same order as detailed in the previous subsection (e.g., $W3 = W2 + \mathit{SlopeOne}$). As a comparison baseline, the best result of the hybrid switching strategy (i.e., *BS4*) was added to the plot.

The results show that most of the weighted hybrid configurations (except for *W2*) perform better than the *BS4* baseline. Moreover, the performance increases (or at least remains stable) when new algorithms are added. Exceptions are the *W9* and *W10* configurations, which show a decreased performance (increased RMSE) towards the previous configurations. This is caused by the algorithms that are added in those configurations, namely *Constant5* and *Constant1*. The *Constant* algorithm predicts always the same recommendation score (i.e., here either 1 or 5). When considering the weighted average formula used for this hybrid strategy (see Section 5.6), adding a *Constant* algorithm in the equation will function as a general weighting factor for the final prediction value.

This weighting factor can either boost or decrease the final recommendation score. When the *Constant5* algorithm is used in the calculation of the final prediction score, all predicted scores (for that user) will be slightly increased or similarly decreased when applying *Constant1*. The reason that *Constant1* has a bigger (negative) impact on performance becomes clear when we inspect the distribution of the rating values for our dataset. In previous chapters we showed how this distribution was slightly skewed towards the higher values of the rating scale (see Fig. 2.12 in Chapter 2) and therefore decreasing the final recommendation score will (on average) worsen performance more than increasing the score.

We can take a closer look at the weight vectors produced by our optimization procedure. Specifically, we are interested in how the algorithms (on average) contribute to the final prediction scores, what algorithms are used, and how many algorithms (i.e., algorithms with non-zero weights) are usually combined. We focus on the weights generated by the *W7* hybrid configuration, which showed the best performance.

Inspecting the complete set of weights produced by our optimization procedure for all users, we logged for each algorithm the number of times a non-zero weight was generated. Fig. 5.8 shows a pie chart detailing the normalized (percentage) counts for the individual recommendation algorithms.

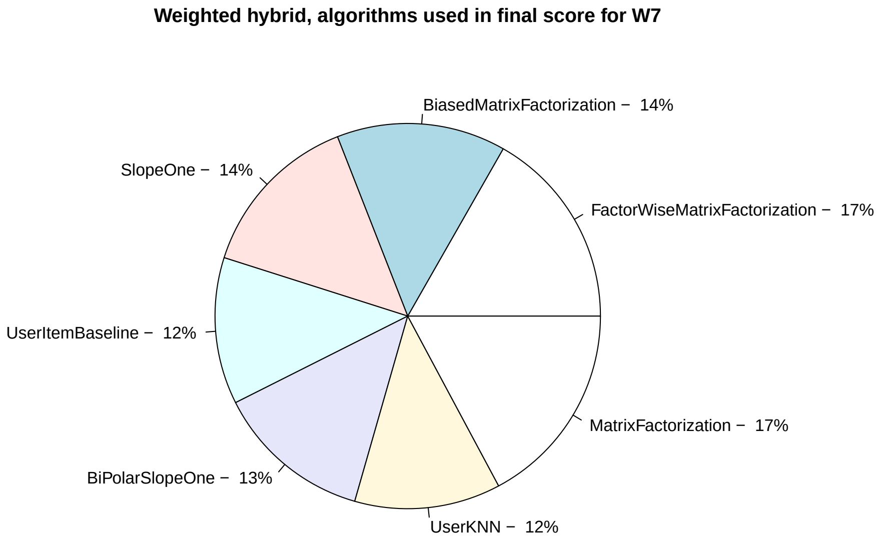


Figure 5.8: The distribution of the usage of each algorithm in the weight vectors for all users. The results are normalized in percentage.

The figure shows an approximately even distribution of the values, which indicates that the system involves every algorithm about the same number of times in the final prediction score. It seems that the weighted hybrid strategy is able to make use of all given algorithms, without degrading the performance when variations of the same algorithm are present (as was the case for the hybrid switching strategy).

Aside from how much the algorithms are used, it is also interesting to know how many algorithms usually contribute to the final prediction score for a user. Again inspecting the complete set of weight vectors produced for all users, we logged the amount of non-zero weights in each weight vector. Fig. 5.9 shows the resulting histogram for the *W7* hybrid configuration.

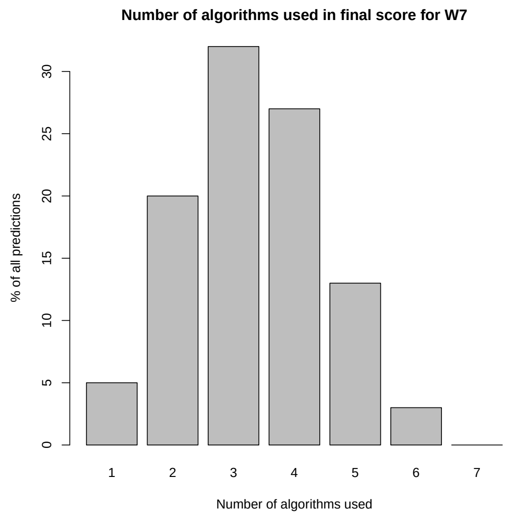


Figure 5.9: The histogram of the number of algorithms used for all users (normalized in percentage).

From the histogram we learn that the weighted hybrid strategy is usually combining multiple algorithms together. For some users, the results of as many as 6 algorithms are combined into the final recommendation score. For others, on the other hand, only a single algorithm is used. It is interesting to see that the weighted hybrid strategy does indeed revert to a hybrid switching strategy when it seems appropriate.

A final interesting aspect of the generated weight values, is the weight value itself. Table 5.4 displays for every algorithm (used in *W7*) the average weight value over all weight vectors for all users without counting the zero weights.

Algorithm	Average Weight
UserKNN	0.426
UserItemBaseline	0.421
SlopeOne	0.389
BiasedMatrixFactorization	0.282
BiPolarSlopeOne	0.257
MatrixFactorization	0.249
FactorWiseMatrixFactorization	0.201

Table 5.4: The average weight values (in range [0,1]) over all users for every algorithm used by the weighted hybrid configuration $W7$.

The order of the averaged out weights for the algorithms, matches the order of the performance results for the individual algorithms. This confirms that the results of our offline optimization strategy do in fact correlate with the final results as obtained by the test set.

5.8 Discussion

Fig. 5.10 overviews the final results of our hybrid strategy evaluation. The three result values indicate the results as obtained by the three respective recommendation strategies: best individual algorithm (same for all users), user-specific best switching, and user-specific weighted hybrid. For each of these systems the best results are shown in the graph i.e., *UserKNN* (best individual), *BS4* (best switched), and *W7* (best weighted). The differences between the results (all of which are found to be statistical significant $p < 0.05$) confirm the original hypothesis that a user-specific hybrid switching strategy will yield better results than an individual algorithm and a weighted hybrid system will outperform even the hybrid switching strategy.

5.9 Conclusion

We started by noting the existence of vast numbers of recommendation algorithms available to tackle the information overload problem. Combining multiple algorithms together, seemed a sensible approach to

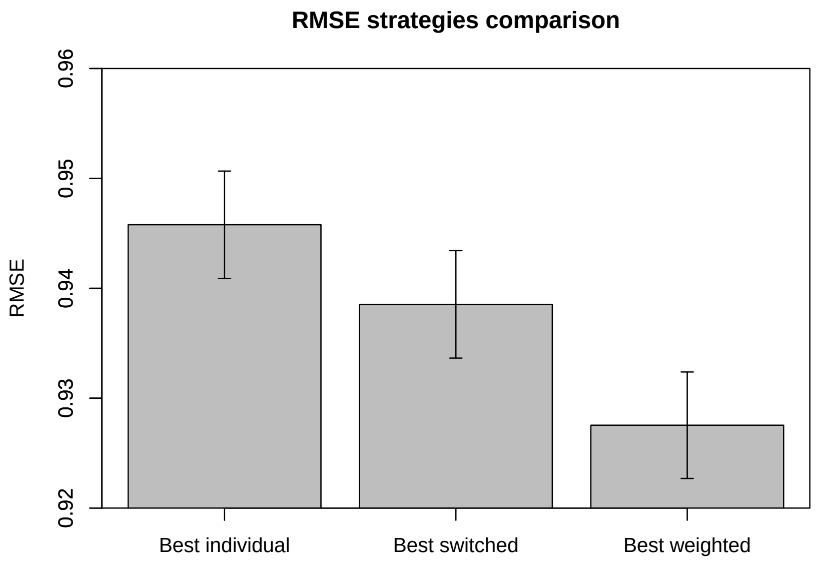


Figure 5.10: A comparison of the best RMSE results obtained by the three systems compared in our evaluation: individual algorithms, a switching approach and a weighted hybrid approach. All of these differences were found to be statistical significant.

harvest the union of their merits. However, combining algorithms into hybrid recommender systems can be cumbersome; often manual configuration is required such that the recommender can not be easily re-used for other scenarios.

We considered the configuration of a hybrid system as an optimization problem which generated hybrid recommender systems that are automatically fine-tuned towards individual users. Focus was on the commonly used *switching* and *weighted* hybridization techniques. We demonstrated an approach that allowed the hybrid recommender system to optimize recommendations offline.

Results showed that the *switching* strategy was highly sensitive to the used individual algorithms i.e., best results when most different algorithms were used. The *weighted* strategy, on the other hand, was more robust and, even with the simple binary search optimization procedure, it obtained significantly better results by blending the individual algorithms into user-specific ensembles.

Although the evaluation of our methods focused on the popular accuracy metric RMSE, other offline calculable metrics can be optimized for. The evaluation intended to show the success of the optimization procedure, and the specific advantages of the weighted hybrid procedure over a hybrid switching approach. Implementing the weighted average strategy in a hybrid recommender allows for easy adding new algorithms or variations of existing algorithms without fundamentally disrupting user experience. The quality of offline optimization depends however on the quality of the available data and will therefore always be fundamentally limited in potential in comparison to online and user interactive approaches. This effect is known in research as *the magic barrier*.

Chapter 6

Online Optimization of Personalized Hybrid Recommender Systems

6.1 Introduction

In the previous chapter we pursued the ideal hybrid recommender which was capable of integrating all known recommendation algorithms and auto-adapting its hybrid configuration to dynamically generate optimal recommendations for individual users. Now, we build on these results and try to get the recommender out of the lab by assessing and improving its ability towards meeting real-world requirements for an online recommendation scenario.

With the term *online* in this chapter, we refer to both the interpretation as an Internet-connected system which can be interacted with by online users, and the idea of an *online algorithm* which is an algorithm that solves problems in real-time based on limited data input. A typical example of an online algorithm is the *first fit* algorithm found in the bin-packing world [140]. Given a set of trucks to load packages on, the algorithm decides for every package what its optimal place would be to minimize the total number of used trucks. An online variant of such an algorithm decides what truck to use based only on the knowledge of any previous packages. While more optimal algorithmic solutions take into account the complete set of all packages, such data is not always available in online systems. The same situation applies to the recommender systems domain. In previous chapters we always started from a rating dataset, but what if we start without ratings, and for each added rating we want to provide an updated recommendation list? Such an online recommendation scenario will be the focus of this chapter, both from the Internet-connected perspective as from the online algorithm perspective.

The most important requirement specific for an online recommender system is **scalability**. For online systems it is extremely difficult to predict the number of active users since online popularity is very variable. An online system may serve a number of users ranging from just a few hundreds to many thousands and even millions. More importantly, the number of users may change very quickly in short periods of time because of online viral effects. Therefore an online system should be able to dynamically scale with the workload it is presented with.

Another requirement for online systems is **responsiveness**. Nowadays users have grown accustomed to fast and responsive online services. Whether they are searching on Google, posting updates to Facebook or watching videos on YouTube, they expect an instant response from the system they interact with. Responsiveness in terms of a recommender system scenario would mean that user interactions have immediate visible effects. For example, a user that rates a recommended movie as *bad* does not want to see that movie in its recommendation list anymore (even though the system may only calculate new recommendations once a day).

While recommender systems in the past often acted as *black boxes* where ratings go in and recommendations come out [81], users nowadays expect some kind of explanation about the origin of the recommendations. Online platforms like IMDb or Amazon[1] display their recommendations with accompanying titles as 'People who liked this also liked...', 'Frequently Bought Together' or 'Customers Who Bought This Item Also Bought'. Despite their simplicity, the titles succeed in explaining to the users how the recommendations are calculated. Even though the explanation may be an oversimplification of the true recommendation calculation process, it may still serve to inspire user trust and loyalty [141]. Therefore it is important for an online (recommender) system to be (or at least *seem*) **transparent** to the user.

Finally, users should have some form of **control**. It has been shown that dynamic user interaction with a recommender system increases user satisfaction and may even boost the relevance of predicted con-

[1]`http://www.amazon.com`

tent [79, 80, 142]. Our online recommender system should therefore interactively offer some way for users to be in control of their resulting recommendations or at least have some way of influencing and guiding the system other than by merely providing ratings. In conclusion, the discussed real-world challenges for an online recommender system can be summarized in the following list of requirements (*REQs*).

- **REQ1** Responsiveness
- **REQ2** Scalability
- **REQ3** System transparency
- **REQ4** User in control

We refer to these as **REQ1 [Responsiveness]**, **REQ2 [Scalability]**, **REQ3 [Transparency]** and **REQ4 [Control]**. In this chapter, we evaluate and improve the ability of our hybrid recommendation strategy to meet these requirements in an online recommendation scenario. But first we discuss some related work that specifically focuses on the above online requirements.

Research Questions

- How can hybrid optimization be applied in online environments?
- How can system and user requirements be aligned?

6.2 Related Work

Responsiveness for recommender systems translates to being able to react in (almost) real-time to the arrival of new ratings in the system. Most recommendation algorithms need to retrain their complete model to integrate new data which can rarely be done in real-time. For some specific recommendation algorithms, online updating approaches have been developed such as SVD [143] or MatrixFactorization [144]. In [145], the *StreamRec* recommender system was demonstrated which allowed instant recommendation updates using an underlying item-based collaborative filtering approach. Since the online updating approaches are usually algorithm-specific, few research actually focuses on real-time updating hybrid models.

For related work regarding scalability, we refer to our chapter focusing on high-performance recommending (Section 4.2.1).

Providing system transparency and user control in a recommender system should prevent users from feeling trapped inside a *filter bubble*[2] of tailored information. Explanations have been known to positively increase the user perceived system transparency [146]. User control in a system is however difficult to achieve. Aside from processing ratings, recommendation algorithms usually do not provide the tools for users to allow fine-grained preference feedback. In [142] *meta-recommendation systems* were introduced. The authors experimented with a hybrid system called *MetaLens* that allowed users control over their recommendations by means of a preference screen where a number of item features could be filtered on. Their user-study confirmed that users preferred the advanced level of control offered by their system. Another interactive recommender system is the TasteWeights system by Bostandjiev et al. [79]. As we already discussed in Section 3.3.1, their user study also indicated that explanations and interaction with a visual representation of the hybrid system increases the user satisfaction and recommendation relevance. The same results were found by Gretarsson et al. [80].

While separately each of our online requirements has been tackled in related research, to our knowledge no work exists that takes all four requirements into account at the same time. We focus on realistically deploying our hybrid optimization approach to an online environment in a scalable and user friendly way.

6.3 Online Optimization for Hybrid Recommenders

Fig. 6.1 illustrates the optimization process for one user of the hybrid recommender based on the results of the previous chapter. The process starts with the concept of a rating dataset. We assume a user has already expressed an opinion about a number of items present in the system. In a first step, we use the rating dataset to create multiple fold datasets which are then split according to some pre-set ratio into training and test fold datasets. Fig. 6.1 illustrates the situation with 3 fold datasets. Each training subset of the fold datasets is provided as input to a number of recommendation algorithms (2 algorithms a_1 and a_2 depicted in the figure as a black square and triangle shape). At the same time the complete rating dataset is also provided as input to instances of the same

[2]`http://www.thefilterbubble.com`

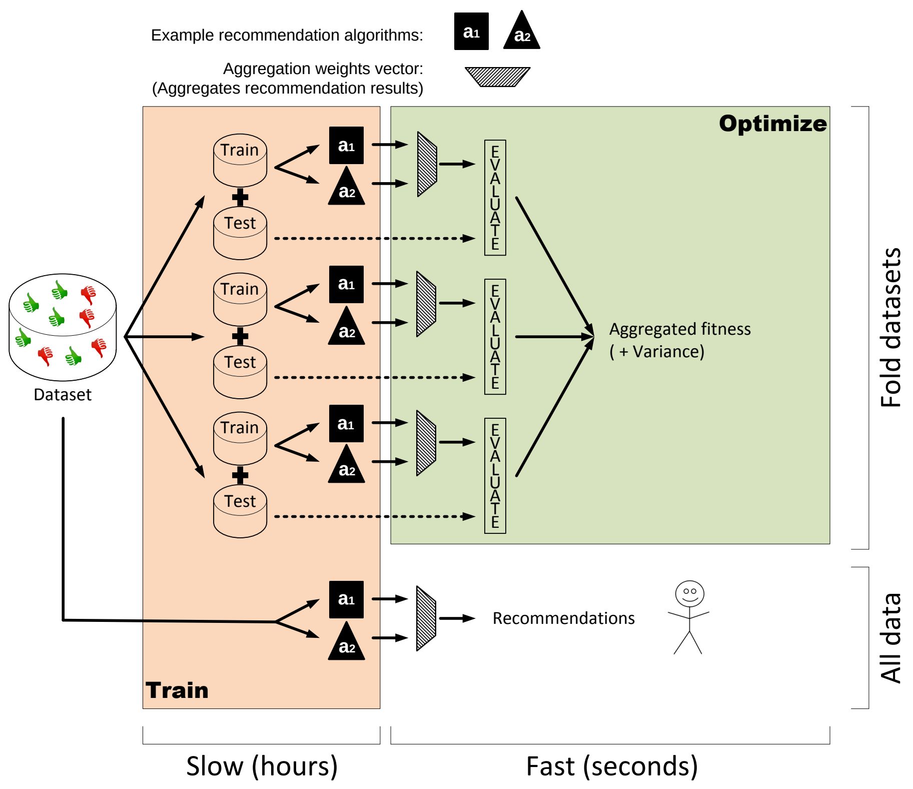


Figure 6.1: The optimization process for the hybrid recommender illustrated for one user, using 3 folds and 2 (individual) recommendation algorithms.

algorithms. Each algorithm then, in parallel, trains its models based on the given input. In the figure, 2 algorithms are defined and so 4 instances of those algorithms (3 for the fold datasets and 1 for the complete rating dataset) will be trained, which results in the computation of 8 models. This computational step can be potentially very slow depending on which recommendation algorithms are involved. Algorithms like MatrixFactorization are generally accepted to train fast [147], while other algorithms like KNN methods can be very slow [148] (depending on the parameters e.g., neighborhood size). Although this computation phase will be very slow, it needs to be run only once in order for the system to be able to present a user with recommendations. After this initial computation the system will be able to incorporate new rating data and react to user responses in real-time as we will show later in this section.

When the training of the algorithm models has finished, the system uses the output i.e., recommendations to optimize a weight vector used for the configuration of the final hybrid recommendation list. We integrate a *weighted* hybrid approach (which performed best in the previous section) and therefore such a weight vector is needed for the aggregation of the individual recommendation list. This aggregation is represented as a vertical trapezoid shape in the figure that takes multiple recommendation results as input and outputs one hybrid recommendation list. In a first optimization step, the outputs of the recommendation algorithms trained on the fold datasets are aggregated using an initial start weight vector (identical for all folds). Since the fold datasets were split in training and test sets (and models were only trained on the training sets), the remaining test sets can be used to evaluate the quality of the aggregated result. We do not specify an exact method of evaluation as this will depend on the end goal of the recommender (e.g., user satisfaction, recommendation accuracy, item coverage, etc.). The output of the evaluation must however be quantifiable into a numeric value so that it can be compared and measured. Three evaluation values result from the scenario as depicted in the figure, one for each fold. The evaluation scores are aggregated, by some chosen aggregation function e.g., *arithmetic mean*, into a single *fitness* value indicating the quality of the current weight vector. The variance of the individual evaluation scores between the folds must also be taken into account to indicate the consistency of the performance of the weight vector over the different dataset folds.

The weight vector is then step-wise optimized by applying standard optimization procedures borrowed from the machine learning domain until a certain number of iterations has passed or a sufficient fitness value has been reached. In the previous chapter we illustrated the offline optimization procedure using binary search and *RMSE* as evaluation criteria. The optimization method can be any chosen method as long as it has a fast convergence rate to the optimal value. The training of the algorithms will not be run often, so it does not matter if it takes a long time (i.e., hours) to complete. This step of optimizing the weight vector however will be executed frequently and therefore should be as fast as possible (i.e., complete in a matter of seconds). When the weight vector has been optimized, it can be used to generate the final recommendations by applying it to aggregate the algorithm models which were trained on the complete rating dataset (at the bottom of the figure).

When applying optimization methods, part of the data is often dedicated for the evaluation of the objective function. Because of our proposed procedure of training and testing on fold datasets, all of the rating data can still be integrated in the models of the final (non-fold) recommendation algorithms. Furthermore because of its high-level specification, the hybrid optimization procedure can be applied to different methods of recommendation strategies (e.g., rating prediction versus item prediction).

6.3.1 Avoiding Overfitting

One potential problem of the optimization procedure as described above is *overfitting* [131]. For users with a low number of ratings the system is prone to overfit. Such users could be handled in two ways: we could require more ratings from the user before calculating the recommendations, or train the models on the few ratings available and as weights use a default pre-computed weight vector that has shown to yield good results for many other users of the system (i.e., non-personalized approach).

To prevent overfitting for users with a large number of ratings, it is better to not use all data for the optimization but instead only a random subsample of the dataset. Creating (and optimizing for) multiple randomly subsampled datasets at the same time is even better, and so this is implemented in our approach. By optimizing for multiple folds at the same time and taking into account the agreement of the evaluation of the models (i.e., the variance) the optimization process is forced to generalize over the complete rating dataset as well as over random subsampled subsets. If the provided ratings are good indicators of possible future ratings, then the system should be able to generalize well. By changing the number of fold datasets and the training-test split ratio the process can be fine-tuned for the specific needs and properties of every use case.

Using very few folds, say in the extreme case only 1, requires very few computational effort but dramatically increases the overfitting risk. The opposite extreme case where the number of folds (and training-test ratio) is so high that every data point (i.e., rating) in a certain fold serves as the test set while all other ratings make out the rating dataset is referred to in literature as *Leave-one-out* cross-validation [149]. While this method is very thorough in using all data, it is computationally very expensive. A well-accepted meet-in-the-middle approach in recommender systems literature is the k-fold cross validation method where k folds are generated and used for testing, k often set to 10 for robustness [150, 151]. It improves the chance of generalizability (reduces overfitting) with only limited additional computational burden. Ultimately also the specifics of the involved scenario will be important factors e.g., the size of the dataset, number of samples/ratings per user, etc. In Fig. 6.1, 3-fold cross validation is illustrated.

6.4 A Responsive Online Recommender

In the introduction we defined our requirements for an online recommender system, one of which was **REQ1 [Responsiveness]**. For the proposed hybrid optimization process both slow and fast components were discussed i.e., the training of the models versus the optimizing of the weight vectors. By combining both components, the system can be made responsive, or at least *appear* responsive to the user.

As most recommender systems, the proposed system in this work suffers from the cold-start syndrome [152]. Without any data, no models can be trained, no weights optimized and therefore no recommendations can be generated. Two common ways of dealing with the cold-start problem is either by presenting the user with a list of default non-personalized recommendations (e.g., most popular items), or not presenting any recommendations at all and requiring (more) data from the user before presenting any results.

As soon as data (i.e., user ratings) are available, the models can be trained. Since the optimization process requires the output of trained models, the initial training step must be completed first. While the initial training of the models may be slow, only the first time it will block the recommendation process i.e. when the models are trained on a new user for the first time. With the models trained, the optimization step is designed to complete almost instantly, which can be leveraged to making the system feel responsive.

For a system to feel responsive it must react to user input in almost real-time. If we were to add new ratings to the training sets and require the models to be recomputed, the system would be too slow. Instead we propose to add new ratings to the test fold datasets. As shown in Fig. 6.2, by adding new ratings directly to the test datasets, they affect the optimization of the weight vector which in turn influences the final recommendation list. So by adding newly provided ratings to the test fold datasets and instantly re-optimizing the user's weight vector, the new rating can trigger changes in the final recommendation list.

Every now and then the individual models can be retrained offline

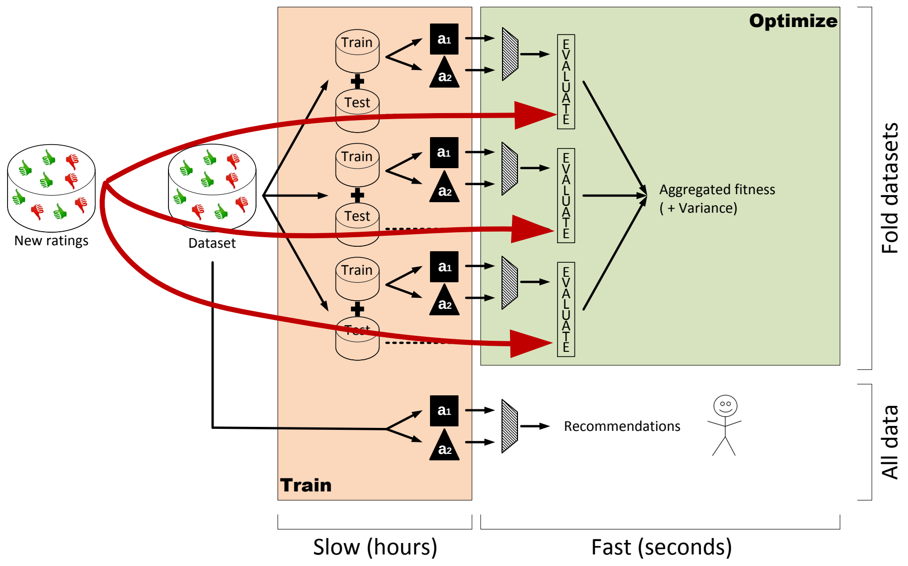


Figure 6.2: The optimization process, detailing how new ratings are added to the test fold datasets where they can have an instant affect on the final recommendations without the need for retraining the individual models.

(incorporating the new ratings since last training) and then be inserted back into the online system, all of this hidden from the user. That way, the system is capable of calculating powerful and complex models and at the same time respond in real-time to provided user feedback (which was requirement **REQ1 [Responsiveness]**).

6.5 Server-clients Structure

An important requirement for online recommenders is **REQ2 [Scalability]**. For online systems it is very hard to predict a realistic number of engaged users. There might be thousands of users or even millions depending on the popularity. Therefore, for online recommenders, scalability will be even more important than for closed environment (offline) recommenders. For a recommender to be scalable, its underlying model must be scalable i.e., able to handle a growing number of users or data without exponentially taking more time to calculate. When considering our hybrid model which integrates multiple fold datasets and various individual recommendation algorithms, it may seem like some compromise to scalability will have to be made. We will show however that by adopting a client-server architectural design, our hybrid system parallelizes extremely well, which allows it to scale naturally to available hardware and large user bases.

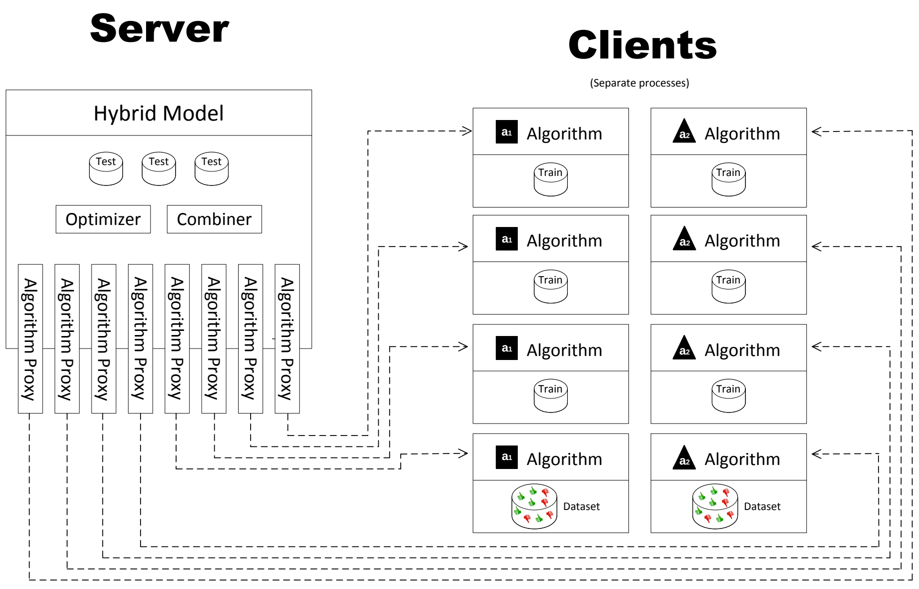


Figure 6.3: The client-server architectural design, illustrating how each individual algorithm is executed in a separate process (potentially on a different machine) and communicates with the *Hybrid Model* by means of proxy objects.

Fig. 6.3 illustrates the architectural design applied to the scenario from Fig. 6.1. There are 3 fold datasets and 2 individual recommendation algorithms (the symbolical black square and triangle). For each of the training fold datasets, instances of both algorithms are trained in addition to the instances trained on the complete rating dataset, bringing the total number of algorithm instances for this scenario to *8*. The main principle of the client-server approach is to isolate parts of the system that can run in parallel into their own separate processes. The figure shows the main server process i.e., the *Hybrid Model*, which stores the test fold datasets, optimizes weight vectors (*Optimizer* component), combines the final recommendations (*Combiner* component), and communicates with the instances of the individual algorithms.

Instead of running the client instances in the same process as the server (and thus limiting their ability to parallelize), they are executed in separate processes and communication is handled by *Algorithm Proxy* components. Communication by means of proxy components allows the *Hybrid*

Model to interact with the algorithm instances independent of their true location, which may be on the same computing node, another node in the local network, or a random computer across the Internet.

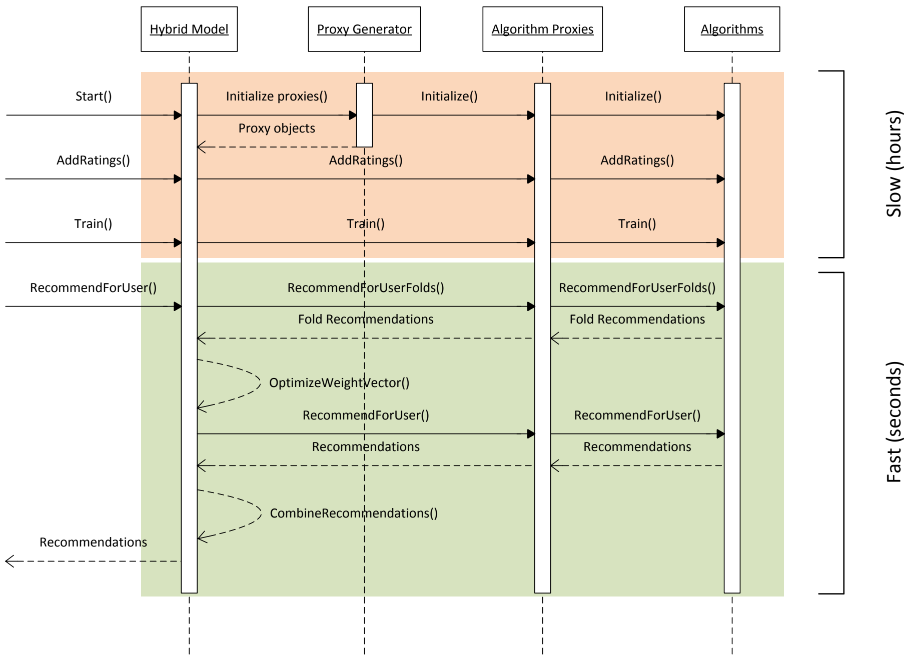


Figure 6.4: Sequence diagram illustrating the execution flow of the complete recommendation process from initialization to returning the recommendation results.

The main control flow of the recommendation process is depicted in Fig. 6.4. When the system is first started, the proxies are initialized by the *Proxy Generator* component. This component initializes the processes of the individual recommendation algorithms across available computing nodes. If multiple computing nodes are available, the component attempts to distribute the processes over the nodes as equally as possible. A link to the proxy objects is provided back to the *Hybrid Model* to allow future communication. When the model is initialized, ratings can be added, which are processed in training and test datasets and passed to the appropriate algorithm proxies (training fold datasets to the fold algorithms, full rating dataset to the non-fold algorithms).

When the *Train()* command is triggered, the command is delegated to all the algorithm proxies in parallel, which causes all of them to start training at the same time in their own separate processes. Since the total time will be equal to the maximum execution time over all trained algorithms, this phase may take long (i.e., hours) to complete. When all algorithms have completed training, the *Hybrid Model* is notified and may start accepting recommendation requests for specific users.

The request for recommendations for a specific user triggers a chain of events eventually leading to the final recommendations. First the weight vector for that user must be calculated (if not already available) by the optimization procedure in the *Hybrid Model*. The optimization requires the test fold datasets (which are available in the *Hybrid Model*) and the recommendations for the algorithms trained on the training fold datasets. With the weight vector available, all that remains is to apply it in the final phase which is the combination of the results of the individual recommendation algorithms trained on the complete rating dataset.

The main advantage of our client-server architecture is the deployment flexibility. Because the principal calculating components are decoupled and running in their own separate processes controlled by one server (i.e., the *Hybrid Model*), they can be distributed across computing nodes as desired. This allows to take into account the specific properties of the individual algorithms. Algorithms that require a lot of RAM memory may be deployed on dedicated machines, while disk-intensive algorithms may be deployed on machines with specially equipped hard drives. Furthermore, all of the algorithms are executed in parallel which reduces the main scalability of the system to the scalability of the least scalable integrated individual recommendation algorithm. The only computations affected by the number of integrated individual recommendation algorithms are the optimizing and combining processes in the *Hybrid Model*. We will show however that the impact of these effects on the general scalability of the system is limited.

6.5.1 Performance Optimization: Prefetching

Implementing the above described approach requires some optimization to avoid that bottlenecks as network speed may compromise the performance of the system. To illustrate the effect of network speed on overall performance, consider the pseudocode algorithm for weight vector evaluation as shown in the appendix (Algorithm 6). Assume we are in a rating prediction scenario and are evaluating the quality of a weight vector using the popular *RMSE* metric as objective function.

The code fragment shows two procedures which are needed for the evaluation of a given weight vector *weights_vector*. Here the *RMSE* value serves as fitness value allowing to compare (and therefore optimize) the quality of different weight vectors. *RMSE* is calculated by comparing all the ratings of the given user in the test fold dataset with the predicted score of the algorithms. The predicted score is calculated using a simple weighted average formula to aggregate the individual prediction scores of the recommendation algorithms.

Although the naive code fragment functions correctly, it will not be very efficient considering our client-server architecture. The reason for this, is line 18:

$$prediction \leftarrow \mathbf{algorithm}.get\_recommendation(user, item)$$

While the weight vector evaluation and *predict* functions will run in the server process of the *Hybrid Model* (Optimizer component), the above line of code requests the recommendation value for a certain user and item from an algorithm proxy, triggering the request to be passed to the actual process of the recommendation algorithm which may be running on another computer. So every time the request is made, in the background a network connection may be set up and torn down for the required communication between the algorithm proxies and the actual algorithm processes. Such a request is individually considerably fast, but in the pseudocode fragment the request would be called for every rating in the test fold dataset and for every algorithm proxy, which could limit the performance of the optimization method in the *Hybrid Model*.

We implemented the proposed approach in Python using the *XML-RPC*[3] package for the communication between the algorithm proxies and the actual algorithm processes. The *XML-RPC* package wraps every request as an XML document that is transported over HTTP. While the overhead of one request is small, the accumulated overhead of many of such requests greatly influenced the end performance of our system. Since the performance of the optimizing part of the *Hybrid Model* should be very high to meet our **REQ1 [Responsiveness]** requirement, a prefetching strategy was devised.

Instead of requesting the prediction values at the moment they are needed in the calculation, it proved better to request them all at once before the start of the calculations. Many small data requests can as such

[3]`http://docs.python.org/2/library/xmlrpclib.html`

be bundled in one single network request, which dramatically reduces network connection overhead. The recommendation value is needed for each rated item and for each algorithm; so if our system integrates 3 algorithms and the current user (of which the weight vector is being optimized) has rated 100 items, then 300 single data requests for prediction values would be transferred over the network. With the prefetching approach, only 3 requests are made (one for every algorithm). Implementing a prefetching approach was necessary to guarantee the performance of both the *Optimizer* and *Combiner* components of the *Hybrid model*.

6.5.2 Limitations

While the proposed model aims for flexibility and performance, its complexity imposes heavy constraints on underlying hardware configurations. For the client-server architecture to be truly effective, every process should be able to run on a dedicated processor core. Since data is replicated in multiple folds and over multiple instances, the available RAM memory of the system will also be a limiting factor. Although our proposed approach in theory can be deployed on any hardware configuration, an optimal configuration would be a cluster of computing nodes with a total number of dedicated processors of at least the number of spawned processes, linked together with a high-speed network connection. Every algorithm will be instantiated ($\#folds + 1$) times: once to train on all rating data and once per fold dataset. Each algorithm runs in its own process and so the total number of required processor cores can be determined by Formula 6.1. Note that the *Hybrid Model* itself also runs in a separate process (hence the additional $+1$).

$$required\ processor\ cores = 1 + ((\#folds + 1) \times \#algorithms) \quad (6.1)$$

6.5.3 Online User Interface

Previous sections addressed the online requirements of **REQ1 [Responsiveness]** and **REQ2 [Scalability]**, which were both focused on the system side of the recommender. The remaining requirements are the transparency of the system (**REQ3 [Transparency]**) and user control (**REQ4 [Control]**), both of which directly affect the user side of the recommender and thus need to be integrated in the interaction process between user and system i.e., the user interface (UI). In this section we discuss some UI considerations for our online hybrid optimized recommender system.

As mentioned earlier in Chapter 3 (Section 3.3.1), adding explanations to recommendation results is an easy way to increase the transparency of a recommender system and by extension the user satisfaction. The origin of for example collaborative filtering results may be easily captured by phrases as ‘recommended because similar users have liked this item’. For recommender systems based on hybrid principles however, it will be more difficult to explain in a concise sentence why items are being recommended.

Also in Chapter 3, we presented a visualization framework for hybrid recommendation results. This framework introduced explanations for hybrid recommendations by presenting users with the calculation formula and resulting recommendation value (see Fig. 3.16). While this provided optimal transparency of the recommendation process, such a visualization is only appropriate for advanced or administrative users.

The framework does however have an interesting hybrid configuration interface that can be re-used for both the purpose of system transparency and user control. Our online recommender system is based on a weighted hybrid strategy which by its very nature offers components for user control: the weight vectors. The weights in the weight vector, model the contribution of each individual recommendation algorithm to the final hybrid recommendation output and thus can be used as proxies for the *importance* of the algorithm for a specific user. By allowing users not only to inspect their weight vector but also to modify the individual weights manually, users can directly influence and fine-tune their recommendation lists to their specific (and maybe contextual) interests. As previously illustrated in Fig. 3.15, weight vectors can be visually inspected and manipulated in a very intuitive interaction process i.e., by sliders. An accompanying pie chart could help to visualize the contribution of every algorithm to the final recommendation score (if all algorithm weights are in the interval $[0, 1]$).

In the previous sections we detailed how the hybrid system can automatically optimize the weight vector for a specific user. This optimization will however be based on some measurable evaluation metric e.g., *RMSE* which might not correspond to the users’ expectations (maybe users prefer *serendipity* instead of *recommendation accuracy*). By allowing to tweak the weight vector manually and thereby overriding the automatically determined weight vector, users are able to fine-tune their recommendations to their own specific expectations. Note that in Section 6.3 we explained how the process of calculating the weight vector and combining the final recommendations could (and should) be computed very fast. Therefore when a user overrides the weight vector, new hybrid recommendations can be generated instantly which provides the user-system interaction process a very natural feel.

6.5.4 System Experts Versus Normal Users

While inspecting and manipulating the weight vector for the individual algorithms is indeed a way of introducing control and transparency to the system, the above approach would still fail for normal non-technical users. For system experts or researchers who are evaluating the recommender system, direct control over the weight vector will be very interesting, but for ordinary users who are oblivious to the technicalities of the recommender system, manually adapting the weights may be a too technical task.

What is possible however, is to simplify the algorithms to the users e.g., instead of saying 'content-based recommender' we could say 'movies similar to the ones you liked'. By translating algorithms to their most defining feature, the effect of changing the weights could be made understandable for normal users. In Fig. 6.5 this scenario is illustrated for three recommendation algorithms. *Novelty* could refer to an algorithm focusing on (i.e., predicting more) novel movies and the same for *Popularity* and *Similarity*.

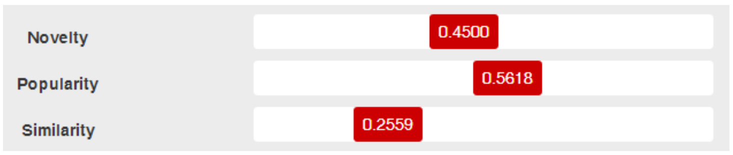

Novelty 0.4500
Popularity 0.5618
Similarity 0.2559

Figure 6.5: Illustration of how algorithms can be translated to their most defining features to make changing the weights more interpretable for non-technical users.

Recommendation algorithms that are not easily translated to an understandable concept for normal users e.g., MatrixFactorization could simply be referred to as 'Best system guess' or 'Determined by magic' as used sometimes in Google services (e.g., Fig. 6.6, which was a sort option in the former Google Reader platform).

As an extension, the system could offer various post-recommendation

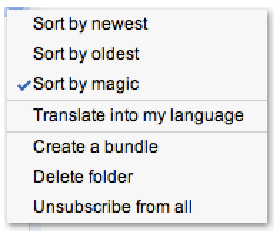


Figure 6.6: Popup on the former Google Reader platform providing the option to *Sort by magic*.

filters like genre filters (often demanded by users for movie recommendation scenarios [89]) which can be easily implemented in the user interface without any modifications to the underlying recommender system. Ultimately, the combination of both using filters and manipulating the weight vector through the user interface provides users with the necessary tools to interactively tailor their recommendations to their own interest in a transparent way.

6.6 Online Optimization Results

We want to evaluate our system in four dimensions focusing on our self-defined online requirements of responsiveness, scalability, transparency and control. The first two requirements can be evaluated experimentally by deploying the system on actual hardware and measuring the resulting execution and response times. We present the results of such experiments in the following sections.

To properly evaluate user-related metrics as control and transparency however, a typical approach is to perform an online experiment involving actual users, which allows the analysis of user behavioral patterns and user satisfaction indicators. For this purpose we created a Google Chrome extension – called MovieBrain – that deployed our proposed hybrid optimization system as a dynamic and interactive movie recommender system that integrated with the IMDb website. Details about this online evaluation and its results can be found in the next chapter.

6.6.1 Scalability

The concept of scalability can focus on two scenarios: *strong scaling* or *weak scaling* [97]. In a *strong scaling* scenario, the amount of work stays constant while the number of *workers* (e.g., computing nodes, processor cores, etc.) varies. The term *weak scaling* refers to the opposite scenario where the number of workers is constant while the amount of work changes. So when a system is referred to as 'scalable' it could mean two things. Either the system is capable of scaling across multiple computing nodes thereby reducing the total execution time through parallel computing (i.e., strong scaling), or the system is capable of processing increasingly bigger workloads without exponentially increasing the execution time (i.e., weak scaling). Either scenario is interesting for our online system and so in this section we investigate both.

The following experiments were run on the High Performance Computing (HPC) infrastructure available for researchers at our university[4]. The computing nodes deployed in the experiments have the following specifications.

- *CPU*: dual-socket quad-core Intel Xeon L5420 (Intel Core microarchitecture, 2.5 GHz, 6 MB L2 cache per quad-core chip), thus 8 cores / node
- *memory*: 16 GB RAM (DDR2 FB-DIMM PC-5300 CL5)

Computing nodes are interconnected by an Infiniband (i.e., high-speed) network and each dispose of a local hard disk (private storage) and have access to shared storage (GPFS) as well.

In the experiments, the complete recommendation process (from initialization to the generation of the final recommendations) is deployed in various experimental configurations. We used our MovieTweetings dataset as simulation data for these configurations. The $200K$ snapshot was used, which includes 200,000 ratings by 25,011 users for 14,732 movies. A split ratio of 6:4 was set for the training-test fold datasets.

Our hybrid optimized approach integrated individual recommendation algorithms as black boxes i.e., only the input and output of the algorithms are taken into account by the system without knowledge of the internal recommendation calculation process. Because of this approach there are no restrictions towards the type of recommendation algorithms that can be integrated. To illustrate this behavior, in the following experiments we use (rating prediction) recommendation algorithms from the MyMediaLite recommendation algorithms software library. As evaluation function for the optimization process (see Section 6.3), *RMSE*

[4]`http://www.ugent.be/hpc/en`

was implemented. The *StochasticHillClimber* method (parameter *MaxEvaluations*=1000) from PyBrain[5], a modular machine learning library for Python, was integrated as optimization function.

6.6.1.1 Strong Scalability

To investigate the strong scaling ability of our system, we experiment with deploying the system on a varying number of computing nodes while keeping the workload constant. The experimental setup is defined in the following list.

- *Dataset*: 200K MovieTweetings snapshot
- *Algorithms*: *MatrixFactorization*, *SlopeOne*, *LatentFeatureLogLinearModel*
- *Computing nodes*: 1, 2, 3, 4, 5 (8 cores per node)
- *Fold datasets*: 2, 4

The 3 MyMediaLite algorithms were selected based on their divergent properties regarding complexity, execution time and RAM consumption as detailed by Table 6.1. Default initialization parameters were used as set in MyMediaLite version 3.10.

	Complexity	Time	RAM
MatrixFactorization	complex	fast	low
SlopeOne	simple	fast	low
LatentFeatureLogLinearModel	complex	slow	high

Table 6.1: The divergent properties regarding complexity, execution time and RAM consumption for 3 rating prediction algorithms from the MyMediaLite recommendation library.

The experiment begins with the startup of the system: computing nodes are initialized and algorithm proxies are constructed. When the system is ready to accept ratings, the rating dataset MovieTweetings is loaded and the individual algorithm models are trained on their (fold) datasets. Then, for 100 randomly selected users (each having more than 20 ratings) the system is sequentially requested to calculate (i.e., optimize) the weight vectors for these users and combine their final hybrid recommendation lists.

[5]`http://pybrain.org`

The experiment was repeated with 1, 2, 3, 4, and 5 computing nodes and for two fold dataset settings: 2 and 4. For each of these configurations, the execution times of the individual phases of the recommendation process were measured and are displayed in Fig. 6.7 (exact numbers available in Table 6.2). The '4 fold, 1 node' configuration failed to complete because the required amount of RAM exceeded the available RAM in a single computing node (16GB).

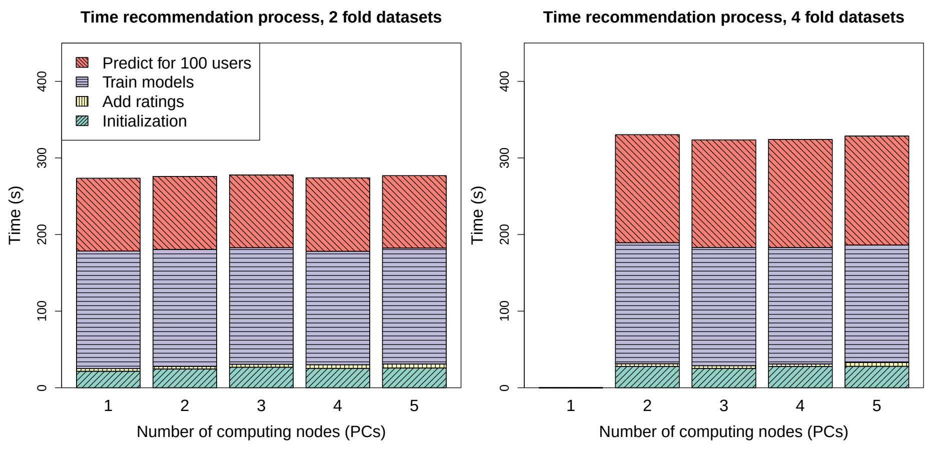


Figure 6.7: The execution times of the individual phases of the complete recommendation process deployed on hardware configurations ranging from 1 to 5 computing nodes and for 2 (left) or 4 (right) fold datasets.

From the figure the initialization time for the different configurations seems identical, but closer inspection reveals a small increase for configurations of more than 1 computing node. This increase in time is caused by the required extra network communication overhead that is needed to signal the other computing nodes. For the same reason also the time for the adding of the ratings increases. The time to train the models and the final prediction time interestingly remain nearly unchanged for an increasing number of computing nodes (both for the 2 fold and 4 fold results). This observed behavior supports our claim that when all processes in the system are able to run in parallel (each on its own dedicated processor core), the end performance would only be limited by the slowest integrated individual recommendation algorithm. Table 6.3 lists for every experimental configuration the consequential number of parallel spawned processes versus the number of available processors. Only the single computing node configurations require more processors than available, and so for these conditions the execution times may be suboptimal.

	#Folds = 2				
#Nodes:	1	2	3	4	5
Start time	21.6	24.5	26.8	25.4	25.7
Rating time	3.4	3.4	3.9	4.7	5.2
Train time	153.5	152.8	152.2	148.1	151.3
Predict time	95.0	95.2	95.0	95.9	94.6
Total time	273.5	275.9	277.8	274.1	276.8
	#Folds = 4				
#Nodes:	1	2	3	4	5
Start time	****	27.9	25.4	27.7	28.0
Rating time	****	3.4	3.4	3.3	5.0
Train time	****	158.0	154.5	152.0	153.4
Predict time	****	141.0	140.3	141.2	142.2
Total time	****	330.3	323.6	324.2	328.6

Table 6.2: The execution times of the individual phases of the complete recommendation process deployed on hardware configurations ranging from 1 to 5 computing nodes and for 2 or 4 fold datasets.

The effect is very limited visible in the training time which is increased by a few seconds. Among the 3 chosen algorithms for this experiment, 2 of them finish fast, which means that 6 out of the 10 parallel computing processes will finish fast, allowing the 2 extra processes to start with only a few seconds delay. For the other (more than 1 node) configurations the total training time will be equal to the time it takes for the slowest algorithm (i.e., *LatentFeatureLogLinearModel*) to complete.

Folds \Nodes	1	2	3	4	5
2	**10/8**	10/16	10/24	10/32	10/40
4	**16/8**	16/16	16/24	16/32	16/40

Table 6.3: The number of spawned parallel processes versus the number of available processor cores (each computing node has 8 cores) for the different experimental configurations.

While the total system execution time does not decrease with an increased number of computing nodes (as expected), it is also interesting to note that it does not increase. Scaling a software system over multiple computing nodes may often increase the communication overhead required to manage the running instances and therefore introduce some form of delay linked with the number of computing nodes. Thanks to a high-speed network infrastructure and some implementation optimizations (e.g., prefetching in Section 6.5.1) we were able to reduce the parallel overhead to an absolute minimum.

When comparing the 2 fold configuration with the 4 fold results, very similar figures can be noted. The time to train the models for a 4 fold configuration is equal to the time for the 2 fold configuration, again illustrating how the training time is independent of the number of folds, algorithms or available computing nodes. When a sufficient number of parallel processors are available, the training time will equal the time for the slowest individual recommendation algorithm to complete its work. The main difference between the 2 and 4 fold configurations is the difference in prediction time. Because the 4 fold configuration has more fold datasets, the optimizer will have to take more data into account to optimize the user weight vectors, which explains the increase in execution time.

6.6.1.2 Weak Scalability

To experiment with the weak scaling ability of our system, we perform a similar experiment but this time the number of computing nodes (i.e., workers) stays constant while varying the dataset size (i.e., amount of work to be processed). The following list describes the experimental setup.

- *Dataset*: $40K, 80K, 120K, 160K, 200K$ MovieTweetings snapshots
- *Algorithms*: *MatrixFactorization*, *SlopeOne*, *LatentFeatureLogLinearModel*
- *Computing nodes*: 5
- *Fold datasets*: 2, 4

The algorithms used in this experiment are identical to those of the previous experiment, again using their default initialization parameters as set in MyMediaLite version 3.10. The properties of the specific MovieTweetings snapshots are detailed in Table 6.4.

Just as before, the system was instructed to run through the consecutive phases of initialization, adding ratings, training models and predicting for 100 randomly selected users with more than 20 ratings. The

	40K	80K	120K	160K	200K
#Ratings	40,000	80,000	120,000	160,000	200,000
#Users	9,063	14,180	19,337	22,259	25,011
#Items	6,798	9,419	11,595	13,445	14,732

Table 6.4: The basic properties of the different MovieTweetings snapshots used in this experiment.

experiment was repeated for iteratively growing dataset sizes and for 2 and 4 fold datasets. The execution times of the individual phases are displayed in Fig. 6.8 and detailed in Table 6.5.

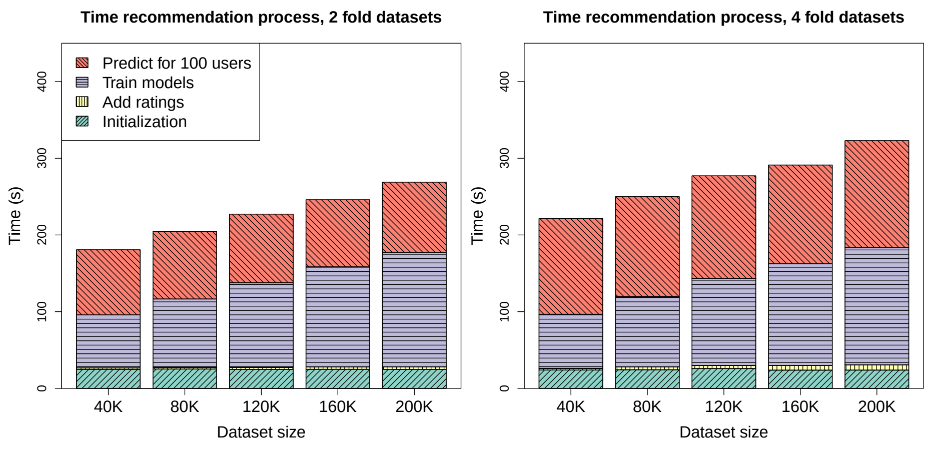


Figure 6.8: The execution times of the individual phases of the complete recommendation process for different sizes of the rating dataset ($40K$ to $200K$) and for 2 (left) or 4 (right) fold datasets.

For the weak scaling experiment every configuration was run on 5 computing nodes each featuring 8 processing cores and so this time the number of spawned processes did not exceed the number of available cores. Since all processes could be divided over 5 different computing nodes no RAM issues occurred and every configuration could be completed.

The initialization time follows the same patterns as in previous results, but the time to add the ratings increases more. This makes perfect sense as the increasing datasets will require more time to process. While the time to train the models remained the same in previous results, here the training time increases linearly with the increasing dataset size. This was again to be expected since the end performance of the system will be

	#Folds = 2				
Dataset size:	40K	80K	120K	160K	200K
Start time	24.9	25.1	24.9	25.0	24.9
Rating time	1.6	2.0	2.5	2.8	3.2
Train time	69.2	89.4	110.2	130.6	149.4
Predict time	85.0	88.1	89.7	87.5	91.4
Total time	180.7	204.5	227.3	245.9	268.9
	#Folds = 4				
Dataset size:	40K	80K	120K	160K	200K
Start time	23.8	24.3	25.8	23.8	23.8
Rating time	2.2	3.7	4.1	6.0	6.8
Train time	70.7	91.7	113.4	132.5	152.8
Predict time	124.5	130.3	133.8	128.9	139.5
Total time	221.3	250.0	277.1	291.3	322.9

Table 6.5: The execution times of the individual phases of the complete recommendation process deployed on 5 computing nodes for varying dataset sizes ($40K$ to $200K$) and for 2 or 4 fold datasets.

depending on its slowest component, the *LatentFeatureLogLinearModel* algorithm, which takes linearly more time to train for increasing rating dataset sizes. The time to train for the 2 fold dataset configuration is again equal to the 4 fold dataset configuration as was observed in the strong scaling scenario.

Two observations can be noted regarding the prediction times. First, the time it takes to sequentially predict for 100 random users is again higher for the 4 fold configuration than the 2 fold, which is caused by the increased complexity in optimizing the user weights vector over multiple fold datasets. Second, the prediction time also seems to increase as the dataset size grows larger. The reason for this is linked with the selection of the random users for each dataset. While in previous experiment the 100 random users were selected and then re-used for the different configurations, here the random selection process had to be repeated for every dataset size (a user selected in the $200K$ snapshot might not be present in the $40K$ snapshot). We counted for each selection of 100 random users per dataset size the total number of ratings for those users and found it to be highly correlated with the final prediction execution time. More ratings will lead to larger cardinalities of the test fold datasets used for the optimization of the user weight vectors, which again increases the complexity of the prediction task. Because in the larger dataset sizes there are more users with >20 ratings, the chance of randomly selecting users with more ratings is larger than for the small dataset sizes.

6.6.2 Responsiveness

While the previous experiments focused on scalability, they also provide some insight into the responsiveness of the system. We have defined the requirement of responsiveness as being able to respond to user requests and changing input data in real-time. Or at least appear as such to a user. In the experiments, we deployed our system and measured the execution times of the different phases. The last phase of the experiment triggered the weight vector optimization by generating recommendations for 100 random users. As the results of the experiments revealed, the total time for this phase was for most configurations below 100 seconds, and therefore <1s per user, which is acceptable for most online scenarios. In less than one second, the live optimization approach managed to take into account the ratings of the user and personalize its weight vector for improved hybrid recommendation results. When no new ratings are available, and the weight vector has already been calculated, the prediction time will be even less (in terms of *ms*). For the configurations that showed higher prediction times (e.g., 139.5*s* for the 4 fold, 200K, weak scaling scenario), parameters of the optimization approach can be modified to speed up the process (e.g., the *MaxEvaluations* setting of the *StochasticHillClimber* method).

6.6.3 Live Optimization Versus Offline Retraining

We proposed a live online optimization strategy that, complementary with offline training, can be used to provide users with a sense of real-time interaction. It would however be interesting to know how the qualitative improvement of the live optimization relates to that of the offline training phase. The frequency in which the offline models should be retrained may be configured depending on how well the live optimization performs. If the live optimization phase shows equal predictive power as retraining the offline models, then the latter may be disregarded altogether (since it is computationally more complex and time-consuming).

To investigate the recommendation quality impact of the live optimization phase versus the offline retraining phase, we propose the following experimental approach. For a number of random users in a dataset we split their ratings in a distinct training set and test set. We then further split the training set into two parts. We thus end up with three datasets, two for training and one for evaluation (test set). We want to compare the recommendation quality in three scenarios. The first is the baseline scenario in which the models are trained offline on half of the training set. The second scenario, also trains the models offline on half of the training set, but as soon as training is completed, the other half of the training set is added online and processed by the live optimization approach. In the third and final scenario the models are trained offline on all of the training set. For each of these scenarios we compare the recommendation quality by calculating the RMSE accuracy metric using the test set. We graphically illustrate the experiments in Fig. 6.9.

By comparing the results from the three experiments, we can inspect the capability of the live optimization approach to use the newly added ratings to improve the recommendation quality (scenario 2). And more interestingly, we can compare this to the scenario where all the ratings (of the training set) would have been included in the offline model (scenario 3).

We run the experiment on two different datasets: the MovieTweetings and MovieLens dataset. Both consist of movie rating data, are identically structured and provide a 100K ratings snapshot. MovieLens does however only integrate users with a minimum number of 20 ratings, while MovieTweetings has no such filtering. To be able to make a fair comparison we therefore also constructed a similar variant of MovieTweetings by removing users with less than 20 ratings. From these three datasets we randomly selected 500 users and simulated the above described scenarios, each time measuring the final RMSE recommendation accuracy metric.

The recommendation algorithms used for these scenarios are the same as described in the scalability experiments: *MatrixFactorization*, *LatentFeatureLogLinearModel* and *SlopeOne* all originating from the MyMediaLite recommendation framework.

Fig. 6.10 shows the results for the MovieTweetings dataset. The three scenarios are presented on the X-axis, their corresponding RMSE values on the Y-axis. The first (baseline) scenario shows the highest RMSE value, which was to be expected as in this scenario the least amount of ratings are used. For scenario 2, where the remaining half of the training ratings were added to the live system, the RMSE value slightly improves.

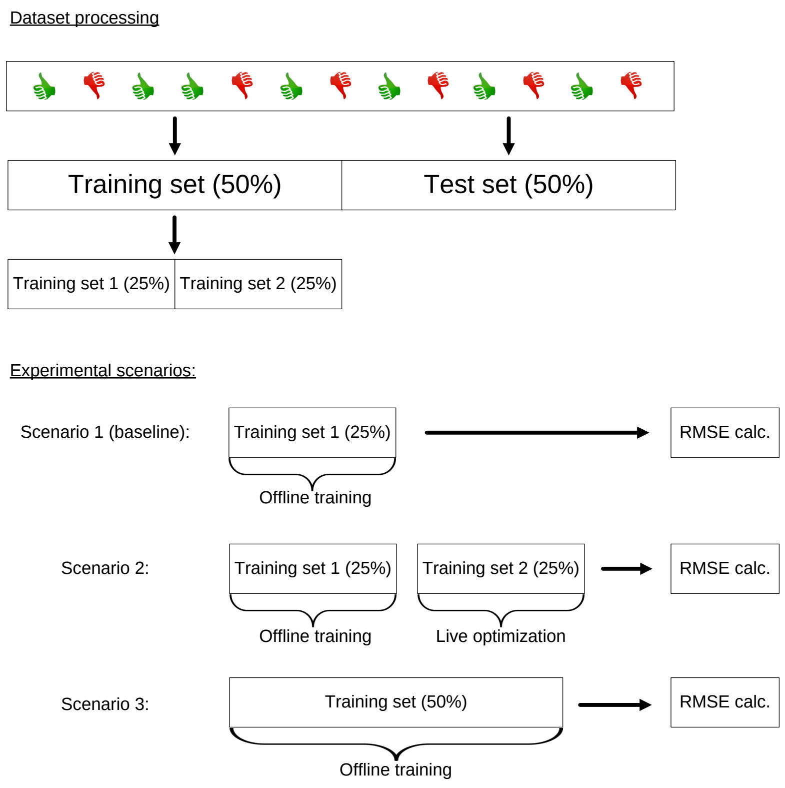


Figure 6.9: Visualization of the data processing strategy and the 3 experimental scenarios for comparing the impact of the offline training phase and live optimization phase on the final RMSE value for 500 random users. Note that the dataset split is performed for every of those 500 users and their resulting RMSE values are averaged.

The biggest RMSE improvement is however clearly found in scenario 3.

Fig. 6.11 shows the results for the other two datasets: MovieLens and MovieTweetings with only users with minimum 20 ratings. For these datasets similar results can be noted: the live optimization approach slightly improves the RMSE value of the baseline scenario, but the best gain is found when all of the ratings are used to train the individual models.

We note that the exact values of the calculated metric (here RMSE) may differ depending on the individual recommendation algorithms, the applied optimization approach, and the tuning of many different configuration parameters (such as the number of fold datasets, etc.). The experiments do however illustrate the relative performance of the live optimization approach versus the predictive power of offline training the models on more data.

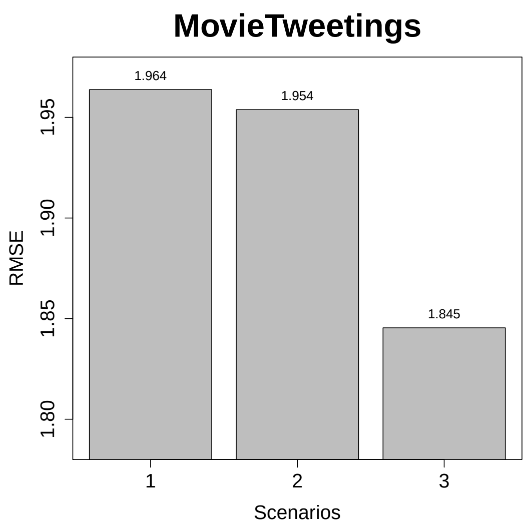


Figure 6.10: The measured RMSE value of 500 random users of the Movie-Tweetings dataset for three experimental scenarios.

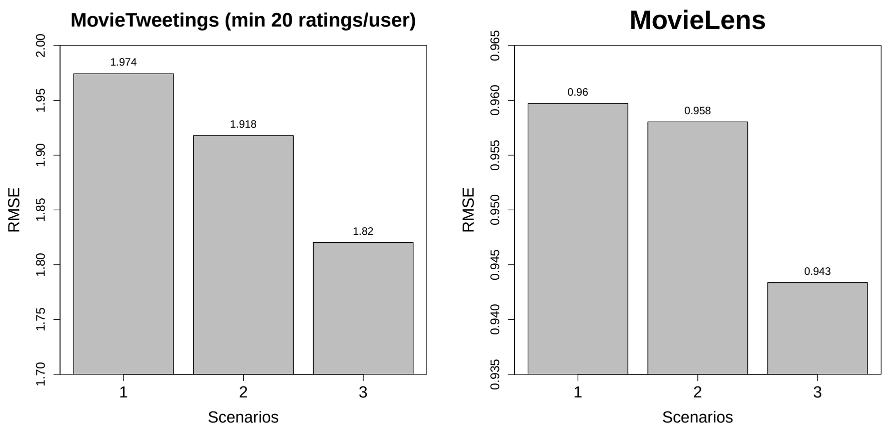


Figure 6.11: The measured RMSE value of 500 random users of the Movie-Tweetings dataset where each user has at least 20 ratings (left), and the Movie-Lens dataset (right) for three experimental scenarios.

6.7 Discussion

We have shown that the performance and scalability of the online recommender system can indeed be reduced to the performance of the slowest integrated individual recommendation algorithm as long as the underlying hardware configuration provides sufficient parallel processing power. The complexity of the hybrid optimizer did however turn out to be influenced by the number of used fold datasets. Because of this, a trade-off will have to be made between having many folds (e.g., 10) to reduce the chance of overfitting the model and having a small number of folds to reduce the complexity (i.e., increasing the speed) of the optimizer component. The number of fold datasets should therefore be sufficiently large while making sure the performance of the optimizer component can still be considered fast enough to guarantee instant responsiveness to new ratings.

From the results we learned that offline retraining the individual recommendation models has a significantly higher impact on recommendation quality than the online optimization approach. While the latter is useful for providing a sense of real-time interactivity with the system, offline retraining should be scheduled as much as possible in order to guarantee the best recommendation results. When dedicated computation hardware is available, we propose to continuously retrain the individual models in the background.

In the end, the presented hybrid optimizing system offers sufficient flexibility for a customized configuration for any specific use case. Configurable components include the individual recommendation algorithms, the number of folds, the evaluation metric (i.e., the optimization goal) and the optimization method itself.

6.8 Conclusion

We discussed the architectural design of our hybrid optimization strategy and detailed realistic implementation issues to assure the system meets the proposed requirements for an online recommendation scenario: responsiveness, scalability, system transparency and user control. By adopting a client-server architecture we showed how the system can be distributed across multiple computing nodes in a very flexible and transparent way, allowing multiple recommendation algorithms to run in parallel for optimal performance. We illustrated how the results and internal processes of the system could be visualized to users in the form of a responsive user interface allowing users an advanced and intuitive level of control over their recommendation lists.

Through experimental evaluation we validated the architectural design and our claim that the performance and scalability of the system can be reduced to the performance and scalability of the worst (i.e., slowest) individual recommendation algorithm integrated in the hybrid system. The added overhead of the hybrid optimization was shown to be very limited as long as a sufficient number of computing nodes (or parallel processor cores) are available.

Chapter 7

Online Evaluation of Personalized Hybrid Recommender Systems

7.1 Introduction

We have evaluated our proposed hybrid recommender system offline and discussed its ability to run in an online environment. What now remains is to make the system available to real users and measure their interaction and user satisfaction in an online evaluation experiment. Organizing user experiments is easy for companies with a large customer base, lots of user data and a high-traffic website; it is however less straightforward for a single academic researcher with limited resources. Here, we present our approach towards exposing our dynamic hybrid recommender system to online users by means of a Google Chrome extension called ‘MovieBrain’. We expand on issues as user data collection, hardware deployment, and online experiences.

First we define some of the properties of our recommender system – which we refer to as the *brain* (recommender) – that will have to be taken into account. The brain recommender is a typical *rating prediction* based recommender system which takes user ratings as input and for each user outputs a list of unrated items which are predicted to receive the highest rating values. While the brain (recommender) has no limitation on what item domain it is applied to, it does have some specific runtime properties. For one, the brain is customizable, which means that users can change settings and interact with the brain other than merely providing ratings. Furthermore, the brain is scalable in terms of hardware deployment, meaning that it can run on a single computing machine but is also able to flexibly distribute its processing tasks across multiple nodes if available. Like many other rating prediction based recommendation algorithms, the brain requires some time to train its model during which it is unresponsive and thus unable to answer any user requests. After the initial training, the brain is ready to recommend items to users.

Considering our academic context, the available resources to implement our online evaluation project are quickly summed up: no existing user base, no online platform, no team of programmers or marketeers, and a very limited financial budget. What we do have at our disposal is a small webserver and access to a high-performance computing (HPC) infrastructure which is however shared with other university members and works on a request-only basis. These constraints, resources and properties can be summarized in the following list.

Brain recommender properties

- User customizable
- Scalable hardware deployment
- Unresponsive training phase

Available resources

- Small webserver
- Limited financial budget
- Access to a shared HPC environment

In the following sections we detail how we expose our brain recommender system to live user testing while respecting the aforementioned constraints and resources of our typical academic context. Note that for related work, we refer to previous chapters where we already discussed research on online and user-centric experiments.

Research Questions

- How can a user-centered experiment be deployed with limited resources?
- How do users interact with a self-configuring hybrid recommender system?

- How useful is the hybrid system to users?

7.2 MovieBrain: a Movie Focus

The item domain of movies has been much explored in the context of recommender systems research, mostly because of the availability of public rating datasets as the MovieLens and Netflix dataset. For many researchers these datasets used to be the only source of realistic rating data available. For the online evaluation of our brain recommender system we also opt for the movie domain, but for different reasons. We would like to integrate our recommendation service with an existing information system to be able to primarily focus on the recommendation job and keep the implementation overhead and required user effort to a minimum. We chose the movie domain because of the integration possibility with the IMDb website. This website offers a vast amount of information on a wide catalog of movies and also collects and aggregates ratings on a 1 to 10 star rating scale. Interestingly, many users provide such ratings and furthermore the ratings can be made public. If we were able to somehow extend the functionality of the IMDb platform we could re-use the ratings already provided by IMDb users and thus reduce the initial threshold of asking users for ratings before being able to provide recommendations (i.e., avoid the cold-start problem).

Another reason for focusing on the movie domain is the availability of our MovieTweetings dataset which consists of IMDb ratings that have been posted on Twitter. The MovieTweetings dataset nicely complements the public ratings on the IMDb platform and helps to further alleviate any cold-start symptoms. To summarize, in order to evaluate our brain recommender system we focus on the movie item domain by extending the IMDb platform and complement existing public ratings with the MovieTweetings dataset. To do so, we created a Google Chrome (browser) extension which we named *MovieBrain* and discuss more extensively in the following sections.

7.3 A 3-tier Architecture

The online evaluation task of our brain recommender system naturally requires at least a two-component architecture: a **visual front end** for user interaction and a **back end** for computation purposes. We add

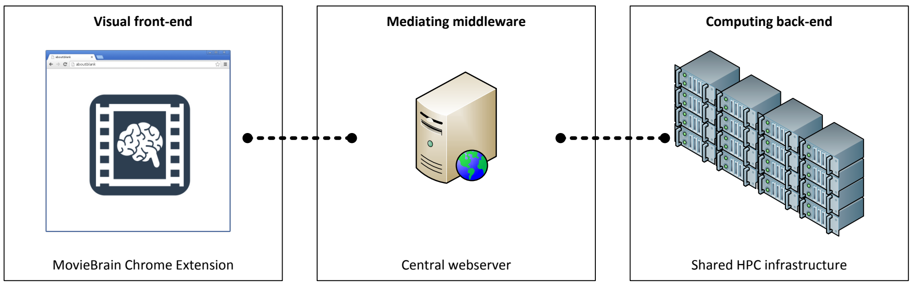


Figure 7.1: The 3-tier architecture connecting the visual front end with the computing back end through a mediating middleware layer.

however an additional component in the middle for control, caching and data management. As indicated in Fig. 7.1, a Google Chrome extension (i.e., a plugin for the Google Chrome Internet browser) is used to handle user interaction on the front end side. A webserver functions as the mediating middle layer and in the back end our university-shared HPC infrastructure powers the brain itself.

7.4 Computing Back End

As mentioned, our university offers its researchers a HPC infrastructure free of charge. Since the infrastructure is shared among researchers, certain limitations are however in effect. For one, all computation tasks need to be compiled in the form of job scripts which require to specify in advance the anticipated resource usage. These resources include the number of computing nodes, processing cores per node, the amount of used RAM and expected computation time. Jobs crossing their resource boundaries are killed instantly. Furthermore all jobs are limited to a maximum continuous execution time of 72 hours. Jobs submitted to the infrastructure are processed in a queue and systematically started by a job scheduler that takes into account the availability of the requested resources and overall fair-use quota among all users. As of a result, heavy resource dependent jobs will be executed only when the resources are available and priority will be given to users that have been using the HPC less frequent. Since a limited budget does not allow the acquisition of a dedicated calculation server (or renting e.g., Amazon EC2 machines), the constraints imposed by the infrastructure at hand will need to be aligned carefully with the properties of the brain recommender. We summarize the following list of infrastructural challenges that need to be tackled at the computing back end of our MovieBrain project.

HPC challenges

- Jobs scheduled instead of executed instantly
- Limited job execution time
- Job execution depending on available (and requested) resources

With a dedicated calculation server, setting up the computing back end would be a straightforward task of installing the brain recommender on the server and executing it. In the case of our shared infrastructure the task of running the brain requires more attention. The job execution time on the cluster is limited but obviously we want the brain recommender to remain continuously active during the online evaluation. Assuming the online evaluation period will exceed 72 hours, a *multi-job strategy* imposes itself.

7.4.1 Multi-Job Strategy

Instead of running the brain in a single job, we schedule multiple *brain jobs* such that their execution times overlap. By maintaining an index to running jobs, the system (i.e., the middleware) can be pointed to a new active job when an old one is killed for exceeding the execution time limit. By starting new jobs at fixed intervals (e.g., every 50 hours) the brain recommender could (at least theoretically) be active indefinitely.

One problem with this approach is that jobs are only started when the requested resources are actually available. Due to the infrastructure being shared, the availability of resources may vary greatly from e.g., 100 free computing nodes with 8 parallel processing cores and 16GB of RAM each, to 1 partially available or even no available nodes. The brain recommender – in its current configuration[1] – requires at least 8 processing cores and 8GB of RAM. Fortunately however[2], one of the properties of the brain recommender was hardware scalability, which allows to align job resource requirements with the computing hardware availability. When few resources are available, the brain jobs can be set

[1]The amount of needed resources depends on the specific configuration of the recommender e.g., involved recommendation algorithms, number of dataset folds, optimization method, etc.

[2]This is of course no coincidence. The brain has been made to scale across hardware exactly to be able to cope with such changing hardware configurations.

to run on multiple shared computing nodes with limited resources rather than requiring a dedicated computing node which may not be available.

The combination of submitting multiple jobs overlapping in time, and the flexibility to adjust job resource requirements, greatly increases the possibility to keep a brain recommendation process active for an online evaluation period on our shared HPC environment. We do however need to take into account the situation where the complete HPC cluster is busy and no jobs can be started regardless of how few resources are requested. Since such a situation may occur beyond our control, the best way of dealing with this is making sure that no data is lost, and users are gently made aware of the temporary unavailability of the brain at the front end of the MovieBrain system (i.e., Chrome extension). Regardless of the hardware platform, handling unavailability will be crucial anyway to prevent user frustration in the inevitable event of a server crash, Internet outage or other unforeseeable circumstances.

7.4.2 Multi-Brain Strategy

Using the multi-job strategy, the challenges imposed by our shared HPC environment can be met. The challenges originating from the intrinsic runtime properties of the brain recommender however remain. The brain requires a training phase during which internal models representing user preferences are learned. During such training, the brain is unresponsive. In the previous chapter we illustrated our live optimization approach which handled new ratings in real-time. Our results showed however that offline retraining yields the best quality improvement and should preferably be run in the background as much as possible.

We therefore propose also a *multi-brain strategy* which entails starting multiple brain jobs in parallel. One brain may be training, while another (already trained) brain process remains available to respond to user queries. Iteratively training either one of the brain processes, keeps the brain as updated as possible to new user data without being offline. Extending this approach to multiple parallel brain servers allows to load balance user requests and thus increases the perceived user responsiveness even further.

In conclusion, Fig. 7.2 illustrates the complete process flow of the brain recommender executing within the constraints of the available computing back end. Note that the combined *multi-job* and *multi-brain* strategies require a substantial amount of parallel running jobs, which maps per-

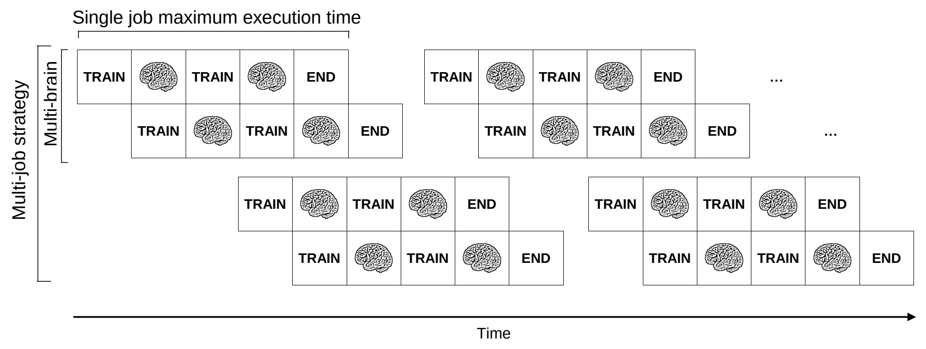


Figure 7.2: The multi-job and multi-brain strategies illustrated in time. The multi-brain strategy alternately trains brain processes to be as responsive as possible to new user data (i.e., ratings). The multi-job strategy lets overlap multiple jobs to make sure there is always an active brain process when a job on the HPC infrastructure exceeds its maximum allowed execution time.

fectly on a HPC infrastructure but rather poorly on non-parallel hardware.

7.5 Mediating Middleware

The middleware layer, where our webserver resides, links the visual front end with the computing back end. All communication from front end to back end is passing through the webserver. This indirect communication architecture has many advantages including added control, security, and improved data management. User interactions on the front end can be filtered and sanitized before sending them to the multiple back end calculation servers. For example, when the same action is repeated multiple times by a user in a short period of time (e.g., user constantly refreshing her recommendation list), those actions can be easily aggregated into a single request and the results can even be cached in the middleware to prevent users from unnecessarily overloading the back end. Shielding the front end from directly triggering back end actions also increases security by not exposing the valuable hardware to possibly malignant users.

Fig. 7.3 depicts the middleware in more detail. An external application programming interface (API) exposes brain recommender functionality to the outside. The API directs requests to the *request controller* component which are then directed to the *HPC controller* if necessary. The middleware also provides data management and thus the request con-

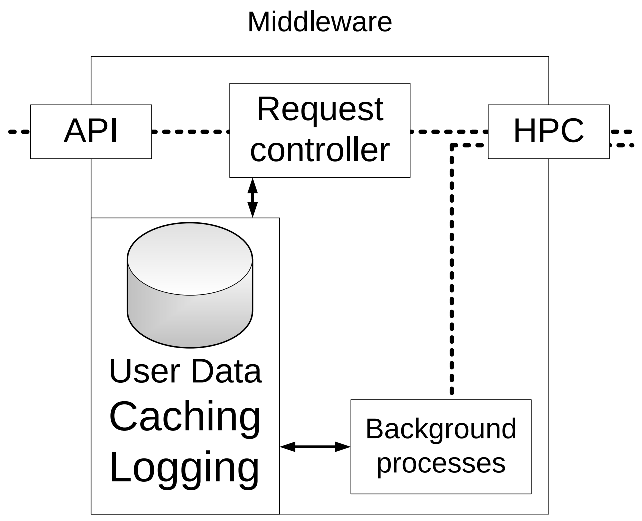


Figure 7.3: A more detailed view of the middleware functionality. The middleware connects requests from the front end to the back end, manages a database and periodically runs background processes for e.g., the steering of the brain processes in the HPC infrastructure.

troller has access to a local database which can be used for the storage of user data (e.g., user ratings), caching and logging. By keeping track of the training cycle of the brain servers in the back end, the middleware is capable of predicting which user requests should be answered from cache and which need to be transferred to the brain (e.g., after retraining the brain, recommendation results could be different). Through such a caching strategy, requests to the brain back end can effectively be reduced to the bare minimum.

Apart from exposing the functionality of the brain recommender, the middleware also runs *background processes*. Those processes (implemented as *cronjobs* on our linux-based webserver) e.g., checking the IMDb profiles of active users for new ratings and keeping the database up to date. Another crucial background task is the steering of the brain eco-system in the back end. The middleware keeps track of running brain processes, initiates the training of brains and periodically starts new brain jobs on the HPC to ensure the availability of active brain processes. When a new brain process is started, user ratings from the database are imported and complemented with ratings from the MovieTweetings dataset to avoid cold-start symptoms for new users or unrated items as much as possible.

7.6 Visual Front End

Since the middleware exposes the brain functionality through an API, there is no limitation as to what types of front end systems could be attached. Using the API to construct an HTML-based website would be just as easy as building e.g., a smartphone app. However, for our MovieBrain front end, other aspects than mere implementation issues should also be taken into account. The brain recommender system offers movie recommendations, but this alone, may not suffice to attract users to the system. If the front end would be a stand alone website, users would need to register, login, provide ratings and only after having contributed a considerable amount of effort would they be able to actually benefit from the provided recommendation service.

We aim to optimally attract online users to our system by reducing the required effort to benefit from the service and integrating as much as possible in users' personal workflows. For that, a web browser plugin is extremely well suited.

Browser plugins integrate seamlessly in people's everyday Internet activity (i.e., browsing the web), they allow to inject custom code into existing websites and track users browsing behavior. We opted for a Google Chrome extension, but a Firefox plugin would have suited equally well. Our MovieBrain Chrome extension extends the IMDb website functionality by offering user customizable movie recommendations (IMDb currently only offers a limited and static recommendation list). The beauty in this workflow lies in the fact that for users interested in the MovieBrain service, their ratings already available in IMDb can simply be re-used. Furthermore IMDb is a widely used and popular web platform, so requiring that users have an IMDb account should not be preventing user adoption, on the contrary, MovieBrain may benefit from increased user trust through its association with the familiar IMDb platform.

Another advantage of a front end Chrome extension is improved scalability. Since an extension is a self-contained file hosted at the client side, the impact on the webserver will be limited to HTTP calls to the API.

A Chrome extension may manifest itself in various forms[3]. Most visually, icons can be added in the browser toolbar or address bar triggering certain popups or actions, but just as easy web code (mostly HTML and *JavaScript*) can be injected in websites browsed by the user. The

[3]For more information visit `https://developer.chrome.com/extensions/overview`.

combination of both offers a more than sufficient set of tools for the construction of our MovieBrain front end.

Overall, the look and feel of the front end is based on the popular Bootstrap[4] framework and custom JavaScript code to make it as responsive as possible and meet the current-day high-quality design standards that Internet users have become accustomed to.

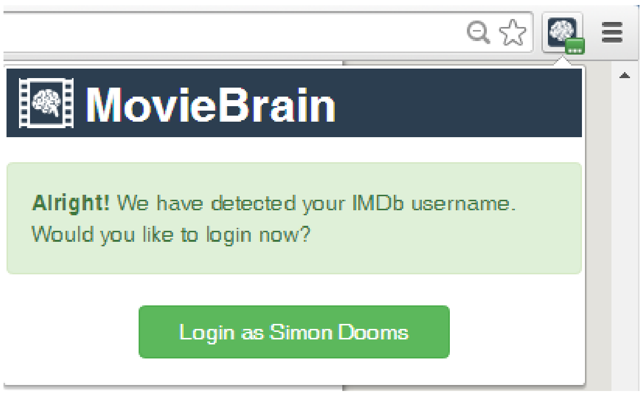
7.6.1 Login Made Easy

Our MovieBrain project integrates with the IMDb platform and so there is a one-to-one mapping of MovieBrain users to IMDb users. Therefore we can simply identify users with their IMDb userid, rather than requiring them to come up with new (MovieBrain-) user credentials. With reduced user effort in mind, the tedious step of providing user login and password can even be avoided altogether by injecting some (client-side) user-detecting code into the HTML source of web pages on the IMDb domain. The MovieBrain Chrome extension installs a small icon in the browser toolbar advertising the MovieBrain service to the user. When users click the icon for the first time, a popup is shown asking users to browse to (or refresh) an IMDb page after being logged in to IMDb. By doing so, we are able to detect the IMDb userid and at the same time authenticate the user without asking for credential data. The user is notified that its IMDb username has been detected successfully by a small change in the appearance of the toolbar icon, and logging in to the MovieBrain system is then literally as easy as clicking a button (illustrated in Fig. 7.4).

7.6.2 Background Detection

As mentioned, a Chrome extension has the ability to inject web code into browsed websites. This allows us to detect – and act on – some interesting user actions. Other than detecting the IMDb username, the MovieBrain front end detects two additional events. Firstly, the front end detects when a user rates a new movie on IMDb (using the web-based rating system). Detecting new ratings is interesting because the middleware can be notified of this event, triggering (in the background processes) the scanning of the user's IMDb profile for the new ratings, which can subsequently be inserted into the brain processes leading to

[4] `http://getbootstrap.com`



Alright! We have detected your IMDb username. Would you like to login now?

Figure 7.4: Screenshot of the MovieBrain login form where the IMDb username is automatically detected instead of troubling the user with typical user/password forms.

an overall more responsive recommender system. The second detected event is a change in the total ratings count of the user. Whenever a new rating is detected, the total number of ratings (of the respective user) is retrieved from IMDb and added as a badge indication to the MovieBrain icon in the toolbar (Fig. 7.5). The badge indication is an easy way for users to keep track of their number of ratings and may even serve as a reminder to keep rating movies once in a while.

7.6.3 Getting Recommendations

After a user has clicked the login button, personalized movie recommendations are shown (Fig. 7.6). New users will however be unfamiliar to the brain until their user data is transferred and the brain retrained. Be-

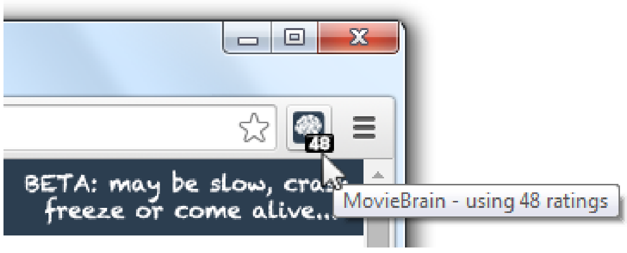
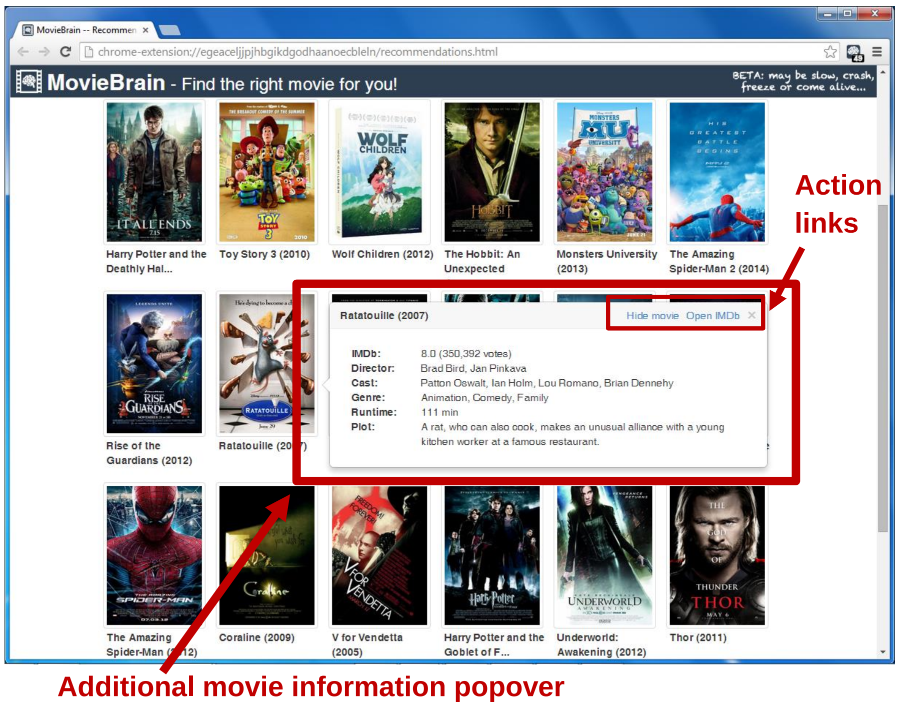

48
BETA: may be slow, crash, freeze or come alive...
MovieBrain - using 48 ratings

Figure 7.5: Screenshot of the badge indication of the MovieBrain icon which advertises the number of user ratings to stimulate active rating behavior.


MovieBrain -- Recommen
chrome-extension://egeaceljjpjhbgikdgodhaanoecbleln/recommendations.html
MovieBrain - Find the right movie for you!
BETA: may be slow, crash, freeze or come alive...
Harry Potter and the Deathly Hal...
Toy Story 3 (2010)
Wolf Children (2012)
The Hobbit: An Unexpected
Monsters University (2013)
The Amazing Spider-Man 2 (2014)
Rise of the Guardians (2012)
Ratatouille (2007)
The Amazing Spider-Man (2012)
Coraline (2009)
V for Vendetta (2005)
Harry Potter and the Goblet of F...
Underworld: Awakening (2012)
Thor (2011)
Hide movie
Open IMDb
Action links
Additional movie information popover

Ratatouille (2007)

IMDb: 8.0 (350,392 votes)
Director: Brad Bird, Jan Pinkava
Cast: Patton Oswalt, Ian Holm, Lou Romano, Brian Dennehy
Genre: Animation, Comedy, Family
Runtime: 111 min
Plot: A rat, who can also cook, makes an unusual alliance with a young kitchen worker at a famous restaurant.

Figure 7.6: Screenshot of the MovieBrain front end illustrating the visualization of the recommendation results. By default only movie thumbnails and titles are shown but when clicked a popover information panel reveals additional movie details.

cause of the *multi-brain* strategy, brain models are retrained constantly and so the maximum time new users will have to wait for personalized recommendations is the time it takes for one such training to complete (about 10 minutes for our current brain configuration). Before the availability of their (personalized) recommendations, users are shown a non-personalized list of popular movies to keep their attention. When available, recommendations will be requested from the brain and thereafter served from cache (in the middleware) until further retraining or the arrival of new data (i.e., new ratings or user settings) may have caused changes.

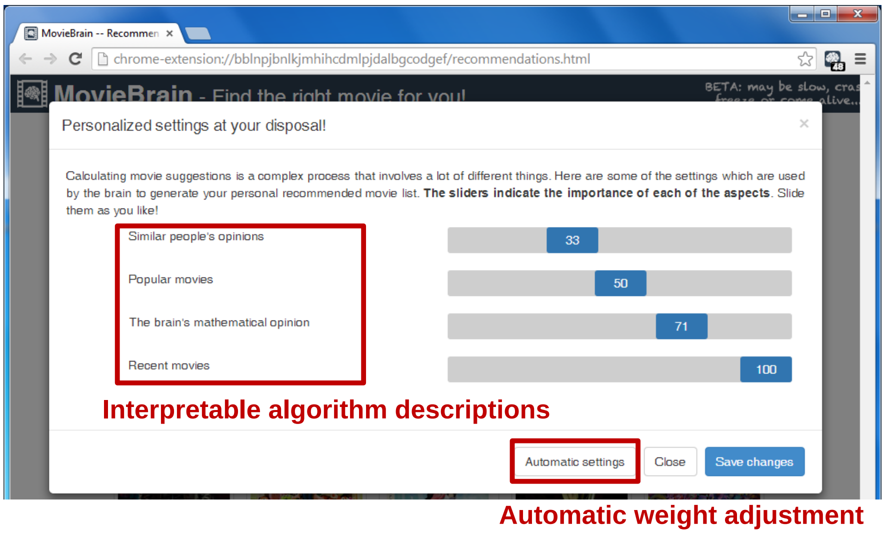
For the visualization of the recommendation results we combined all of our experience from research in previous chapters and experiments. Users want visual and attractive user interfaces, but as we noted in previous results, some users will want more information than others. We therefore opted for a two-level visual design where movies are initially represented by their movie poster thumbnails and titles. Clicking a movie, triggers an additional popover information panel with more detailed movie information and includes two action links: *Hide* and *Open IMDb*. The '*Hide*' link hides the movie from the recommendation list and notifies the middleware to prevent it from showing up again in future recommendation lists. Clicking the '*Open IMDb*' link opens the movie information page on the IMDb website allowing users to read more information, extend their watchlists or rate the movie. By default we show no more than 20 movies but more can be dynamically loaded by clicking the '*More*' button at the bottom of the page (visible in Fig. 7.9).

7.6.4 System Transparency and User Control

The added value of the brain recommender system was customizability in the sense that users can interact with settings of the recommender which allows them to improve the recommendation quality according to their own personal expectations. The front end supports user interaction by providing a settings dialog window with sliders and control buttons (Fig. 7.7). Changed settings are reported to the back end brain processes using the middleware API causing the recommendations for the respective user to be recalculated in a matter of seconds (no *slow* retraining needed).

MovieBrain currently integrates 4 individual recommendation algorithms. For each of these algorithms a visual slider element allows to tweak its importance in the recommendation calculation process. Next to each slider a simplified description of the recommendation algorithm (i.e., in terms of its most defining feature) makes the settings more interpretable for non-technical users. There is also an option to set the settings to '*Automatic*' which allows the system to choose appropriate settings for the user (using the optimization strategy detailed in previous chapters).

In Chapter 3 (Section 3.3.2), we learned that users greatly appreciate the ability to filter movie recommendations on *genre* to more closely match volatile contextual expectations; sometimes users just want a *comedy* instead of a highly interesting and sophisticated documentary. In the MovieBrain front end, users can filter genres by intuitively dragging genre buttons to either an *exclude* or *include* area (Fig. 7.8). Excluding a genre, causes all recommended items from that genre to be discarded from the recommendation list. If genres are dragged onto the include area, only recommended items containing at least all of these genres will


MovieBrain -- Recommen
chrome-extension://bblnpjbnlkjmhihcdmlpjdalbgcodgef/recommendations.html
MovieBrain - Find the right movie for you!
BETA: may be slow, cras
freeze or come alive..
Interpretable algorithm descriptions
Automatic weight adjustment

Personalized settings at your disposal!

Calculating movie suggestions is a complex process that involves a lot of different things. Here are some of the settings which are used by the brain to generate your personal recommended movie list. **The sliders indicate the importance of each of the aspects**. Slide them as you like!

Similar people's opinions	33
Popular movies	50
The brain's mathematical opinion	71
Recent movies	100

Automatic settings | Close | Save changes

Figure 7.7: Screenshot of the MovieBrain front end illustrating the settings page which allows to modify the importance of individual algorithms. Simplified algorithm descriptions allow users to understand the settings, and weights can be set to automatic.

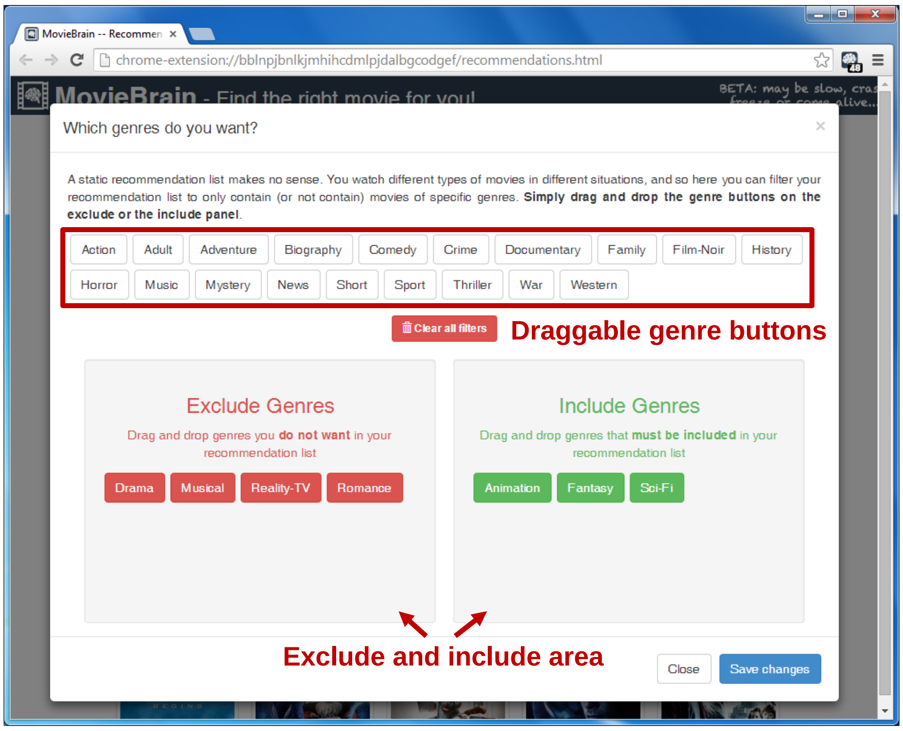
be displayed. This filtering scheme enables users to model fine-grained preferences like 'show me all recommended romantic comedies but exclude animation movies'. Genres are filtered in the middleware after the recommendations are requested from the brain which eliminates unnecessary communication to the brain processes and speeds up the overall system.

An overview of the general layout of the MovieBrain front end is shown in Fig. 7.9. At the top of the recommendation visualization screen the settings and genre filtering screens can be triggered as modal HTML forms.

7.7 Online Evaluation Results

The MovieBrain Chrome extension was made publicly available on the Chrome web store[5]. As more users installed and used the extension, more data was collected which could ultimately be used to evaluate the user experience of the brain recommender in a very realistic usage scenario.

[5]`https://chrome.google.com/webstore`


Which genres do you want?

A static recommendation list makes no sense. You watch different types of movies in different situations, and so here you can filter your recommendation list to only contain (or not contain) movies of specific genres. **Simply drag and drop the genre buttons on the exclude or the include panel.**

Drag and drop genres you **do not want** in your recommendation list

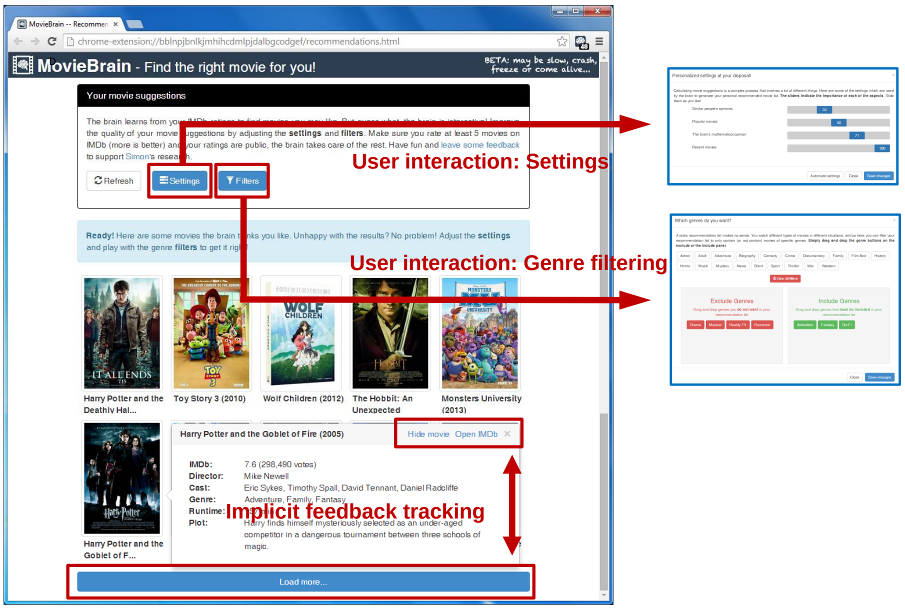
Drag and drop genres that **must be included** in your recommendation list

Figure 7.8: Screenshot of the MovieBrain front end illustrating the genre filtering feature. Genre buttons can be dragged to an include or exclude area which are subsequently filtered in the middleware.

All requests to the middleware API were logged so that user interaction with the front end and typical user behavioral patterns could potentially be analyzed.

Since we wanted the logged data to be a good representative of realistic user behavior, we tried to attract genuinely interested users and avoided recruiting artificial test users as much as possible. Attracting real users to an online service is however challenging. Simply publishing the Chrome extension in the online store does not suffice since there are thousands of other extensions competing for attention. To stimulate real user adoption, we therefore created a complementary website[6], engaged social media channels and published blog posts on appropriate online channels (e.g., *reddit.com*). At the time of analysis, a total of 70 users had installed the extension. In the following subsections we present

[6]`http://www.themoviebrain.com`


MovieBrain -- Recommen
chrome-extension://bblnpjbnlkjmhihcdmlpjdalbgcodgef/recommendations.html
MovieBrain - Find the right movie for you!
BETA: may be slow, crash, freeze or come alive...
Your movie suggestions
Refresh
Settings
Filters
User interaction: Settings
User interaction: Genre filtering
Harry Potter and the Deathly Hal...
Toy Story 3 (2010)
Wolf Children (2012)
The Hobbit: An Unexpected
Monsters University (2013)
Harry Potter and the Goblet of F...
Harry Potter and the Goblet of Fire (2005)
Hide movie
Open IMDb
Implicit feedback tracking
Load more...

The brain learns from yo[illegible] the quality of your movie [illegible]uggestions by adjusting the **settings** and **filters**. Make sure you rate at least 5 movies on IMDb (more is better) an[illegible] your ratings are public, the brain takes care of the rest. Have fun and leave some feedback to support Simon's resea[illegible]h.

Ready! Here are some movies the brain t[illegible]nks you like. Unhappy with the results? No problem! Adjust the **settings** and play with the genre **filters** to get it rig[illegible]

IMDb:	7.6 (298,490 votes)
Director:	Mike Newell
Cast:	Eric Sykes, Timothy Spall, David Tennant, Daniel Radcliffe
Genre:	Adventure, Family, Fantasy
Runtime:	[illegible]
Plot:	Harry finds himself mysteriously selected as an under-aged competitor in a dangerous tournament between three schools of magic.

Figure 7.9: Screenshot of the MovieBrain front end illustrating the general layout, user interaction options via the *Settings* and *Genre filtering* and implicit feedback tracking through the monitoring of the *Hide*, *Open IMDb*, and *More* links.

their activity patterns as measured over 107 days. All data generated by testing or administrative accounts has been removed.

7.7.1 Brain Recommender Configuration

The deployment of the hybrid recommender configuration in the MovieBrain extension used the following settings:

- *Dataset*: MovieTweetings (latest data)
- *Algorithms*: *MatrixFactorization*, *UBCF*, *MyLogPopular* and *MyRecentMovies*
- *Computing nodes*: 2
- *Fold datasets*: 3
- *Train-test ratio*: 6:4
- *Optimization method*: StochasticHillClimber (PyBrain)

For the *MatrixFactorization* algorithm, we used the MyMediaLite framework with its default settings. The UBCF recommender is a typical user-based collaborative filtering algorithm which finds similar users and bases its recommendations on their ratings. This recommendation algorithm is also available in the MyMediaLite framework, but for performance reasons we used our own implementation. The default MyMediaLite version calculates all neighbors from all users before being able to generate recommendations for one user. This process is not only slow and inefficient but also requires (for our dataset) more than 16GB of RAM to keep all similarity values in memory and so could not be used on the default computing nodes of our HPC infrastructure. We implemented our own UBCF algorithm that integrated an intelligent caching approach (see Section 4.5 in Chapter 4) which greatly reduced the required RAM memory and increased the overall calculation speed of the algorithm. The remaining *MyLogPopular* and *MyRecentMovies* algorithms are also self-implemented algorithms which simply rank movies by either their popularity or recentness.

7.7.2 Click Tracking

To gain insight into the resulting user satisfaction, specific user interactivity patterns can be analyzed. In total, 3 satisfaction indicators we consider particularly interesting: clicking on either the *More* link, *Hide* link, or the *Open IMDb* link in the front end. Each of these actions can be regarded as an implicit indicator of user satisfaction e.g. a user that wants to load more recommendations may be positively interested in the system. Fig. 7.10 shows the number of clicks on the three available action links per user.

All three figures clearly show typical long tailed distributions; a few users have a high number of clicks but most users close to none. When considering the *More* clicks, a significant difference can be noted between the highest number of clicks (almost 400) and the lowest (no clicks). It seems that while some users really take the time to browse through endless movie suggestions, most users don't process more than 20 recommendations (i.e., the maximum number of recommendations shown). These users may simply be happy with their first 20 results, or on the other hand may be so disappointed by the first results that they have lost interest.

When we inspect clicks on the *Open IMDb* link however, we find that the number of users that clicked the *Open IMDb* link at least once is

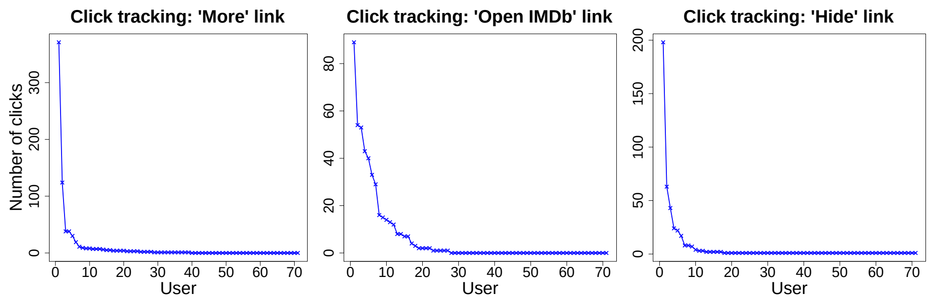


Figure 7.10: The number of clicks on the three action links: *More* (left), *Open IMDb* (middle) and *Hide* (right) per user. Users on the X-axis are sorted by number of clicks to improve visual interpretation.

greater than the number that clicked the *More* link and so data seems to show that a lot of users are indeed happy with (or at least intrigued by) their first 20 results.

One user has used the *Hide* feature almost 200 times. However, while at first we considered hiding recommendations negative implicit feedback, it turned out that users tend to use the feature in another way. Users that hide movie recommendations do not necessarily hate the movie, they might just be bored by getting the same recommendation, or maybe they already saw the movie but didn't rate it. One user wrote the following feedback on a Reddit blog post about the MovieBrain extension:

"*The settings and filters are easily adjustable for when you're trying to get recommendations based on specific criteria, and I personally love the ability to hide selections, because it allows me to constantly be looking for new titles to add to my giant 'to watch' list.*"

The interpretation of hiding a recommendation result may thus not necessarily be negative since users might use the feature to maximize their recommendations browsing experience.

7.7.3 User Activity

It would be interesting to know how frequent users use the MovieBrain extension. For this purpose we analyzed user activity in relation to time. In Fig. 7.11 we show for every user on the X-axis, the days they were active on the extension. The users have been sorted chronologically in the order of their first installation of the extension. We note that the skewed shape of the graph is the result of various promotion campaigns for the MovieBrain extension. The plot e.g., shows how users 20 to 50 all installed the extension in a short period of time, which was the effect of promoting the extension on Reddit.

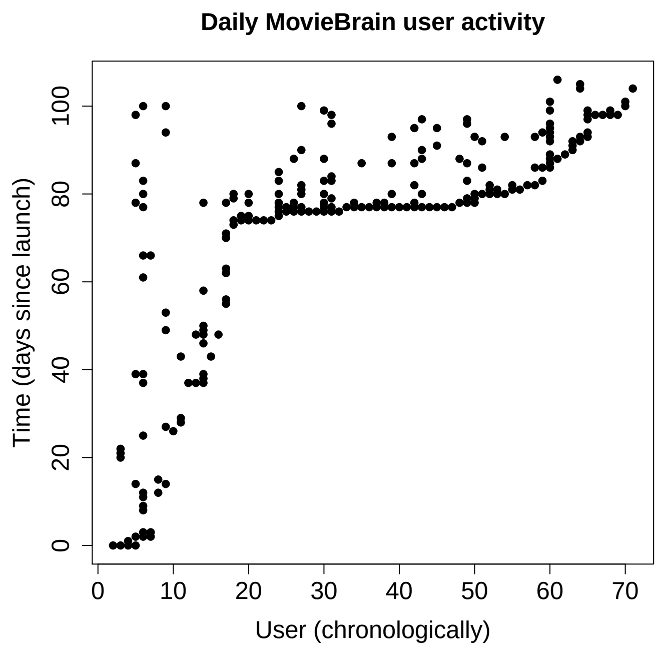


Figure 7.11: The daily MovieBrain user activity. For every user on the X-axis we plot their active days on the Y-axis. Users are sorted chronologically by the date they first installed the extension.

While 25 users have installed the extension and never used it again, we can see how most users actually do keep using the extension over time. Some users even keep actively using it up to 100 days after they installed it. We can thus hypothesize that the system succeeds in presenting sufficiently different results over time, or that it provides sufficient flexibility to adapt to changing contexts and environments.

Fig. 7.12 depicts the user activity in more detail, split up in different action categories. For every user on the X-axis all of their activities as logged by the API in the middleware are summed and displayed as a stacked barplot.

A number of observations can be made. First, interaction patterns are user-dependent and may differ significantly from one user to another.

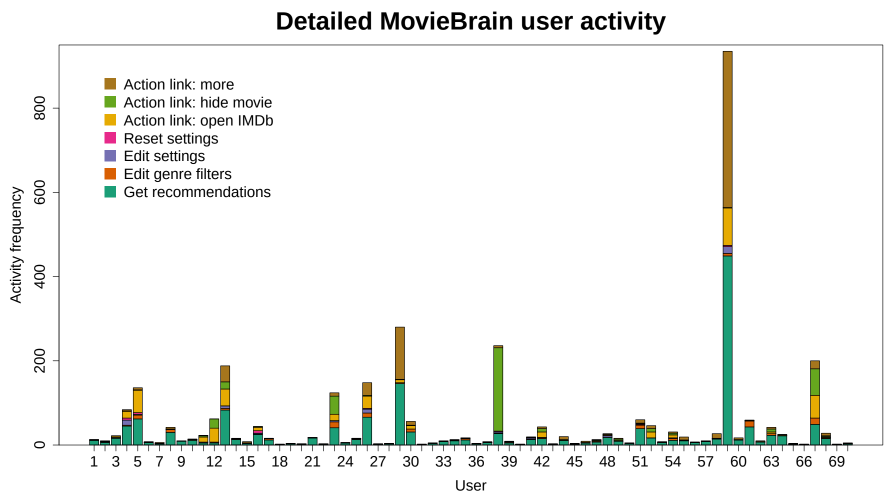


Figure 7.12: A stacked barplot illustrating the number of activities for each user as logged by the API in the middleware.

While some users have interacted with the extension over 200 times (one user even more than 900 times), other users show very limited activity. Second, the most observed interaction is the 'get recommendations' action, which makes sense as it represents the primary objective of the service. Most observed after that, are the click tracking actions, and then the settings and filter interactions. When we sum up all activities over all users (Fig. 7.13), we find this confirmed. Overall, 55% of all observed interactions are recommendation requests, 33% represents click tracking, and the remaining 12% of the activities are interactions with the settings and filters of the system.

7.7.4 API and System Stability

The stability of the system and API has been proven to be acceptable. Over the course of more than 100 days the system remained stable and available to its users. The only time the system was actually unavailable was due to an unforeseen power outage in the data center that powers the distributed brain processes. Users were made aware of the problem by means of social media and an appropriate notification message in the extension itself.

We do note that although the system is scalable and was built for high performance, the number of concurrent users was rather limited and so no

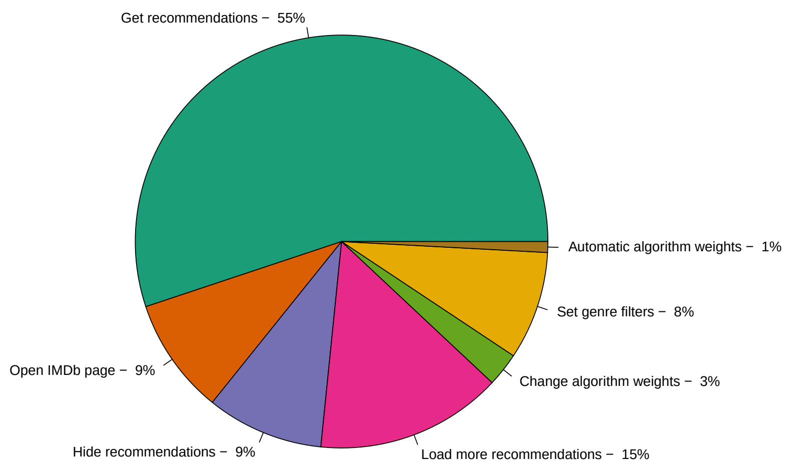


Figure 7.13: A comparison of the user activities captured during an analysis of the MovieBrain Chrome extension involving 70 users.

additional scaling measures such as load balancing, or allocating more calculation servers needed to be taken. Fig. 7.14 shows the measured API activity over the course of the 107 days online evaluation. In the figure on the left we grouped API requests per 5 minutes which reveals the detailed spikes in user activity during the evaluation period. The maximum spike is about 120 requests per 5 minutes, which can still be easily handled, even by our small webserver. In the figure on the right we show the same measurement, but with requests grouped per days.

7.7.5 Settings and Filters Interaction

The MovieBrain extension allows users to influence the recommendation process by changing settings and applying filters. These activities are also logged by the API and thus can be analyzed to provide us insight into typical user behavior. Fig. 7.15 shows how the genre filters are used by our users. Of the 70 users in total, about half of them actively used the genre filters feature and are shown on the X-axis. The Y-axis represents the number of *include* or *exclude* filters respectively with the positive and negative bars. Include filters are genres that the user wanted included for all recommendation results. So including the genre 'action' would result in recommending only movies that have at least the action genre contained in their metadata information. The exclude filters allow to model the opposite behavior, where all movies containing a given genre

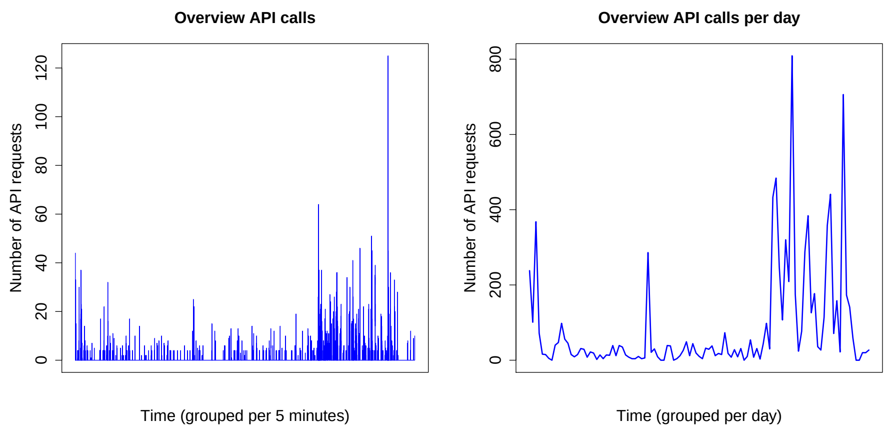


Figure 7.14: A detailed measurement of the API requests that were generated by the MovieBrain extension since the launch of the service. The figure on the left shows the requests grouped per 5 minute, the figure on the right shows the same analysis grouped per day.

can be excluded from the recommendations.

The results in the plot show how users tend to use the exclude feature more than the include. We hypothesize it is easier stating *what we don't like* as opposed to isolating *what we do like*. Furthermore the exclude filters are less restrictive[7] than the include filters especially when multiple filters are combined. We note that the few users in the plot with a high number of include filters, are most likely experimenting or playing with the filters feature since combining that much include filters will result in almost no recommended movies. The top-3 genres that were included are *comedy*, *adventure* and *action*, the top-3 genres that were excluded are *documentary*, *film-noir* and *music*.

Apart from the genre filters, users could also interact with settings that allowed to configure the individual influence of 4 recommendation algorithms. It is interesting to see how many users actually use this feature, and how different their manual settings are. In total 21 users (i.e., 30%) have overridden the automatic settings. For all these users, we show in Fig. 7.16 their selected manual settings for each of the 4 available algorithms which are values contained in the interval [0,100]. Note that the users were sorted according to their value on the Y-axis to

[7]Excluding a genre removes all recommended items from that genre, while including a genre causes **only** items of that genre to be recommended.

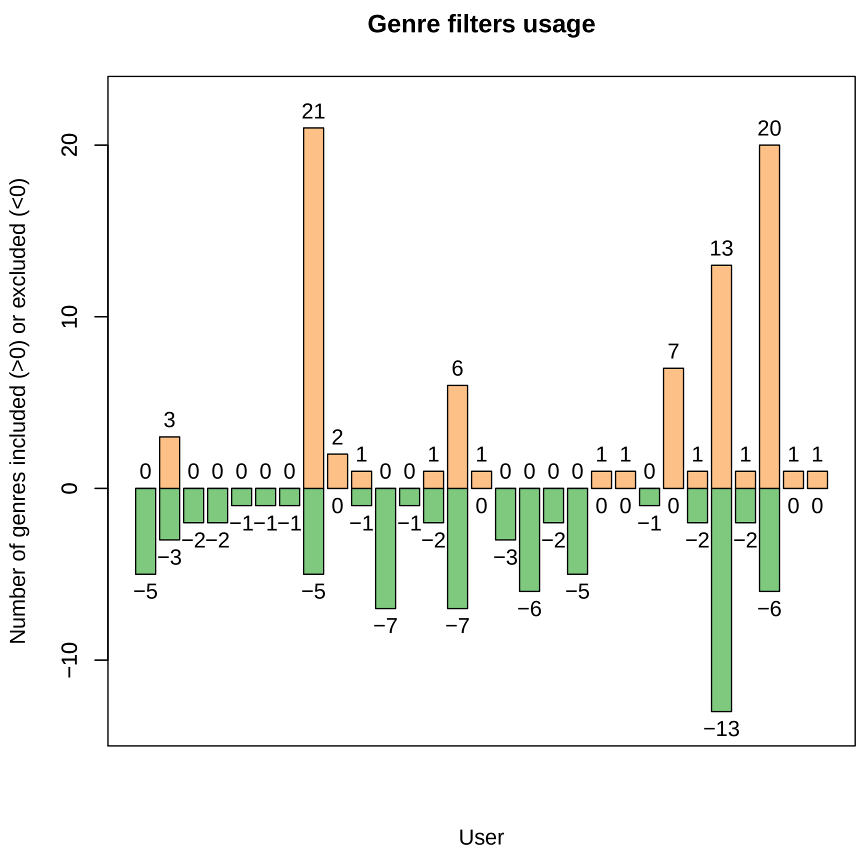


Figure 7.15: Bar plots illustrating how users engage with the genre filters feature. For every user that has actively used the genre filters, we show their total number of included genres on the positive bars and the number of excluded genres on the negative bars.

improve the clarity of the graph and therefore there is no correspondence of users on the X-axis (e.g., user 20 in the MatrixFactorization results plot may not be the same user 20 in the UBCF plot).

The most important observation that can be made from the settings plot is that almost all values are different for all users. So when users are allowed to configure their own settings, they will be different, which again supports our striving for user-specific approaches in recommender systems. The values for each algorithm range almost linearly between 0 and 100. This means that no single algorithm was selected by all users as 'the best', but rather every user showed a different appreciation for different algorithms.

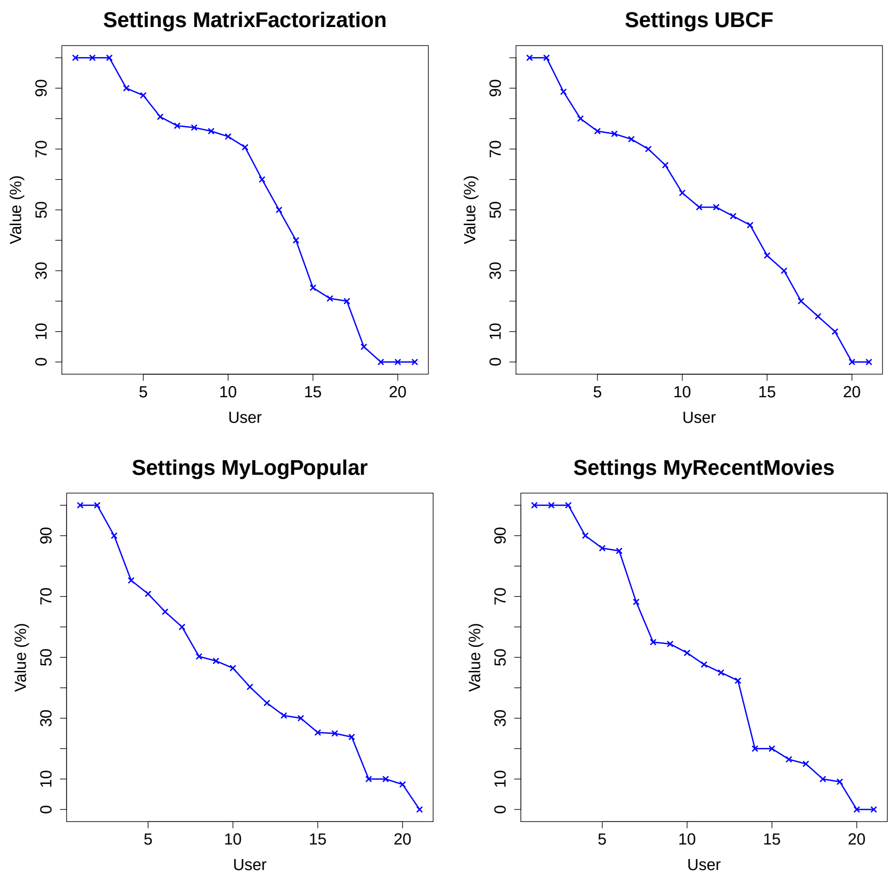


Figure 7.16: The individual manual settings that 21 users have set for 4 different recommendation algorithms. Note that to simplify each graph, the users on the X-axis have been sorted according to their value on the Y-axis and so they should not be compared over multiple plots.

So filters and settings are used by respectively 50% and 30% of all users of the MovieBrain extension, which is not overwhelmingly much. We however analyzed the difference between users that interacted with either the setting or filter features and users that did not, and found an interesting pattern. Fig. 7.17 shows again the user activity in relation to time, but we have split up the user activity of the different types of users (in the figure referred to as interactive and non-interactive users). The interactive users represent the users that have at least once used either the setting or genre filters (red squares in the figure). Interestingly, we found that the interactive users generate over 80% of all activity in the

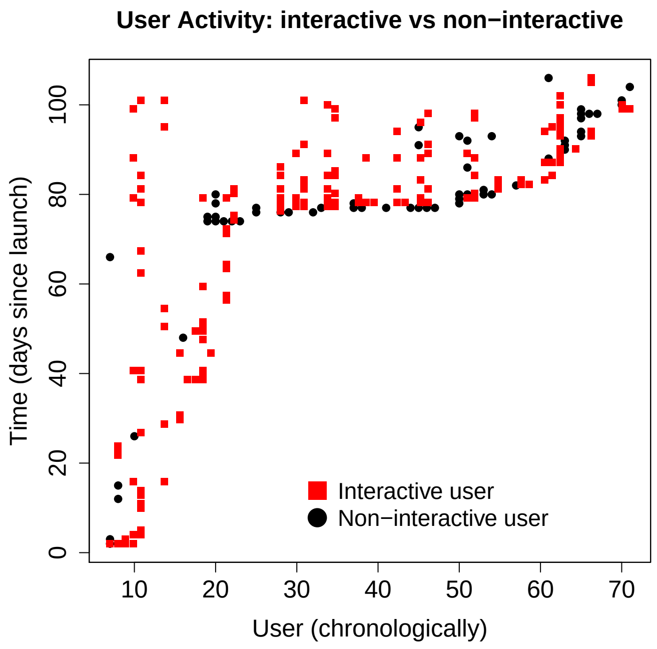


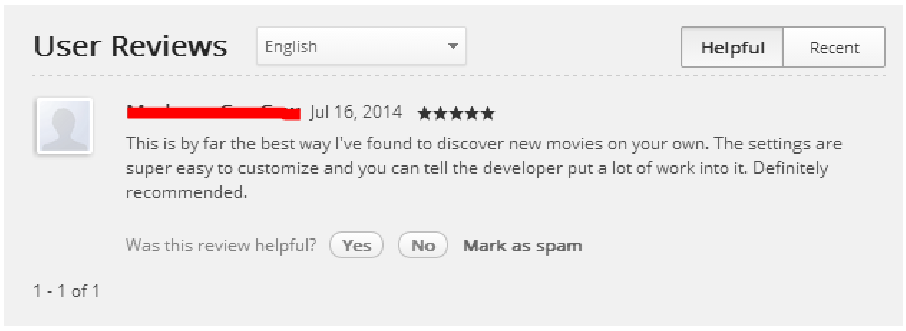
Figure 7.17: Daily MovieBrain activity where we distinguish between users who use the setting or filter features (interactive users) and users who do not (non-interactive users). The interactive users generated over 80% of all observed activity.

MovieBrain extension. We found the activities of *getting recommendations* and updating settings or filters to be highly correlated (Pearson correlation values of 0.97 and 0.94, both $p < 0.05$) and interactive users tend to use the MovieBrain service more often (as the figure shows). We must be careful not to confuse correlation with causality, but still the link between customer retention and recommendation customizability is a very interesting observation.

7.7.6 Subjective User Feedback

We conclude the results section of our MovieBrain analysis by discussing some of the more subjective user feedback we collected during the online evaluation experiment. While we logged a great number of implicit feedback indicators, users were also able to explicitly communicate their opinions and ideas through a number of online channels. The extension itself allowed feedback by means of an HTML feedback form, the extension could be reviewed on the Chrome web store, discussed on blogs and interacted with on social media (Facebook and Twitter).

Almost all explicit feedback was overall positive. Users liked the novel interaction features, were intrigued by the results and generally enjoyed the system. An example of a review provided by a user on the Chrome web store is displayed in Fig. 7.18.


User Reviews

English

Helpful | Recent

[redacted] Jul 16, 2014 ★★★★★

This is by far the best way I've found to discover new movies on your own. The settings are super easy to customize and you can tell the developer put a lot of work into it. Definitely recommended.

Was this review helpful? Yes No Mark as spam

1 - 1 of 1

Figure 7.18: A review provided by a user of the MovieBrain extension on the Chrome web store. Apart from some minor bug reporting, explicit user feedback was overall positive.

The only non-positive feedback that we did receive, concerned mostly minor bugs in the system such as already rated movies still appearing in the recommendations. All of which were easily fixed in a few consecutive updates[8] of the extension. One problem we found however to be quite harmful for the perceived recommendation quality of the system was the '*Bieber problem*'.

7.7.6.1 The Bieber Problem

We define the Bieber problem as the problem of avoiding the recommendation of items that are extremely 'popular' (in terms of attracting attention) but at the same time considered really bad by many. Recommending such an item may have disastrous effects on the perceived recommendation quality in general. In a sense, the Bieber problem could be considered a modern version of the Napoleon Dynamite problem [153] where the extreme bipolarity of users' opinions (they either love it or hate it) makes items difficult to recommend.

[8]The Google Chrome web browser has an auto-updating policy which made it very easy to push updates of the MovieBrain extension to users.

Our dataset included a movie titled "Justin Bieber's Believe" which was rated by a significant amount of people, but in most cases the rating was extremely low[9]. The difficulty for recommender systems is that the number of ratings for a movie usually correlates positively with its public opinion or popularity, and is therefore often integrated as a feature in recommendation calculations. We had to manually adjust the calculation model of our popularity-based recommendation algorithm after a number of users explicitly complained seeing the Justin Bieber movie in their recommendation results.

7.8 About Generalizability

We used the MovieBrain extension to expose our finished and working recommender system as a live online experiment. We note however that much of this work is in fact generalizable. In the introduction we defined a set of properties specific to our brain recommender, but most of these properties are very commonly found in typical (academic) recommender system implementations.

Infrastructure-wise we assumed a worst-case scenario (no budget, and no dedicated hardware) and detailed some strategies to overcome the limitations of using a shared and time-limited HPC infrastructure to power our online experiment. In many cases however, researchers will have access to dedicated hardware (or e.g., rented Amazon EC2 instances) which greatly simplifies the infrastructural challenges (a *multi-job* strategy would not be needed).

We focused on the movie domain because of the attractiveness of integrating the IMDb platform in combination with the availability of the MovieTweetings dataset. Previously however (see Section 2.7 in Chapter 2), we generalized the procedure for mining other domain rating datasets from Twitter (e.g., books, music, etc.). Thus, while we focused on the item domain of movies and IMDb, a similar approach could be taken towards the item domain of books and the Goodreads website, or the music domain and Pandora, or even the shopping domain and the Amazon website could be targeted as such. We have made the code of the MovieBrain Chrome extension publicly available[10] to inspire fellow academic researchers.

[9]The corresponding IMDb score for this movie is 1.5/10 while it is rated by over 15K users.

[10]`https://github.com/sidooms/MovieBrain`

7.9 Conclusion

We started this chapter at the point where most recommender systems research stops: the availability of a working recommender system. As the necessity of evaluating such systems by actual users (rather than offline calculated accuracy metrics) becomes more and more apparent, we show that, even with limited resources, our recommender system can be exposed to a potentially large online user base by integrating an existing website, dataset and web browser.

We introduced a basic 3-tier architecture for supporting our online evaluation experiment. In the back end a shared high-performance computing (HPC) infrastructure was found a suitable hardware environment provided some parallel strategies were implemented. A webserver in the middleware added security, data management and fine-grained control over the calculation process running in the back end. At the front end, we discussed the merits of a browser plugin and detailed our *MovieBrain* Chrome extension which integrated the popular IMDb platform and was powered by the MovieTweetings dataset.

We presented results collected from both an implicit and explicit perspective from 70 genuine users over 107 days. We provided a detailed analysis of how the extension was actually used and reflected on how user behavior at times showed to be significantly divergent. Observed user interaction with the customizability features such as genre filters and settings indicated how involved users are more active, more prone to use the service more often and ultimately succeed in taking control over their recommendation experience.

Chapter 8

Conclusions

8.1 Summary of Chapters

It all started with the information overload problem. The exponentially increasing rate at which content is generated everyday makes it extremely difficult for consumers to find what they want. Recommender systems' primary goal is to bridge the ever expanding gap between relevant content and users by automating content filtering through personalized recommendation approaches. Eagerly combating information overload for many years, has led to hundreds of recommendation algorithms being developed and thus the recommendation challenge has now shifted towards selecting the optimal strategy for a given situation, context and user. In this work we aimed to investigate approaches towards automatically combining multiple recommendation algorithms into personalized hybrid systems fine-tuned specifically for such scenarios. Each of the previous chapters offered a specific building block towards achieving this goal.

Chapter 2 focused on the collection of user data, which provided the fuel to power recommendation engines. Through a number of practical use cases and experiments on an existing website, we showed the value of user feedback and how its different types present different properties and challenges. Implicit feedback is very easy to collect, but contains less intrinsic information compared to explicitly provided feedback such as ratings. The latter is however much harder to obtain from users. We discussed the shortcomings of existing public rating datasets and presented our approach towards building our own public movie rating dataset *MovieTweetings* which we mined from IMDb ratings posted on

Twitter. The MovieTweetings dataset was benchmarked with a number of known recommendation algorithms and compared with another public movie rating dataset *MovieLens*. MovieTweetings showed surprisingly similar features, but was found to be more sparse due to it being a natural dataset and included more recent and current-day popular movies. We successfully generalized our data collection approach of mining ratings from Twitter to other domains such as books, music or video clips and even found those datasets to be overlapping, which may prove extremely valuable for future cross-domain recommender systems research.

Chapter 3 discussed the human-recommender interaction process from two perspectives: feedback mechanisms and recommendation presentation. Above all, users turned out to have often orthogonal opinions about the ideal interaction process. Some users want very active involvement, rate a lot of items and like to express their preferences as fine-grained as possible. Other users however, will not interact with the system at all, and expect a more passive, '*lean back*' experience. The well-known 5-star rating system came out best in multiple independent user studies, but ultimately each specific scenario will require its own customization to properly align both system requirements and the required cognitive effort from users. The duality of users also turned out to be important for user interface design. When recommendations are presented, users will want more or less information and may need proper explanations to go along with the recommended items. We built an interactive recommendation front end that integrated well-known recommendation algorithms and datasets, and allowed us to experiment with strategies to integrate user control and system transparency into a typical hybrid recommendation process. This front end was made publicly available.

Chapter 4 was all about performance. Complex recommendation algorithms typically require lots of computation power especially when processing large datasets or deployed in hybrid configurations. To reduce the computational burden of the recommendation process we analyzed its ability to map processing tasks on distributed and parallel hardware. While most distributed recommender systems rely on the commonly accepted *MapReduce* paradigm, we avoided such technological dependencies. Using a file-based approach, we split up a complex hybrid recommendation algorithm in multiple phases and restructured input and output data for each phase to optimize its distributed computing capability. Some efficiency was however lost because of the synchronization overhead after each phase and mid-computation disk access requirements. We then turned to a data parallelism paradigm to reduce the overhead of multiple phases and prevent load imbalance issues. We showed how a content-based recommendation algorithm could be turned into an embarrassingly parallel problem. By tweaking certain parameters of the distribution process we were able to optimally align processing tasks with underlying computation hardware which allowed us to surpass the parallel efficiency of current state-of-the-art distributed recommender systems. We finally focused on caching strategies for neighborhood-based recommendation algorithms and showed how storing only a limited amount of similarity values can significantly speed up the recommendation process in case of limited RAM scenarios.

Chapter 5 tackled the problem of automatically combining multiple recommendation algorithms into hybrid systems. We experimented with strategies for user-specific optimization (unique hybrid configuration per user) that also allowed a certain level of user control in the calculation process. We compared the hybrid recommendation approaches of *hybrid switching* and *weighted hybridization* and reported results on experiments that included up to 10 individual recommendation algorithms. Data showed how the switching approach was more sensitive and outperformed by the weighting approach. The latter proved more stable which allowed new algorithms to be included without fundamentally disrupting the recommendation experience of users.

Chapter 6 aimed to assess and improve the ability to get the offline optimization system (detailed in Chapter 5) out of the lab and meet online, real-world requirements as scalability, responsiveness, user control and system transparency. We introduced a client-server architecture and showed how it reduced the scalability of the complete system to that of its worst scalable component. The hybrid recommender system was made responsive by disconnecting a fast, real-time optimization phase from a slow training phase which could run in the background. The proposed architecture and its properties were experimentally validated in a number of experiments simulating e.g., weak and strong scaling scenarios.

Chapter 7 finally integrated all the pieces of our *PhD puzzle* in an online evaluation experiment. We built a self-learning and personalized hybrid recommender system and exposed it to actual users in the form of a Google Chrome extension called MovieBrain. We presented a 3-tier architecture where a computing back end was linked with a front end by means of a middleware layer which added data management, security, control and caching features. The back end was designed to run on a high-performance computing environment to guarantee optimal performance and scalability. We reported some specific measures to deal with the limitations of a shared computing environment and detailed how to implement our live optimization and background retraining approach as multiple processing jobs. The Chrome extension in the front end visualized movie recommendation results and allowed users to influence the (back-end) calculation process using dynamic settings and interaction features. User interaction data of 70 users was collected and analyzed over a period of 107 days. The users that engaged the control features, turned out to be more active and more prone to use the MovieBrain recommendation service over longer periods of time. When we analyzed the manual provided settings of the active users, user behavior turned out to be significantly different on an individual per-user basis. Both through implicit interaction and by means of explicit feedback, users indicated their enthusiasm towards our system and fully embraced the enhanced movie recommendation experience that was provided.

8.2 Final Conclusions

Our final conclusion is threefold. First, we conclude that we successfully managed to design, build and test a recommender system that was able to automatically compose itself into hybrid configurations tailored specifically to individual users.

Second, we underline the importance of a user-specific recommendation approach. While the base assumption of recommendation algorithms is usually that '*users are all the same*' and they can be grouped together in clusters of similar neighbors, throughout this work on multiple occasions we found users to be significantly different. Feedback experiments showed how there were active and passive users, users that want complex user interfaces and users that like simple interfaces. Our optimization experiments showed how optimal hybrid configurations per user also turned out to be unique, regardless of whether they were generated by the system or set manually by the users themselves. In the end every user is unique and therefore requires a unique approach.

Third, in order to keep striving towards an improved recommendation experience for users it is important that we shift focus from marginally improving mathematical-based recommendation algorithms towards increasing active user involvement in the recommendation process. While recommender systems were designed to outsource the manual content filtering process to algorithmic intelligence, ironically we believe it to be manual user involvement that will help push the current limit of recommender systems which is known as *the magic barrier*.

8.3 Summary of Contributions

During the course of this work, we have published 5 international journal publications, 8 conference or workshop papers, 1 demo and 1 newsletter (all first author). For a detailed overview of publications, we refer to the '*List of Publications*' in this work. Accompanying presentations and posters can be found on slideshare[1]. We furthermore present a short-list of contributions that we have made publicly available on Github[2] and esteem to be of value for researchers both in and outside the recommender systems domain.

1. **MovieTweetings** (Chapter 2, Section 2.4)
   A movie rating dataset composed out of IMDb ratings that are posted on Twitter. The dataset has the same format as the popular MovieLens dataset to allow easy integration into existing projects. An extended version of this dataset has been used for the *RecSys Challenge 2014*[3] which had over 200 registered participants.
2. **Twitter-ratings** (Chapter 2, Section 2.7)
   A collection of scripts to download and extract rating datasets from Twitter. Any ratings that are posted by means of a *social sharing* feature on a website in a pre-formatted way can be targeted. Examples for Goodreads (books), Pandora (music) and YouTube (video clips) are included.
3. **Recsys-frontend** (Chapter 3, Section 3.3.3)
   A configurable HTML-based front end for movie recommender systems. Movie recommendation results can be visualized and the MyMediaLite software library is integrated to allow experimentation with a list of well-known recommendation algorithms both individually or in a hybrid context.
4. **DistributedCB** (Chapter 4, Section 4.4)
   A parallel and distributed content-based recommendation algorithm. Data is pre-processed to optimize the distribution of work

[1]`http://www.slideshare.net/simondooms/presentations`
[2]`https://github.com/sidooms`
[3]`http://2014.recsyschallenge.com`

and reduce any load imbalance issues to a minimum. The content-based recommendation task is then reduced to an embarrassingly parallel problem that scales across multiple computing nodes and their parallel processing cores.

5. **MovieBrain** (Chapter 7, Section 7.6)
   A Google Chrome extension that integrates the IMDb website and provides customizable movie recommendations. Dynamic sliders and interactive drag and drop genre filters allow to influence the recommendation calculation process. Recommendation results are visualized in a 2-level thumbnail-based design and user interaction is logged by means of click tracking.

8.4 Glimpses of Future Work

Since research is a never ending quest for improvement, there are numerous areas where future work may focus on. We briefly touch some of the, in our opinion, most interesting directions.

The MovieBrain recommender system as designed in this work could be extended into a flexible recommendation service. The API in the middleware could be made publicly available and the back end moved to a more public hosting infrastructure such as Amazon. That way, the recommendation service could be easily integrated with a whole range of public movie information systems that may benefit from dynamic and interactive movie recommendations such as Kodi media center software (formerly known as XBMC), the Rotten Tomatoes website, smartphones apps, etc.

One of our main conclusions was that every user is unique and may therefore require a unique and personalized approach towards recommendation. Future work could focus on automatically personalizing parts of the recommendation process, such as feedback collection, recommendation calculation, and recommendation visualization. It may be possible to detect, and therefore dynamically adjust, what user interfaces work best for which users, or what feedback systems work best in certain contexts.

Our hybrid recommender system was able to automatically optimize hybrid configuration parameters and allowed users to manually update and change the settings. An interesting next step may be to suggest settings to users. Based on the collective data of many users, it may be possible to extrapolate trends and for example detect that younger users will prefer more recent movies over *movie classics* they may have never seen.

Last, but certainly not least, we mention the highly promising cross-domain recommendation opportunity made possible by our work on mining rating datasets from Twitter. While not further explored in this work, we have shown data to be available that supports cross-domain recommendation approaches. Users that like certain genres of movies may also have a similar taste in music, or maybe not. Future research could focus on user preferences and more specifically the transferability of those preferences to other item domains.

Appendix

Pseudocode Algorithms

Here we present the pseudocode for some of the algorithms that were discussed in this work. For each algorithm the relevant chapter and section are detailed.

Algorithm 2 Complete CB Recommendation (Chap 4, Sec 4.4.2)

```
1: for all items do
2:     cached_item_similarities ← {empty}
3:     for all users do
4:         if user has not rated item then
5:             calculate Rec(user, item)
6:
7: //Calculates the recommendation value for a (user, item) pair
8: procedure Rec(user, item)
9:     vote ← 0
10:    weights ← 0
11:    for all items $i_r$ rated by user do
12:        weight ← Simil(item, $i_r$)
13:        vote ← vote + (weight × rating for $i_r$)
14:        weights ← weights + weight
15:    return (vote / weights)
16:
17: //Calculates the similarity value of a ($item_1$, $item_2$) pair
18: procedure Simil($item_1$, $item_2$)
19:    if similarity in cached_item_similarities then
20:        return similarity
21:    else
22:        top    ←  The number of metadata item attributes that
                      $item_1$ and $item_2$ have in common (intersection)
23:        bottom ← The total number of metadata item attributes of
                      $item_1$ and $item_2$ (union)
24:        similarity ← top / bottom
25:        Add similarity to cached_item_similarities
26:        return similarity
```

Algorithm 3 UBCF Recommendation Calculation (Chap 4, Sec 4.5.1)

```
for all users do
    for all items do
        calculate Rec(user, item)

//Calculates the recommendation value for a (user, item) pair
procedure Rec(user, item)
    vote ← 0
    weights ← 0
    neighbors ← Neighbors_who_rated_item(user, item)
    for all neighbors do
        simil ← neighbor_similarity
        rating ← neighbor_rating
        vote ← vote + (simil × rating)
        weights ← weights + simil
    return (vote / weights)

procedure Neighbors_who_rated_item(user, item)
    for all neighbors who rated item do
        neighbor_similarity ← Pearson(user, neighbor)
    return The 20 most similar neighbors together with their
            original rating for item (neighbor_rating)
```

Algorithm 4 Best Switching Selection Strategy (Chap 5, Sec 5.5)

```
 1: $default\_algo \leftarrow$ Determine_default_algorithm()
 2: for all $users$ do
 3:     $algo\_selection\_list \leftarrow \{empty\}$
 4:     for all $algorithms$ do
 5:         $RMSE$, $variance \leftarrow$ evaluate_algorithm($user$, $algorithm$)
 6:         if $variance < VARIANCE\_THRESHOLD$ then
 7:             $algo\_selection\_list \leftarrow algo\_selection\_list + (RMSE,\ algorithm)$
 8:     if length of $algo\_selection\_list > 2$ then
 9:         $user\_algorithm \leftarrow$ select $algorithm$ with lowest $RMSE$
10:     else
11:         $user\_algorithm \leftarrow default\_algo$
12:
13: procedure Determine_default_algorithm
14:     $all\_algo\_values \leftarrow \{empty\}$
15:     for all $algorithms$ do
16:         $algo\_values \leftarrow \{empty\}$
17:         for all $users$ do
18:             $RMSE \leftarrow$ evaluate_algorithm($user$, $algorithm$)
19:             $algo\_values \leftarrow algo\_values + (algorithm,\ RMSE)$
20:         $RMSE \leftarrow$ average of $algo\_values$
21:         $all\_algo\_values \leftarrow all\_algo\_values + (algorithm,\ RMSE)$
22:     $algorithm \leftarrow$ select best from $all\_algo\_values$
23:     return $algorithm$
24:
25: procedure Evaluate_algorithm($user$, $algorithm$)
26:     $RMSE\_list \leftarrow \{empty\}$
27:     for all $subtest\_sets$ do
28:         $RMSE \leftarrow$ calculate $RMSE$ of actual ratings in $subtest\_set$ and predicted ratings by $algorithm$
29:         $RMSE\_list \leftarrow RMSE\_list + RMSE$
30:     $RMSE \leftarrow$ average of $RMSE\_list$
31:     $variance \leftarrow$ variance of $RMSE\_list$
32:     return $RMSE$, $variance$
```

Algorithm 5 Weighted Average Strategy (Chap 5, Sec 5.6)

```
1: $weights\_vectors \leftarrow$ random weight vectors, and all individual weight vectors
2: $weights\_vectors \leftarrow$ select the best from $weights\_vectors$
3: $iterations \leftarrow 0$
4: $current\_RMSE \leftarrow$ Evaluate_weights_vector($weights\_vector$)
5: //Start worse than current_RMSE
6: $previous\_RMSE \leftarrow current\_RMSE + 1$
7: while ($iterations < MAX\_ITERATIONS$) and ($previous\_RMSE > current\_RMSE$) do
8:     $previous\_RMSE \leftarrow current\_RMSE$
9:     $weight\_vector \leftarrow$ Optimize_weights_vector($weights\_vector$)
10:    $current\_RMSE \leftarrow$ Evaluate_weights_vector($weights\_vector$)
11:    $iterations \leftarrow iterations + 1$
12:
13: procedure OPTIMIZE_WEIGHTS_VECTOR($weights\_vector$)
14:    $old\_RMSE \leftarrow$ Evaluate_weights_vector($weights\_vector$)
15:    for all $weights$ in $weights\_vector$ do
16:        $new\_RMSE$, $new\_weight \leftarrow$ upwards binary search for improved weight
17:        if $new\_RMSE < old\_RMSE$ then
18:            return $weights\_vector$ with $new\_weight$
19:        $new\_RMSE$, $new\_weight \leftarrow$ downwards binary search for improved weight
20:        if $new\_RMSE < old\_RMSE$ then
21:            return $weights\_vector$ with $new\_weight$
22:    return $weights\_vector$
23:
24: procedure EVALUATE_WEIGHTS_VECTOR($weights\_vector$)
25:    $RMSE\_list \leftarrow \{empty\}$
26:    for all $subtest\_sets$ do
27:        $RMSE \leftarrow$ calculate $RMSE$ of actual ratings in $subtest\_set$ and predicted ratings by calculating the weighted average score with $weights\_vector$
28:        $RMSE\_list \leftarrow RMSE\_list + RMSE$
29:    $RMSE \leftarrow$ average of $RMSE\_list$
30:    $variance \leftarrow$ variance of $RMSE\_list$
31:    if $variance > VARIANCE\_THRESHOLD$ then
32:        return worst case $RMSE$, so this $weights\_vector$ will be discarded
33:    else
34:        return $RMSE$, $variance$
```

Algorithm 6 Weight Vector Evaluation (Chap 6, Sec 6.5.1)

```
1: //Calculates the fitness i.e., RMSE value for a weight vector
2: $RMSE \leftarrow 0$
3: $count \leftarrow 0$
4: for all ratings in the test fold dataset of the user do
5:     $prediction \leftarrow$ Predict(user, item, weights_vector)
6:     $error \leftarrow prediction - rating$
7:     $RMSE \leftarrow RMSE + (error \times error)$
8:     $count \leftarrow counts + 1$
9: $RMSE \leftarrow$ SQRT($RMSE$ / $count$)
10: return RMSE
11:
12: //Calculates the weighted prediction using the weight_vector
13: procedure PREDICT(user, item, weights_vector)
14:     $numerator \leftarrow 0$
15:     $denominator \leftarrow 0$
16:     for all algorithms do
17:         $weight \leftarrow$ get weight from weights_vector
18:         $prediction \leftarrow$ algorithm.get_recommendation(user, item)
19:         $numerator \leftarrow prediction \times weight$
20:         $denominator \leftarrow denominator + weight$
21:     weighted_prediction_value $\leftarrow$ numerator/denominator
22:     return weighted_prediction_value
```

References

[1] Goldberg D, Nichols D, Oki BM, Terry D. Using Collaborative Filtering to Weave an Information Tapestry. *Commun ACM*. 1992; 35(12):61–70.

[2] Resnick P, Iacovou N, Suchak M, Bergstrom P, Riedl J. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In: *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94. New York, NY, USA: ACM. 1994; pp. 175–186.

[3] Lops P, de Gemmis M, Semeraro G. Content-based Recommender Systems: State of the Art and Trends. In: *Recommender Systems Handbook*, pp. 73–105. 2011;.

[4] Suksom N, Buranarach M, Thein YM, Supnithi T, Netisopakul P. A knowledge–based framework for development of personalized food recommender system. In: *The Fifth International Conference on Knowledge, Information and Creativity Support Systems*. 2010; pp. 25–27.

[5] Tung HW, Soo VW. A personalized restaurant recommender agent for mobile e-service. In: *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*. IEEE. 2004; pp. 259–262.

[6] Berkovsky S, Freyne J. Group-based recipe recommendations: analysis of data aggregation strategies. In: *Proceedings of the fourth ACM conference on Recommender systems*. ACM. 2010; pp. 111–118.

[7] Mooney RJ, Roy L. Content-based book recommending using learning for text categorization. In: *Proceedings of the fifth ACM conference on Digital libraries*. ACM. 2000; pp. 195–204.

[8] Gorgoglione M, Panniello U, Tuzhilin A. The effect of context-aware recommendations on customer purchasing behavior and trust. In: *Proceedings of the fifth ACM conference on Recommender systems.* ACM. 2011; pp. 85–92.

[9] Koenigstein N, Dror G, Koren Y. Yahoo! music recommendations: modeling music ratings with temporal dynamics and item taxonomy. In: *Proceedings of the fifth ACM conference on Recommender systems.* ACM. 2011; pp. 165–172.

[10] Jameson A. More than the sum of its members: challenges for group recommender systems. In: *Proceedings of the working conference on Advanced visual interfaces.* ACM. 2004; pp. 48–54.

[11] Lu HY, Lu J, Al-Hassan M. A Framework for Delivering Personalized E-Government Tourism Services. 2010;.

[12] Castagnos S, Jones N, Pu P. Eye-tracking product recommenders' usage. In: *Proceedings of the fourth ACM conference on Recommender systems.* ACM. 2010; pp. 29–36.

[13] Symeonidis P, Nanopoulos A, Manolopoulos Y. MoviExplain: a recommender system with explanations. In: *Proceedings of the third ACM conference on Recommender systems.* ACM. 2009; pp. 317–320.

[14] Lee DH. Pittcult: trust-based cultural event recommender. In: *Proceedings of the 2008 ACM conference on Recommender systems.* ACM. 2008; pp. 311–314.

[15] Klamma R, Cuong PM, Cao Y. You never walk alone: Recommending academic events based on social network analysis. In: *Complex Sciences*, pp. 657–670. Springer. 2009;.

[16] Gipp B, Beel J, Hentschel C. Scienstein: A research paper recommender system. In: *International Conference on Emerging Trends in Computing.* 2009; pp. 309–315.

[17] Tayebi MA, Jamali M, Ester M, Glässer U, Frank R. Crimewalker: a recommendation model for suspect investigation. In: *Proceedings of the fifth ACM conference on Recommender systems.* ACM. 2011; pp. 173–180.

[18] Paparrizos I, Cambazoglu BB, Gionis A. Machine learned job recommendation. In: *Proceedings of the fifth ACM Conference on Recommender Systems*. ACM. 2011; pp. 325–328.

[19] Wu X, Zhang Y, Guo J, Li J. Web video recommendation and long tail discovering. In: *Multimedia and Expo, 2008 IEEE International Conference on*. IEEE. 2008; pp. 369–372.

[20] Yu Z, Zhou X. TV3P: an adaptive assistant for personalized TV. *Consumer Electronics, IEEE Transactions on*. 2004;50(1):393–399.

[21] Sales T, Sales L, Pereira M, Almeida H, Perkusich A, Gorgônio K, de Sales M. Towards the upnp-up: Enabling user profile to support customized services in upnp networks. In: *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM'08. The Second International Conference on*. IEEE. 2008; pp. 206–211.

[22] Liu J, Dolan P, Pedersen ER. Personalized news recommendation based on click behavior. In: *Proceedings of the 15th international conference on Intelligent user interfaces*. ACM. 2010; pp. 31–40.

[23] Schafer JB, Konstan J, Riedl J. Recommender systems in e-commerce. In: *Proceedings of the 1st ACM conference on Electronic commerce*. ACM. 1999; pp. 158–166.

[24] Said A, Tikk D, Shi Y, Larson M, Stumpf K, Cremonesi P. Recommender systems evaluation: A 3D benchmark. In: *ACM RecSys 2012 Workshop on Recommendation Utility Evaluation: Beyond RMSE, Dublin, Ireland*. 2012; pp. 21–23.

[25] Herlocker JL, Konstan JA, Terveen LG, Riedl JT. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*. 2004;22(1):5–53.

[26] Shani G. Tutorial on Evaluating Recommender Systems. In: *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10. New York, NY, USA: ACM. 2010; pp. 1–1.

[27] Shani G, Gunawardana A. Evaluating recommendation systems. In: *Recommender systems handbook*, pp. 257–297. Springer. 2011;.

[28] Hayes C, Cunningham P. An on-line evaluation framework for recommender systems. *Tech. rep.*, Trinity College Dublin, Department of Computer Science. 2002.

[29] Amatriain X, Pujol JM, Tintarev N, Oliver N. Rate it again: increasing recommendation accuracy by user re-rating. In: *Proceedings of the third ACM conference on Recommender systems*. ACM. 2009; pp. 173–180.

[30] Amatriain X, Pujol JM, Oliver N. I like it... i like it not: Evaluating user ratings noise in recommender systems. In: *User Modeling, Adaptation, and Personalization*, pp. 247–258. Springer. 2009;.

[31] Kelly D, Teevan J. Implicit feedback for inferring user preference: a bibliography. In: *ACM SIGIR Forum*, vol. 37. ACM. 2003; pp. 18–28.

[32] Hu Y, Koren Y, Volinsky C. Collaborative filtering for implicit feedback datasets. In: *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. IEEE. 2008; pp. 263–272.

[33] Jawaheer G, Szomszor M, Kostkova P. Comparison of implicit and explicit feedback from an online music recommendation service. In: *HetRec '10: Proceedings of the 1st International Workshop on Information Heterogeneity and Fusion in Recommender Systems*. New York, NY, USA: ACM. 2010; pp. 47–51.

[34] Gantner Z, Rendle S, Freudenthaler C, Schmidt-Thieme L. MyMediaLite: A free recommender system library. In: *Proceedings of the fifth ACM conference on Recommender systems*. ACM. 2011; pp. 305–308.

[35] Ekstrand MD, Ludwig M, Konstan JA, Riedl JT. Rethinking the recommender research ecosystem: reproducibility, openness, and LensKit. In: *Proceedings of the fifth ACM conference on Recommender systems*. ACM. 2011; pp. 133–140.

[36] Said A, Tikk D, Cremonesi P. Benchmarking. In: *Recommendation Systems in Software Engineering*, edited by Robillard MP, Maalej W, Walker RJ, Zimmermann T, pp. 275–300. Springer Berlin Heidelberg. 2014;.

[37] Burke R. Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*. 2002;12:331–370.

[38] Adomavicius G, Tuzhilin A. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible

extensions. *Knowledge and Data Engineering, IEEE Transactions on*. 2005;17(6):734–749.

[39] Bellogín A. Performance prediction and evaluation in Recommender Systems: An Information Retrieval perspective. Ph.D. thesis, Universidad Autonoma de Madrid. 2012.

[40] Jannach D, Zanker M, Felfernig A, Friedrich G. *Recommender systems: an introduction*. Cambridge University Press. 2010.

[41] Ekstrand M, Riedl J. When recommenders fail: predicting recommender failure for algorithm selection and combination. In: *Proceedings of the sixth ACM conference on Recommender systems*. ACM. 2012; pp. 233–236.

[42] Kille B, Albayrak S. Modeling Difficulty in Recommender Systems. In: *Workshop on Recommendation Utility Evaluation: Beyond RMSE (RUE 2011)*. 2012; p. 30.

[43] Srinivas KK, Gutta S, Schaffer D, Martino J, Zimmerman J. A Multi-Agent TV Recommender. In: *proceedings of the UM 2001 workshop 'Personalization in Future TV'*. 2001; .

[44] Herlocker JL, Konstan JA, Borchers A, Riedl J. An algorithmic framework for performing collaborative filtering. In: *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 1999; pp. 230–237.

[45] Bobadilla J, Serradilla F, Bernal J. A new collaborative filtering metric that improves the behavior of recommender systems. *Knowledge-Based Systems*. 2010;23(6):520–528.

[46] Peralta V. Extraction and Integration of MovieLens and IMDb Data. *Tech. rep.*, Technical Report, Laboratoire PRiSM, Université de Versailles, France. 2007.

[47] Töscher A, Jahrer M, Bell RM. The bigchaos solution to the netflix grand prize. *Netflix prize documentation*. 2009;.

[48] Piotte M, Chabbert M. The pragmatic theory solution to the netflix grand prize. *Netflix prize documentation*. 2009;.

[49] Koren Y. The bellkor solution to the netflix grand prize. *Netflix prize documentation*. 2009;.

[50] de Castro PA, de França FO, Ferreira HM, Von Zuben FJ. Applying biclustering to perform collaborative filtering. In: *Intelligent Systems Design and Applications, 2007. ISDA 2007. Seventh International Conference on.* IEEE. 2007; pp. 421–426.

[51] Hurrell E, Smeaton AF. A conversational collaborative filtering approach to recommendation. In: *Advances in Visual Informatics*, pp. 13–24. Springer. 2013;.

[52] Said A, Fields B, Jain BJ, Albayrak S. User-centric evaluation of a k-furthest neighbor collaborative filtering recommender algorithm. In: *Proceedings of the 2013 conference on Computer supported cooperative work.* ACM. 2013; pp. 1399–1408.

[53] Dooms S, De Pessemier T, Martens L. A user-centric evaluation of recommender algorithms for an event recommendation system. In: *RecSys 2011 Workshop on Human Decision Making in Recommender Systems (Decisions@ RecSys' 11) and User-Centric Evaluation of Recommender Systems and Their Interfaces-2 (UCERSTI 2) affiliated with the 5th ACM Conference on Recommender Systems (RecSys 2011).* Ghent University, Department of Information technology. 2011; pp. 67–73.

[54] Yi J, Nasukawa T, Bunescu R, Niblack W. Sentiment analyzer: Extracting sentiments about a given topic using natural language processing techniques. In: *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on.* IEEE. 2003; pp. 427–434.

[55] Morstatter F, Pfeffer J, Liu H, Carley KM. Is the Sample Good Enough? Comparing Data from Twitter's Streaming API with Twitter's Firehose. In: *ICWSM.* 2013; .

[56] Gena C, Brogi R, Cena F, Vernero F. The impact of rating scales on user's rating behavior. In: *User Modeling, Adaption and Personalization*, pp. 123–134. Springer. 2011;.

[57] Bellogín A, de Vries A, He J. Artist popularity: do web and social music services agree. In: *Int. Conf. on Weblogs and Social Media (ICWSM), Boston.* 2013; .

[58] Baeza-Yates RA, Ribeiro-Neto BA. *Modern Information Retrieval - the concepts and technology behind search, Second edition.* Pearson Education Ltd., Harlow, England. 2011.

[59] Marlin BM, Zemel RS. Collaborative prediction and ranking with non-random missing data. In: *Proceedings of the third ACM conference on Recommender systems*. ACM. 2009; pp. 5–12.

[60] Cremonesi P, Koren Y, Turrin R. Performance of Recommender Algorithms on Top-n Recommendation Tasks. In: *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10. New York, NY, USA: ACM. 2010; pp. 39–46.

[61] Koren Y. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2008; pp. 426–434.

[62] Campos PG, Díez F, Cantador I. Time-Aware Recommender Systems: A Comprehensive Survey and Analysis of Existing Evaluation Protocols. *User Modeling and User-adapted Interaction*. 2014; .

[63] McNee SM, Riedl J, Konstan JA. Being Accurate is Not Enough: How Accuracy Metrics Have Hurt Recommender Systems. In: *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06. New York, NY, USA: ACM. 2006; pp. 1097–1101.

[64] Bellogín A, Castells P, Cantador I. Precision-oriented evaluation of recommender systems: an algorithmic comparison. In: *RecSys*. 2011; pp. 333–336.

[65] Koren Y, Bell RM, Volinsky C. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer*. 2009;42(8):30–37.

[66] Cremonesi P, Garzotto F, Negro S, Papadopoulos A, Turrin R. Comparative Evaluation of Recommender System Quality. In: *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11. New York, NY, USA: ACM. 2011; pp. 1927–1932.

[67] De Pessemier T, Dooms S, Deryckere T, Martens L. Time Dependency of Data Quality for Collaborative Filtering Algorithms. In: *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10. New York, NY, USA: ACM. 2010; pp. 281–284.

[68] Campos PG, Díez F, Sánchez-Montañés M. Towards a More Realistic Evaluation: Testing the Ability to Predict Future Tastes

of Matrix Factorization-based Recommenders. In: *Proceedings of the Fifth ACM Conference on Recommender Systems*, RecSys '11. New York, NY, USA: ACM. 2011; pp. 309–312.

[69] Said A, Wetzker R, Umbrath W, Hennig L. A hybrid PLSA approach for warmer cold start in folksonomy recommendation. In: *Proceedings of the RecSys'09 Workshop on Recommender Systems & The Social Web.* CEUR-WS Vol. 532. 2009; pp. 87–90.

[70] Li B, Yang Q, Xue X. Can Movies and Books Collaborate? Cross-Domain Collaborative Filtering for Sparsity Reduction. In: *IJCAI*, vol. 9. 2009; pp. 2052–2057.

[71] Pan W, Xiang EW, Liu NN, Yang Q. Transfer Learning in Collaborative Filtering for Sparsity Reduction. In: *AAAI*, vol. 10. 2010; pp. 230–235.

[72] Fernández-Tobías I, Cantador I, Kaminskas M, Ricci F. Cross-domain recommender systems: A survey of the state of the art. In: *Proc. 2nd Spanish conf. on Information Retrieval. CERI.* 2012; .

[73] Goldberg K, Roeder T, Gupta D, Perkins C. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval.* 2001;4(2):133–151.

[74] Cosley D, Lam SK, Albert I, Konstan JA, Riedl J. Is seeing believing?: how recommender system interfaces affect users' opinions. In: *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM. 2003; pp. 585–592.

[75] Hill W, Stead L, Rosenstein M, Furnas G. Recommending and evaluating choices in a virtual community of use. In: *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM Press/Addison-Wesley Publishing Co. 1995; pp. 194–201.

[76] Preston CC, Colman AM. Optimal number of response categories in rating scales: reliability, validity, discriminating power, and respondent preferences. *Acta psychologica.* 2000;104(1):1–15.

[77] Harper FM, Li X, Chen Y, Konstan JA. An economic model of user rating in an online recommender system. In: *User Modeling 2005*, pp. 307–316. Springer. 2005;.

[78] Sparling EI, Sen S. Rating: how difficult is it? In: *Proceedings of the fifth ACM conference on Recommender systems*. ACM. 2011; pp. 149–156.

[79] Bostandjiev S, O'Donovan J, Höllerer T. TasteWeights: a visual interactive hybrid recommender system. In: *Proceedings of the sixth ACM conference on Recommender systems*. ACM. 2012; pp. 35–42.

[80] Gretarsson B, O'Donovan J, Bostandjiev S, Hall C, Höllerer T. Smallworlds: Visualizing social recommendations. In: *Computer Graphics Forum*, vol. 29. Wiley Online Library. 2010; pp. 833–842.

[81] Herlocker JL, Konstan JA, Riedl J. Explaining collaborative filtering recommendations. In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM. 2000; pp. 241–250.

[82] Bilgic M, Mooney RJ. Explaining recommendations: Satisfaction vs. promotion. In: *Beyond Personalization Workshop, IUI*, vol. 5. 2005; .

[83] Tintarev N. Explanations of recommendations. In: *Proceedings of the 2007 ACM conference on Recommender systems*. ACM. 2007; pp. 203–206.

[84] Chen Y, Pu P. CoFeel: Using Emotions for Social Interaction in Group Recommender Systems. In: *First International Workshop on Recommendation Technologies for Lifestyle Change (LIFESTYLE 2012)*. 2012; p. 48.

[85] Devendorf L, O'Donovan J, Höllerer T. TopicLens: An Interactive Recommender System based on Topical and Social Connections. In: *First International Workshop on Recommendation Technologies for Lifestyle Change (LIFESTYLE 2012)*. 2012; p. 41.

[86] Vlachos M, Svonava D. Graph Embeddings for Movie Visualization and Recommendation. In: *First International Workshop on Recommendation Technologies for Lifestyle Change (LIFESTYLE 2012)*. 2012; p. 56.

[87] O'Donovan J, Smyth B, Gretarsson B, Bostandjiev S, Höllerer T. PeerChooser: visual interactive recommendation. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2008; pp. 1085–1088.

[88] Chen L, Pu P. Critiquing-based recommenders: survey and emerging trends. *User Modeling and User-Adapted Interaction.* 2012; 22(1-2):125–150.

[89] Dooms S, De Pessemier T, Verslype D, Nelis J, De Meulenaere J, Van den Broeck W, Martens L, Develder C. Omus: an optimized multimedia service for the home environment. *Multimedia Tools and Applications.* 2013;pp. 1–31.

[90] McCarthy JF, Anagnost TD. MusicFX: an arbiter of group preferences for computer supported collaborative workouts. In: *Proc. ACM Conf. Computer supported cooperative work*, CSCW '98. New York, NY, USA: ACM. 1998; pp. 363–372.

[91] Keckler S, Olukotun K, Hofstee H. *Multicore processors and systems.* Springer. 2009.

[92] Sarwar B, Karypis G, Konstan J, Riedl J. Application of dimensionality reduction in recommender system-a case study. *Tech. rep.*, DTIC Document. 2000.

[93] Sarwar B, Karypis G, Konstan J, Riedl J. Incremental singular value decomposition algorithms for highly scalable recommender systems. In: *5th Int. Conf. Computer and Information Science.* Citeseer. 2002; pp. 27–28.

[94] Takács G, Pilászy I, Németh B, Tikk D. Scalable collaborative filtering approaches for large recommender systems. *J of Machine Learning Research.* 2009;10:623–656.

[95] Anand SS, Mobasher B. Intelligent techniques for web personalization. In: *Proc. Int. Conf. Intelligent Techniques for Web Personalization.* Springer-Verlag. 2003; pp. 1–36.

[96] Herlocker J, Konstan JA, Riedl J. An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. *Information retrieval.* 2002;5(4):287–310.

[97] Hager G, Wellein G. *Introduction to High Performance Computing for Scientists and Engineers.* Boca Raton, FL, USA: CRC Press, Inc., 1st ed. 2010.

[98] Han P, Xie B, Yang F, Shen R. A scalable P2P recommender system based on distributed collaborative filtering. *Expert Systems with Applications.* 2004;27(2):203 – 210.

[99] Xie B, Han P, Yang F, Shen RM, Zeng HJ, Chen Z. DCFLA: A distributed collaborative-filtering neighbor-locating algorithm. *Inf Sci.* 2007;177(6):1349–1363.

[100] Lämmel R. Google's MapReduce programming model-Revisited. *Science of Computer Programming.* 2008;70(1):1–30.

[101] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM.* 2008;51(1):107–113.

[102] Zhao Z, Shang M. User-based collaborative-filtering recommendation algorithms on hadoop. In: *3rd Int. Conf. Knowledge Discovery and Data Mining (WKDD'10).* IEEE. 2010; pp. 478–481.

[103] Schelter S, Boden C, Markl V. Scalable similarity-based neighborhood methods with MapReduce. In: *Proc. 6th ACM Conf. on Recommender Systems.* ACM. 2012; pp. 163–170.

[104] Jiang J, Lu J, Zhang G, Long G. Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop. In: *Services (SERVICES), 2011 IEEE World Congress on.* IEEE. 2011; pp. 490–497.

[105] Lemire D, McGrath S. Implementing a Rating-Based Item-to-Item Recommender System in PHP/SQL. *Tech. Rep. D-01*, Ondelette.com. 2005.

[106] Woerndl W, Schueller C, Wojtech R. A Hybrid Recommender System for Context-aware Recommendations of Mobile Applications. In: *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop.* 2007; pp. 871–878.

[107] Davidson J, Liebald B, Liu J, Nandy P, Van Vleet T, Gargi U, Gupta S, He Y, Lambert M, Livingston B, Sampath D. The YouTube video recommendation system. In: *Proceedings of the fourth ACM conference on Recommender systems*, RecSys '10. 2010; pp. 293–296.

[108] Cornelis C, Guo X, Lu J, Zhang G. A fuzzy relational approach to event recommendation. In: *Proceedings of the Indian International Conference on Artificial Intelligence.* 2005; .

[109] Amatriain X, Jaimes A, Oliver N, Pujol JM. Data Mining Methods for Recommender Systems. In: *Recommender Systems Handbook*, pp. 39–71. 2011;.

[110] Cornelis C, Lu J, Guo X, Zhang G. One-and-only item recommendation with fuzzy logic techniques. *Information Sciences.* 2007; 177(22):4906–4921.

[111] Das A, Datar M, Garg A, Rajaram S. Google news personalization: scalable online collaborative filtering. In: *Proc. 16th Int. Conf. World Wide Web.* ACM. 2007; pp. 271–280.

[112] De Pessemier T, Vanhecke K, Dooms S, Martens L. Content-based recommendation algorithms on the Hadoop mapreduce framework. In: *Proc. 7th Int. Conf. Web Information Systems and Technologies.* Ghent University, Department of Information technology. 2011; .

[113] Hochbaum DS, Shmoys DB. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J ACM.* 1987;34(1):144–162.

[114] Bilolikar V, Jain K, Sharma M. An Annealed Genetic Algorithm for Multi Mode Resource Constrained Project Scheduling Problem. *Int J of Computer Applications.* 2012;60(1):36–42.

[115] Gomez-Gasquet P, Segura-Andres R, Franco D, Andres C. A makespan minimization in an m-stage flow shop lot streaming with sequence dependent setup times: MILP model and experimental approach. In: *6th Int. Conf. Industrial Engineering and Industrial Management.* 2012; pp. 332–339.

[116] Liu M, Zheng F, Wang S, Xu Y. Approximation algorithms for parallel machine scheduling with linear deterioration. *Theoretical Computer Science.* 2012;.

[117] Ahmadizar F. A new ant colony algorithm for makespan minimization in permutation flow shops. *Computers & Industrial Engineering.* 2012;.

[118] Amdahl G. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proc. spring joint computer Conf.* ACM. 1967; pp. 483–485.

[119] Qasim U. Active Caching For Recommender Systems. Ph.D. thesis, New Jersey Institute of Technology, New Jersey. 2011.

[120] Qasim U, Oria V, fang Brook Wu Y, Houle ME, Özsu MT. A partial-order based active cache for recommender systems. In: *Proc. ACM Conf. Recommender systems (RecSys 2009).* 2009; pp. 209–212.

[121] Seth S, Kaiser G. Towards using Cached Data Mining for Large Scale Recommender Systems. In: *Proc. Conf. Data Engineering and Internet Technology (DEIT 2011).* 2011; .

[122] Lemire D, Maclachlan A. Slope one predictors for online rating-based collaborative filtering. *Society for Industrial Mathematics.* 2005;5:471–480.

[123] Adomavicius G, Tuzhilin A. Context-aware recommender systems. In: *Recommender systems handbook*, pp. 217–253. Springer. 2011;.

[124] Hussein T, Linder T, Gaulke W, Ziegler J. Hybreed: A software framework for developing context-aware hybrid recommender systems. *User Modeling and User-Adapted Interaction.* 2012;pp. 1–54.

[125] Song Y, Zhang L, Giles CL. Automatic tag recommendation algorithms for social recommender systems. *ACM Transactions on the Web (TWEB).* 2011;5(1):4.

[126] Aksel F, Birturk A. An Adaptive Hybrid Recommender System that Learns Domain Dynamics. In: *International Workshop on Handling Concept Drift in Adaptive Information Systems: Importance, Challenges and Solutions (HaCDAIS-2010) at the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases.* 2010; p. 49.

[127] Balabanović M, Shoham Y. Fab: content-based, collaborative recommendation. *Communications of the ACM.* 1997;40(3):66–72.

[128] Pazzani MJ. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review.* 1999;13(5-6):393–408.

[129] Han EHS, Karypis G. Feature-based recommendation system. In: *Proceedings of the 14th ACM international conference on Information and knowledge management.* ACM. 2005; pp. 446–452.

[130] Salehi M, Pourzaferani M, Razavi SA. Hybrid attribute-based recommender system for learning material using genetic algorithm

and a multidimensional information model. *Egyptian Informatics Journal.* 2013;.

[131] Polikar R. Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE.* 2006;6(3):21–45.

[132] Sill J, Takács G, Mackey L, Lin D. Feature-Weighted Linear Stacking. *CoRR.* 2009;abs/0911.0460.

[133] Wolpert DH. Stacked generalization. *Neural networks.* 1992; 5(2):241–259.

[134] Bao X, Bergman L, Thompson R. Stacking recommendation engines with additional meta-features. In: *Proceedings of the third ACM conference on Recommender systems.* ACM. 2009; pp. 109–116.

[135] Lommatzsch A, Kille B, Kim JW, Albayrak S. An adaptive hybrid movie recommender based on semantic data. In: *Proceedings of the 10th Conference on Open Research Areas in Information Retrieval.* Le centre de hautes etudes internationales d'informatique documentaire. 2013; pp. 217–218.

[136] Bellogín A. Predicting performance in recommender systems. In: *Proceedings of the fifth ACM conference on Recommender systems.* ACM. 2011; pp. 371–374.

[137] Ricci F, Rokach L, Shapira B, Kantor PB, eds. *Recommender Systems Handbook.* Springer. 2011.

[138] Pu P, Chen L, Hu R. A user-centric evaluation framework for recommender systems. In: *Proc. 5th ACM conf. recommender systems.* ACM. 2011; pp. 157–164.

[139] Knijnenburg BP, Willemsen MC, Gantner Z, Soncu H, Newell C. Explaining the user experience of recommender systems. *User Modeling and User-Adapted Interaction.* 2012;22(4-5):441–504.

[140] Xia B, Tan Z. Tighter bounds of the First Fit algorithm for the bin-packing problem. *Discrete Applied Mathematics.* 2010; 158(15):1668–1675.

[141] Tintarev N, Masthoff J. Designing and evaluating explanations for recommender systems. In: *Recommender Systems Handbook*, pp. 479–510. Springer. 2011;.

[142] Schafer JB, Konstan JA, Riedl J. Meta-recommendation systems: user-controlled integration of diverse recommendations. In: *Proceedings of the eleventh international conference on Information and knowledge management*. ACM. 2002; pp. 43–51.

[143] Brand M. Fast Online SVD Revisions for Lightweight Recommender Systems. In: *SDM*. SIAM. 2003; .

[144] Rendle S, Schmidt-Thieme L. Online-updating regularized kernel matrix factorization models for large-scale recommender systems. In: *Proceedings of the 2008 ACM conference on Recommender systems*. ACM. 2008; pp. 251–258.

[145] Chandramouli B, Levandoski JJ, Eldawy A, Mokbel MF. StreamRec: a real-time recommender system. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011; pp. 1243–1246.

[146] Tintarev N, Masthoff J. Effective explanations of recommendations: user-centered design. In: *Proceedings of the 2007 ACM conference on Recommender systems*. ACM. 2007; pp. 153–156.

[147] Nikulin V, Huang TH, Ng SK, Rathnayake SI, McLachlan GJ. A very fast algorithm for matrix factorization. *Statistics & Probability Letters*. 2011;81(7):773–782.

[148] Jahrer M, Töscher A, Legenstein R. Combining predictions for accurate recommender systems. In: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2010; pp. 693–702.

[149] Moore AW. Cross-validation for detecting and preventing overfitting. *School of Computer Science Carneigie Mellon University*. 2001;.

[150] Duda RO, Hart PE, Stork DG. *Pattern classification*. John Wiley & Sons. 2012.

[151] Witten IH, Frank E. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann. 2005.

[152] Schein AI, Popescul A, Ungar LH, Pennock DM. Methods and metrics for cold-start recommendations. In: *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 2002; pp. 253–260.

[153] Thompson C. If you liked this, you're sure to love that. *The New York Times*. 2008;21.