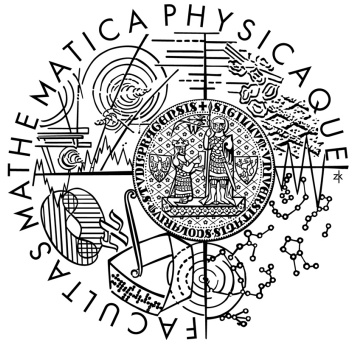
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Petr Cvengroš

Universal Recommender System

(Univerzální doporučovací systém)

Department of Software Engineering

Supervisor of the master thesis: prof. RNDr. Peter Vojtáš, DrSc.
Study programme: Computer Science
Specialization: Software Engineering

Prague 2010-2011

I would like to thank my supervisor prof. RNDr. Peter Vojtáš, DrSc. for his valuable comments and support. I would also like to thank the travel agency system administrator for lending the data set. I would like to thank attendees of the User-Web seminary for their comments and ideas on the work. Last but not least, thanks to my parents for supporting my studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Univerzální doporučovací systém

Autor: Petr Cvengroš

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: prof. RNDr. Peter Vojtáš, DrSc., Katedra softwarového inženýrství

Abstrakt: Doporučovací systémy jsou programy, které se uživateli nabízejí objekty (např. knihy nebo hudbu), které by pro něj mohly být zajímavé. Tyto systémy získávají vzrůstající popularitu a jsou intenzivně studovány výzkumnými skupinami po celém světě. Ve webových systémech, jako jsou internetové obchody nebo komunitní servery, bývají obvykle k dispozici různé datové zdroje, které mohou být využity k doporučování, např. atributy uživatelů a objektů, hodnocení objektů uživateli nebo nepřímá zpětná vazba získaná ze zaznamenaného chování uživatele. V této práci představujeme koncept Univerzálního doporučovacího systému (Unresyst), který dokáže využít těchto datových zdrojů a zároveň je doménově nezávislý. V práci navrhujeme způsoby využití systému Unresyst, ze současných metod používaných k doporučování vybíráme jako nejvíce vhodnou knowledge-based metodu kombinovanou s kolaborativním filtrováním. Dále analyzujeme datové zdroje v různých systémech a zobecňujeme je tak, aby byly doménově nezávislé. Navrhujeme architekturu systému Unresyst, popisujeme rozhraní systému a způsoby zpracování datových zdrojů. Dále přizpůsobujeme Unresyst na tři data sety z reálných systémů, vyhodnocujeme přesnost doporučení a srovnáváme ji se současnými algoritmy pro kolaborativní filtrování. Srovnání ukazuje, že kombinování různých datových zdrojů může zlepšit přesnost doporučovaní kolaborativního filtrování a může být použito v systémech, pro které standardní kolaborativní není vhodné.

Klíčová slova: Doporučovací systémy, Internetové obchodování, Doménová nezávislost, Knowledge-based doporučování, Kolaborativní filtrování

Title: Universal Recommender System

Author: Petr Cvengroš

Department: Department of Software Engineering

Supervisor: prof. RNDr. Peter Vojtáš, DrSc., Department of Software Engineering

Abstract: Recommender systems are programs that aim to present items like songs or books that are likely to be interesting for a user. These systems have become increasingly popular and are intensively studied by research groups all over the world. In web systems, like e-shops or community servers there are usually multiple data sources we can use for recommending, as user and item attributes, user-item rating or implicit feedback from user behaviour. In the thesis, we present a concept of a Universal Recommender System (Unresyst) that can use these data sources and is domain-independent at the same time. We propose how Unresyst can be used. From the contemporary methods of recommending, we choose a knowledge based algorithm combined with collaborative filtering as the most appropriate algorithm for Unresyst. We analyze data sources in various systems and generalize them to be domain-independent. We design the architecture of Unresyst, describe its interfaces and methods for processing the data sources. We adapt Unresyst to three real-world data sets, evaluate the recommendation accuracy results and compare them to a contemporary collaborative filtering recommender. The comparison shows that combining multiple data sources can improve the accuracy of collaborative filtering algorithms and can be used in systems where standard collaborative filtering doesn't work well.

Keywords: Recommender Systems, E-commerce, Domain-Independence, Knowledge-Based Recommender, Collaborative Filtering

Contents

1. Introduction

A present-day internet user is facing a plentitude of options. E-shops are offering a wide choice of various products, internet newspapers publish thousands of articles every day, loads of videos are published or uploaded by users every second. When a user is deciding what to buy, read or see, there are by far too many options to browse and choose the optimal one. Search engines don't help much, because a query like "find something I would like" is too vague. That's where recommender systems emerge. They help users to deal with the information overload, retailers to offer the most appropriate product for each client which results in increased customer satisfaction and loyalty.

The key feature of recommender systems is *personalization*. Unlike search engines, recommenders take into account the personality and past behaviour of each user. A typical recommender wouldn't present the same set of items to two different users.

Recommender systems are programs operating on large amount of data in software systems. Recommender systems try to present items such as books, music, news, etc. that are likely to be interesting for a given user. These systems may be helpful for users that are choosing between a large number of items and aren't willing to browse information about all available items.

Traditional recommender systems are specific to a particular domain of recommended items. Our view is more general and involves domain independence and utilization of any relationships among the entities in the domain. The thesis aims to create a prototype of a *Universal Recommender* system, a system applicable to various domains, predicting relationships of the given type using already known relationships. Our recommender adds a new layer above algorithms implemented in recommender frameworks, making the recommender easily applicable to any domain and system.

1.1 Thesis Structure

The *Introduction* chapter describes basic ideas of the thesis.

In the *Analysis* chapter we specify cases in which recommender systems can be used, we list various options of implementing a recommender. Finally we give a formal definition of a recommender system and explore the works that have been made on the universal recommender topic.

In the *Recommender Algorithms* chapter we give an overview of algorithms that are currently used for recommending, we judge their applicability to a universal recommender system and finally choose a combination of algorithms that will be used for a universal recommender.

In the chapter called *Relationship in Studied Systems* we present three domains in which we would like to use our universal recommender. We study the relationships that can be observed between the entities in the system and how they can be used for recommending.

In the chapter *Universal Recommender Design and Implementation* we propose Unresyst, a universal recommender system. We describe its interfaces, architecture and actions that should be performed in the given application parts.

Finally we present some details about the Unresyst prototype implementation, including its data model.

In the chapter *Verifying the Universal Recommender on Real-World Data* we describe how we adapted the Unresyst recommender to data sets containing real data.

In the chapter *Evaluating Recommender Results* we propose off-line methods and metrics that can be used for evaluating a universal recommender. We describe the results obtained for Unresyst prototype on the studied data sets and propose methods on how the results can be improved.

In the chapter *Conclusion and Future Work* we give a summary of the thesis and depict actions that can be done to continue the work done in the thesis.

1.2 Thesis Concepts

The section describes the main concepts of the thesis. We list the most important features and describe the purpose of the thesis.

1.2.1 Motivation

Modern web applications like e-shops or social networks tend to have much information about items in the system, they collect much data about users and their behaviour. When implementing a recommender in the system, it would be natural to use all the available knowledge for generating recommendations. However, current algorithms used for recommending usually employ only a single source of knowledge: a user-item rating. Using all available knowledge for recommending would lead to better recommendation accuracy and better reasoning for the provided recommendations.

Using multiple sources of data for recommending is also a current trend in the recommender research field, as illustrated by the current recommender system competition, organized by members of the Netflix prize winning team[1].

We would like our Universal recommender system (Unresyst) to take advantage of all available relationships. Let's show our idea on an example. Suppose, we have to generate recommendations in a system like Last.fm [1]. Registered users can let the system know what music they are playing on their computers through a media player plugin. Moreover, users can fill in some basic personal information, like gender and age. Songs and artists can be marked by tags[2]. In a recommender, we would like to be able to provide user-artist recommendations as illustrated on the figure 1.1. The links between a user and an artist he/she has listened is displayed as solid line. The similarity derived from user attributes and artist tags is displayed as dotted line. The user-item recommendations, we would like to generate, are displayed as dashed lines.

Then there are some domain-specific rules we would like to incorporate to the recommendation process. Some music is usually disliked by listeners with given demographic information. For example girls are seldom fans of "Heavy Metal" music, so we would like to decrease the chance of "Heavy Metal" music being

[1] See the *KDD cup recommender competition* at `http://kddcup.yahoo.com/index.php`

[2] Tag is a short textual notice, provided by a user to describe the song or an artist. For example, music can be tagged as "Rock", "Jazz", "Pop" or "Slow", "Indie".

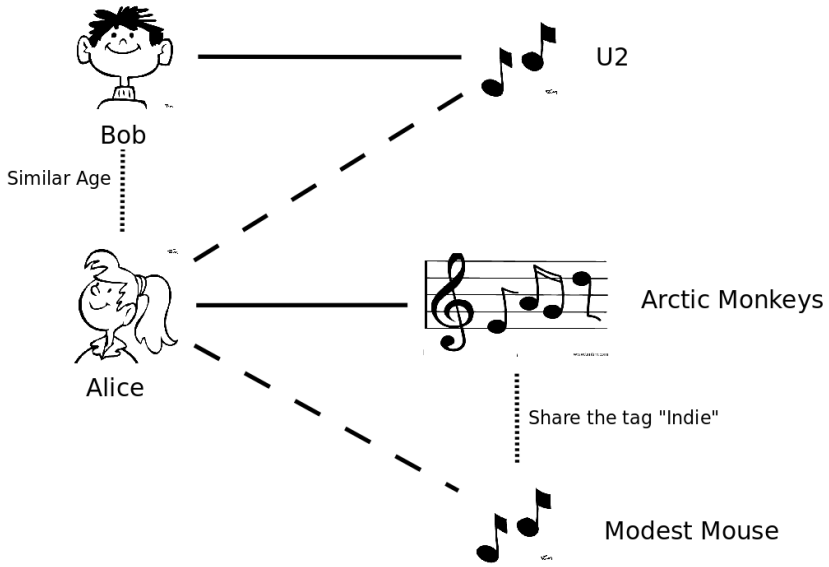


Figure 1.1: Using domain-specific similarity for recommending

recommended to girls. The situation is illustrated on the figure 1.2, the expected negative preference is displayed as a zig-zag line.

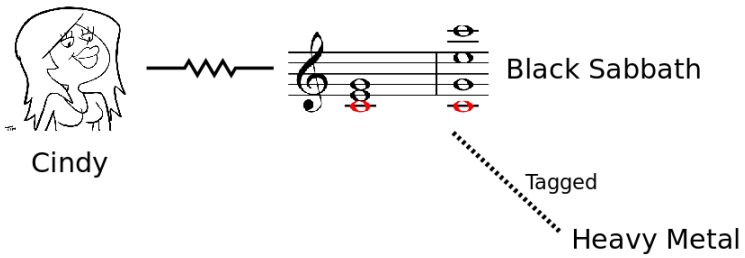


Figure 1.2: Applying domain specific rules for deprecating recommendations

Moreover, some artists are more likely to be played in a given period, e.g. because they have released a new album recently or they are on a tour. Such artists should be recommended more often to listeners. This artist property doesn't depend on the particular listener, it should just "help" the artist to get to recommendations for more users.

The obtained *compiled predictions* can be used directly for recommending or they can serve as an input for an ordinary recommender algorithm. More about recommending in the Last.fm data set can be found in chapter 6.1.

1.2.2 Main Features

The main concepts of the thesis are:

Problem analysis and design of a solution. The main goal of the work is to analyze the problem of a universal recommender, to create an architecture

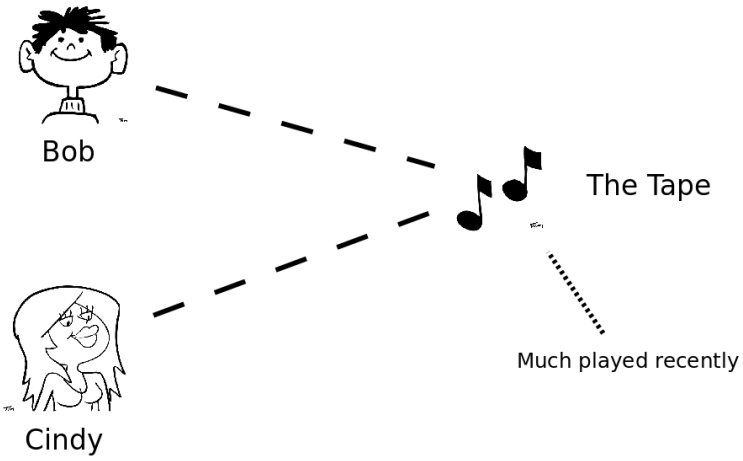


Figure 1.3: Using item property for recommendations

of the recommender system, design its interfaces and provide an implementation draft. The interfaces should be usable in real-world applications.

Domain independence. The recommender should work on any domain it's adapted to. It can't rely on any domain-specific assumptions.

Using various relationships. For the recommendation, the recommender should be able to use any number of relationships available in the domain.

Recommending given relationship. The recommender should be able to predict any given relationship, not just the "rating" relationship as most recommenders do.

A recommender, not a framework. The adaptation to the given domain should be simple and should require minimum changes in the parent system. As opposed to recommendation frameworks that give resources to implement a recommender, our system should be a complete recommender ready to be adapted and used on an arbitrary domain.

Verification on various domains. The recommender should be adapted to at least three domains to verify its universality and usability of the recommender interfaces.

1.2.3 Thesis Classification

The recommender system research field can be divided into the following categories, as described in the paper [2]:

1. Gathering user preference
2. Algorithms transforming past user actions and other data to recommendations
3. Privacy, legacy and other aspects

4. Measuring recommender accuracy

5. Recommender system implementation

The thesis primarily concentrates on 2: generating recommendations from known data. The area 4 is also partly covered in the thesis, as the results of the recommender on various domains had to be compared to other known approaches. The remaining areas are undoubtedly as important as the selected ones, however the selected fit the best to the demonstration of our universal recommender idea.

Finally we list the topics that are out of the scope of our work:

Scalable implementation The provided recommender application is meant as a prototype and isn't intended to be scalable for large data sets.

Central server for multiple systems. A single Unresyst instance isn't intended to be used as a central server for multiple domains - a Unresyst instance always refers to a single domain. Unresyst doesn't deal with matching the same subjects and objects appearing in multiple systems.

Collecting user preference Unresyst doesn't help the system collecting any user actions for recommending. Implicit and explicit user feedback can be used for generating recommendations in Unresyst, but it has to be handled by the outer system and passed to Unresyst in the form of business rules and relationships.

2. Analysis

In the chapter we discuss the circumstances under which recommender systems are a preferred option both for users and system holders. Later we describe the reasons for using our universal recommender system (Unresyst), we compare our recommender to existing solutions. We propose draft of a process model for activities related to running Unresyst. Finally we formally define the problem of recommending.

Recommender system is a part of a web-based application, that uses data about users and their behaviour to provide them with items which are the most relevant to them. The recommended items are typically things that are liable to user taste, like books, music, news, etc.

2.1 Recommender System Applicability

For a successful implementation of a recommender system, several conditions have to be fulfilled. A summary of applicability conditions for a collaborative filtering recommender can be found in the overview of collaborative filtering recommender systems [3]. We list some of them that are valid for any recommender, independently of the recommender algorithm.

Many items In the domain there are many items that might be interesting for a user. It is not possible for the user to browse all of them.

Choice based on taste The choice of the items depends on the taste of each user. If there were some objective criteria for recommending items to users the recommender system wouldn't be much helpful.

Taste data In the system there have to be some data about the users interacting with items that can be interpreted as expressing taste. It doesn't matter if these are explicit rating data or implicit capture of user behaviour, but there have to be some.

Homogeneous items The items in the domain have some common attributes, they can all be covered by taste data, e.g. they can all be viewed or rated.

2.2 Recommender System Benefits

The section summarizes reasons why recommender systems are useful for both users and system holders.

2.2.1 Benefits for System Users

In a web-system like an e-shop or an internet radio a user is overloaded with items. The system database typically contains thousands of items divided into several categories. The recommender provides user with items that are likely to be interesting without requiring him/her to search for them. This saves user's time and helps him/her spend more time exploring the interesting items than messing

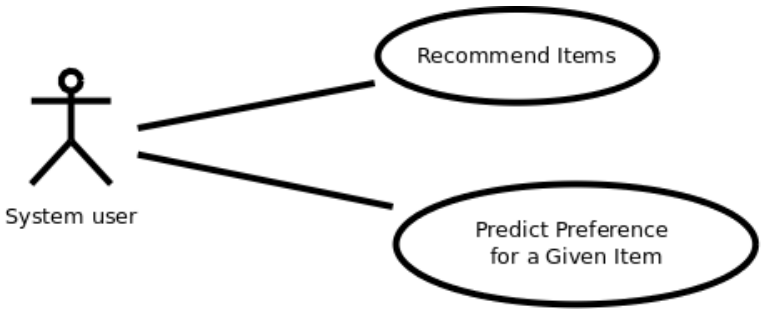


Figure 2.1: Use cases for a recommender system.

around uninteresting items. The provided recommendations are personalized, they reflect the past user's behaviour and available data, so the recommendations are likely to be much accurate.

In comparison to advanced multi-criteria search, recommender systems are better at reflecting users' preference in criteria that aren't easy to express and depend on the taste of the user. The advantage of recommender system is their user-friendliness - they don't require any explicit search terms, they just use the facts they already know about the user. This can be a significant advantage for inexperienced internet users.

Opposed to search engines, recommender systems present items ordered by the expected preference, which can be more accurate when taste is important. However, multi-criteria search can act as a supplement to a recommender system, for performing well defined search queries. E.g. if the user is absolutely certain he/she wants a red lap-top with the given display size at the lowest price, the multi-criteria search is the best option to find such a product.

Recommender systems help users find novel items. When a user only browses and searches the system items, he/she usually gets stuck to a relatively small group of items. The recommendations usually contain items chosen by various criteria, so that they contain both items the user is familiar with and items that are likely to be interesting for the user, but the user wouldn't otherwise discover them.

Apart from that, a recommender system can predict user's interest for a particular item. E.g. the last.fm festival recommender can predict user's interest in a particular festival based on the user's taste and the festival lineup[1]. This also saves user's time, he/she doesn't have to examine details of the items that aren't likely to be interesting for him/her.

2.2.2 Benefits for a System Holder

A satisfied user is a key for the success of a web-system business. If the user easily finds interesting items in the system, he/she tends to return to the system regularly, which results in a high visit rate.

In an e-shop, if a user easily finds items that are appropriate for him/her, he/she is more likely to buy them. In a non-retail system like an internet radio,

[1]see Last.fm festival page at `verb!http://www.last.fm/festivals`

the system holder can also profit from a user finding appropriate items quickly. The user spends more time on pages that provide interesting information for him/her, which leads to more appropriate context advertisements with higher earnings.

Another interesting feature recommender systems provide to e-shop holders is the ability to include promoted items to recommendations. E-shop holders can artificially increase the recommendation rates of the promoted goods so that they appear in recommendations for more users. However, this feature should be used moderately because it distorts the results of the recommender algorithm. A massive propagation of items that are promoted by the system holder but are not very interesting for particular users may lead to a loss of user's trust for the recommender.

And last but not least, a recommender system provides a competitive advantage. Web business leaders like Google [4], Amazon [5] or Yahoo! [6] have already implemented recommenders in their systems, for smaller companies a recommender system may be a key advantage over their rivals.

2.3 Universal Recommender Analysis

In the section we describe the position of the proposed universal recommender system (Unresyst) on the market. We depict advantages and disadvantages of the Unresyst recommender, we compare Unresyst to existing solutions. Finally we propose processes for incorporating Unresyst to a web-based system and for running a system with Unresyst.

The proposed Unresyst recommender is a domain-independent recommender system that is able to use various relationships between the domain entities. Unresyst is an independent software component that logically stands outside the web-based parent system. It communicates with the parent system through a defined API[2]. Detailed information on the system architecture can be found in the section 5.1.

Unresyst itself doesn't contain any domain-specific data or methods. In order to work correctly on the given domain, it has to be adapted to it. Adaptation data make Unresyst use domain knowledge for making recommendations. Adaptation is designed so that it's as easy as possible. Adaptation data are stored in the parent system and are used during the communication with Unresyst. Detailed description on the adaptation can be found in the section 5.2.1.

In order to provide high quality recommendations, Unresyst can take use of any *relationships* between the entities in the system domain. Additionally, the business knowledge can be transformed to *rules* and used for generating recommendations. Both rules and relationships are stored in the adaptation data. See sections 5.2.1, 5.2.1 for details on rule and relationship representation.

2.3.1 Universal Recommender Applicability

Companies that run big global web-systems generally have enough resources to develop their own recommender systems that are specifically designed for their

[2]API stands for Application Programming Interface.

needs. Therefore, Unresyst is aimed at small and mid-size web systems. The goal of the Unresyst concept is to provide companies a way to implement a recommender system without having to develop their own recommender solution, which is often both time and money consuming. With a bit of configuration, Unresyst can provide recommending accuracy comparable to a custom-made recommender system. We suppose the web-system application uses a database to store its data. The data are manipulated trough an SQL[3] or ORM[4] layer.

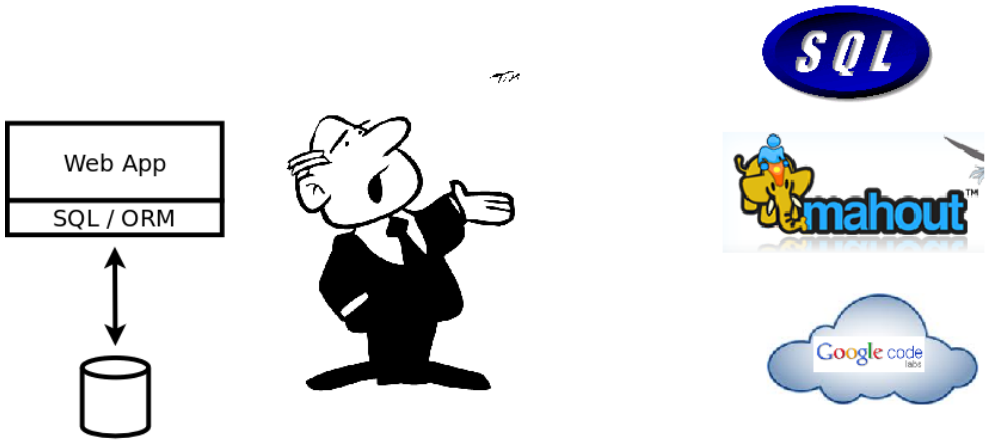


Figure 2.2: A system owner choosing the way of recommender system implementation, deciding between the solutions presented in the following sections.

Unresyst can take use of any attributes and relationships that can be tracked in the parent system. Data that can be used by Unresyst include but are not limited to:

User behaviour In recommendations we can use data about user viewing item pages, the time users spent on each page, analysis of user's mouse move and any other implicit feedback provided by users.

Explicit feedback We can also use the explicit user feedback, like rating on a scale, thumb up/thumb down rating, in systems where such mechanisms are available.

Demographic information In systems where demographic data are available, we can use them for making recommendations. For users we can use attributes like age, place of residence, or education. For items we can use the data about their manufacturers like country or size of the company.

Social relations In systems, where users interact between each other, the social data can be used for providing recommendations. We can use relationships between users like friends, sent personal messages or viewing other user's profile.

[3]SQL stands for Structured Query Language, a language most commonly used for structured data manipulation.

[4]ORM stands for Object-relational mapping, a technique used for manipulating data directly in the application without using SQL.

Other relationships Any other relationships between entities of the system can be used for making recommendations. Such relationships include user-defined tags which can be used for determining similarity of the entities marked with the same tag, or user's preference for items marked with a particular tag.

Recommendations produced by Unresyst can take into account both objective and subjective criteria. Objective criteria for recommending can be expressed in business rules (e.g. not recommending expensive items to users with low income). Subjective criteria, as the expressed preference for an item, are used for recommending.

The most common operations for a recommender, providing recommendations and predicting the preference for a particular item, are performed in time independent of the number of the entities in the system. The exact recommendation and prediction time may vary between the recommender algorithms. However most of the presented current algorithm use some kind of recommendation index which enables them to perform recommendations in constant time.

The Unresyst software component is logically independent of the parent system, it provides an API for all common operations. The API and the domain-specific configuration is made so that the adaptation to the domain and to the parent system is as easy as possible.

2.3.2 Comparison to Other Solutions

In the section we compare Unresyst to other solutions that are available for a small to mid-size company that likes to implement a recommender system at a reasonable time and cost. Unresyst is laying somewhere between the presented options, using the advantages of all and minimizing their drawbacks.

Filtering Items in Queries

For a system holder, a minimalistic approach to item recommending is to implement item filtering in a query language (e.g. SQL) above the system database. This approach can be very efficient when the holder wishes to implement a small set of well-defined filtering rules, that don't imply user's taste. Such an implementation can be fast and cheap, as it doesn't require any significant changes to the system, nor incorporation of a third party component to the system.

However this approach works well only for well-defined and static filtering criteria that aren't costly to evaluate. With an increased size of the rule set, the maintenance of the queries becomes difficult. When the system holder wishes to implement some kind of evaluation of user taste, the situation even complicates. Complex queries combining various data can take a long time to evaluate, which results in slow generating of the system web pages and nervous users.

Simple hard coded filtering queries don't allow exceptions to the expressed rules. E.g. if we set up a hard coded query for not recommending an item to users with the given demographic data, the item won't be recommended even if we have a high preference signs from other sources such as user behaviour.

The Unresyst recommender offers a transparent way of managing the business knowledge that is used for generating recommendations. The rules and relation-

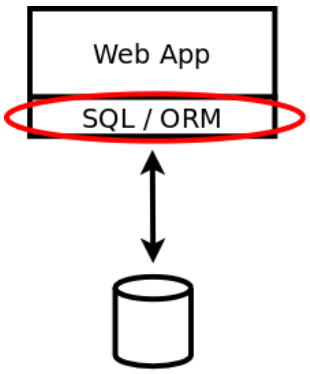


Figure 2.3: Implementing recommendations directly in the application.

ships aren't written in query language but in a more human-friendly form. The business knowledge is kept in one place and is separated from the recommender algorithm. Changing the significance of the rules and relationships can be done there too. All recommendations are produced using the given business knowledge. The rules and relationships are combined according to their significance.

The recommender algorithm which is one of the Unresyst layers, can use complex techniques to exploit the business knowledge. Therefore the recommendations produced by Unresyst are generally more accurate. At the same time the algorithm uses a recommender index that makes the prediction time independent of the number of entities in the system.

The main disadvantage of Unresyst compared to the minimalistic option is the required initiatory effort required to start producing recommendations. Firstly, Unresyst has to be configured for the system, which means specifying the subject and object domain, definition of the predicted relationship and finally writing down the business knowledge in the form of rules and relationships. Then the recommendations have to be incorporated to the parent system - calling of Unresyst methods has to be added to necessary places, the recommender user interface has to be incorporated to the system. However the initial effort pays off as the latter changes in recommender configuration can be done easily at one place.

Running a system that uses a recommender involves some additional effort. The index of the recommender algorithm has to be built before the recommender is used. This can be a time-consuming action, depending on the algorithm and number of entities in the system. The recommender index has to be kept up to date by either rebuilding now and then or by continuously calling methods for index update. These methods keep the index updated with every change in the system but they require some additional time during save and update of the system entities and they might be unavailable for some recommender algorithms. The issues of the recommender index however occur when running any recommender that takes constant time for recommending items.

To summarize the comparison, Unresyst is a better option than filtering items in queries unless the system holder only wishes to implement a small set of fixed filtering rules, that don't need to be changed over time and he/she isn't interested in exploiting the rules by a recommender algorithm.

Using a Recommender Framework

The system holder can choose to implement a recommender based on a recommender framework library like Duine [7] or Mahout [8]. These are software libraries providing implementations of common present-day algorithms like collaborative filtering or content-based algorithms. For details on recommender algorithms see the section 3.

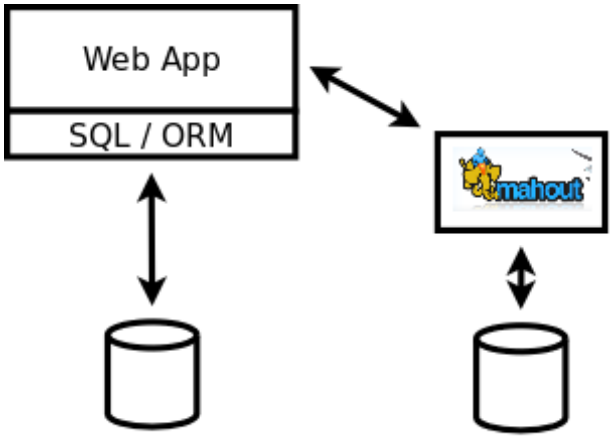


Figure 2.4: Implementing recommendations using a recommender framework.

Recommender frameworks aren't real rivals to Unresyst as they can be used in the Algorithm layer of Unresyst. In this section we compare the usage of mere recommendation framework directly incorporated into the system and the usage of Unresyst with a recommender algorithm below, no matter if the in-built recommender algorithm is used, or if the algorithm implementation is overtaken from a third party library.

Most algorithms in recommender frameworks rely on presence of a single explicit preference indicator like rating. If a system holder likes to use more sources of negative or positive preference data, he/she has to care about converting the preference to the rating used in the recommender algorithm. Using multiple sources of preference data, like data about user behaviour, social data or demographic data about users, is necessary for the algorithm to provide good recommendations. Only recommenders that use all available knowledge can produce high quality recommendations.

The API of most of the present-day recommender frameworks isn't very developer-friendly and let the developer deal with converting between the system entities and recommender entities. This requires additional programming. Moreover, the algorithm API often lacks methods for updating the entities. Therefore in order to propagate the changes in the system data to recommender, the system administrator has to rebuild the index of the recommender algorithm. The recommendations can't be made "on-line" from the current system data. Unresyst brings up a simple and complete interface for using the recommender in the parent system.

In addition, Unresyst enables system holders to run multiple instances of a recommender in one parent system. E.g. there can be a product recommender

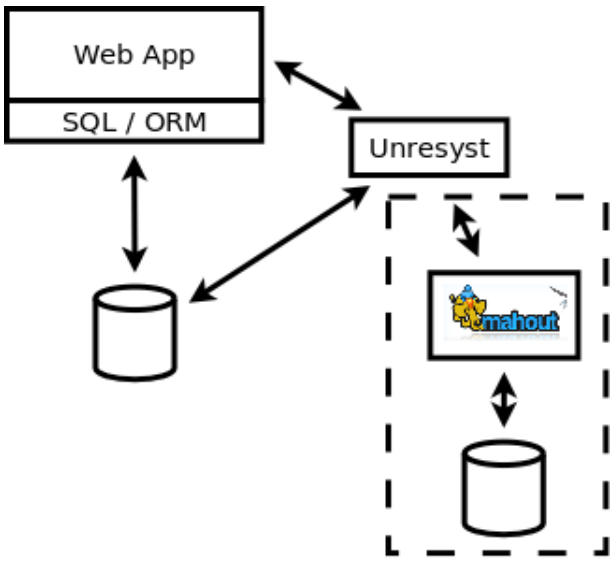


Figure 2.5: Implementing recommendations through Unresyst, optionally using a recommender framework algorithm.

and user group recommender in the system, that is working in one Unresyst installation, but is independent of each other and can be using different rules and relationships to produce recommendations.

To summarize the comparison, incorporating a recommender framework is usually complicated both technically and intellectually. This may be one of the reasons why the present-day recommender frameworks haven't been widely used in real world web-based systems. Unresyst adds a new layer above the algorithms that solves the problems with combining different sources of preference data and enables the system holder to incorporate the business knowledge to recommendations.

Using a Recommender Service

Another option, the system holder can possibly use in future is implementing a recommender using a recommender service. The first attempt to provide a such a service is the Google Prediction API [14]. The service is still in development in Google Labs, available for experiments only through a waiting list. Only little data about the way the service internally works is available for public. There's only one obvious disadvantage of the approach: all data that are to be used for recommending have to be passed and regularly updated through the API. This can raise privacy issues system owner's distrust.

2.3.3 Unresyst Process Model Draft

In the section we propose a design of processes related to Unresyst setup and maintenance. Note that this is only a brief simplified sketch illustrating how real processes could possibly look like. The actual usage of the system would be highly dependent on particular company, its environment and people. A more technical

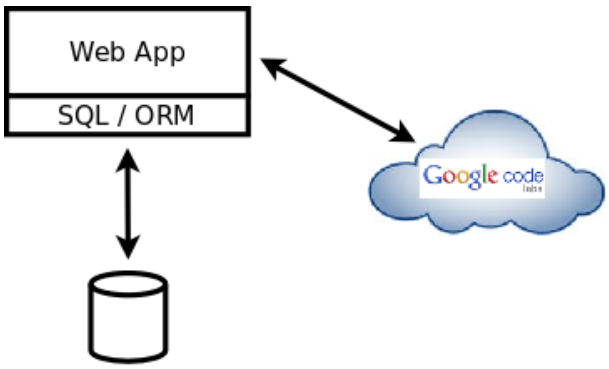


Figure 2.6: Implementing recommendations using a recommender service

view on Unresyst setup and evaluation can be found in chapters about Unresyst architecture (5.1) and evaluation (7).

Unresyst Setup

After the system holder or the company management agrees on using Unresyst for recommending in their system, the setup activities may begin. One of the main roles in the recommender operation is the *Domain Expert*. He/She is familiar with the business the system is operating in, knows a lot about entities and their relationships both in the real world and in the system.

Firstly, the *Business Analyst* interviews the *Domain Expert* in order to find out the requirements for the recommender and type and amount of data that is available for making recommendations. The output of this activity is a report that thoroughly describes the purpose of the recommender and knowledge that is available for recommending in the parent system. The report should at least include the list of recommender instances that will be incorporated to the system and for each recommender it should describe the subjects and objects of recommendations, the description of the predicted relationship and finally a description of all relationships and business rules that are available for the domain. The report is prepared by the *Business Analyst*, is approved by the *Domain Expert* and is imperative for the later activities. This activity should be done properly, multiple iterations of interviewing and recording will be needed.

The report is delivered to the *Data Specialist*. *Data Specialist* is a person highly familiar with the data model of the system. He/she transforms the report into an Unresyst configuration, which formally defines subjects, objects, rules and relationships for each recommender. After the configuration, the *Data Specialist* performs the recommender build, evaluates the results of the recommender and verifies sample recommendations with the *Domain Expert*. *Data Specialist* tries improving the recommender evaluation by tuning the rule and relationship parameters, until a given accuracy threshold is reached.

The *Developer* is next to come. *Developer* is familiar with the implementation of the parent system. He/she incorporates the Unresyst API calls to the appropriate parts of the system, includes the recommender user interface into the parent system interface.

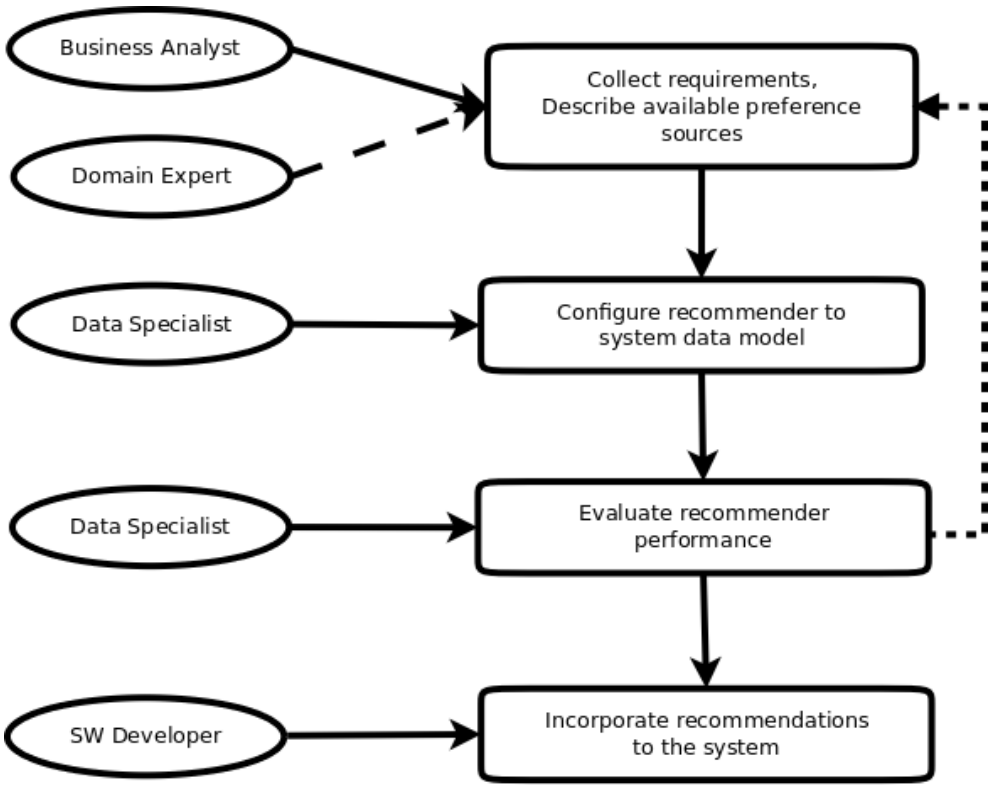


Figure 2.7: Unresyst Setup process, a draft of a process model.

After enough tests have been performed, the version of the parent system using Unresyst is put into operation. Tests should include user-testing inquiring real users of the system.

Unresyst Maintenance

Running a recommender system isn't just a matter of setup. Keeping the recommendation quality high requires a regular maintenance activity. The frequency of the maintenance tasks depend on many factors such as domain dynamics and internal capacities of the company. It should be done at least a few times a year.

Maintenance activities follow selected setup activities in order to improve accuracy of the recommender. *Business Analyst* interviews the *Domain Expert* to find out what has happened in the domain business since the last maintenance activity. Requests for collecting new types of user-data can be included. The news are delivered to the *Data Specialist*, who incorporates them to the recommender configuration. He/She also revises the changes in the system that could have affected the recommendation sources and adjusts the configuration to use any newly available data. Finally he/she evaluates the recommender with the new configuration and tunes the parameters until the desired evaluation is obtained. The new Unresyst configuration and sample recommendations are consulted with the *Domain Expert*. The new configuration is finally tested and put into operation.

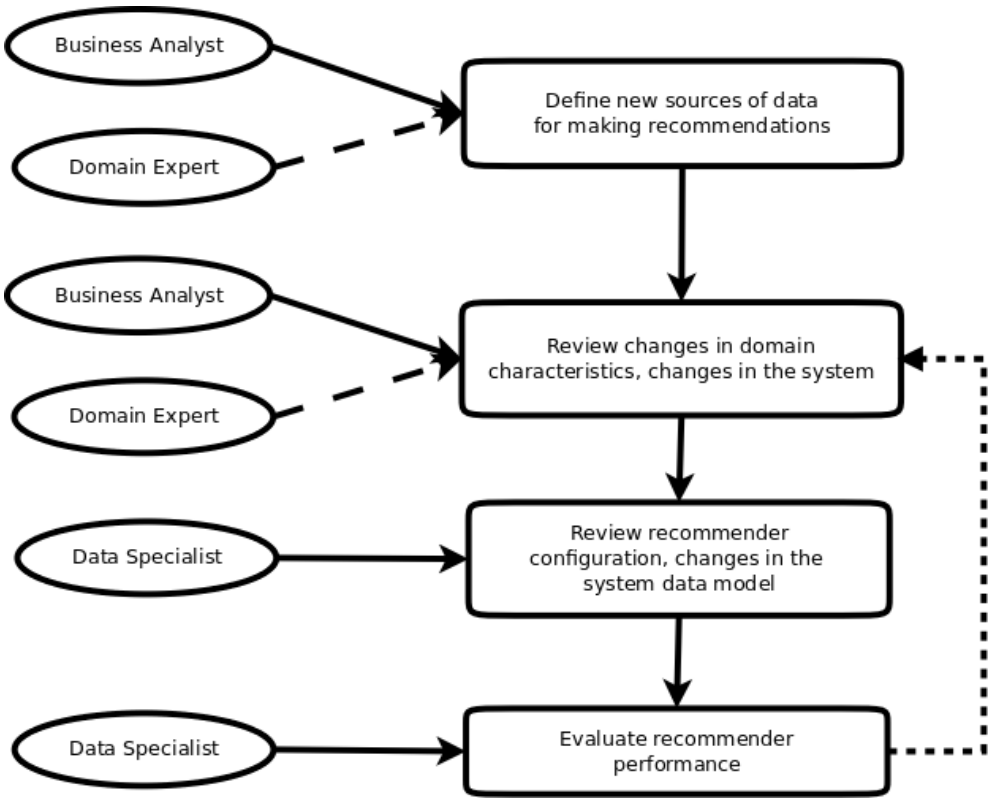


Figure 2.8: Unresyst Review process, a draft of a process model.

2.4 Definitions

In this section we give some basic definitions of terms and problems that appear in the whole thesis.

2.4.1 Basic Terms

The basic terms used both in definitions and throughout the thesis are:

Subject Subject of a recommendation is an entity to which the recommender presents its recommendations. A subject can be a user as well as a user group or any other entity appearing in the parent system.

Object Object of a recommendation is an entity which can be recommended to subjects. Objects can be books, songs, or any other entities.

Predicted relationship Predicted relationship is a subject-object relationship that we try to predict for making recommendations. An example of such a relationship is the relationship *User has listened to Artist*.

Expectancy of the predicted relationship For predicting a relationship we need a rate - a number determining the estimated probability that the *predicted relationship* will occur between the subject and object. In other words, Expectancy can be explained as subject-object preference in means

of the predicted relationship. In our example, the expectancy would mean how much the user is likely to listen to the artist.

2.4.2 Recommender System

Informally, a recommender system is a system that presents some chosen *objects* to a given *subject*. The objects are chosen so that they are likely to invoke a positive response of the subject.

Similarly to the recommender problem definition in the recommender system survey [9], we formally define the recommendation as choosing the object o_s for each subject s, so that the object maximizes the subject's utility function u_R:

$$\forall s \in S : o_s = arg\ \underset{o \in O}{max}\ u_R(s, o), \tag{2.1}$$

where S is the subject domain, O is the object domain. The utility of the object to the subject is measured by the utility function $u_R : S \times O \to [0, 1]$.

In our definition, we parametrize the utility function by the predicted relationship R, $R \subset S \times O$, because our recommender should be able to predict the given relationship R between subjects and objects, e.g. *User has listened to Artist*.

For example a utility for *Bob* of listening to the artist *Beck* can be 0.7 ($u_R(Bob, Beck) = 0.7$), which means *Bob* would be pretty satisfied listening to *Beck*, as he likes such music. *Bob*'s utility listening to *Beatles* can be 0.5 as *Bob* is indifferent to music *Beatles* play. Finally *Bob*'s utility listening to *Rihanna* would be 0.1 as he doesn't like her music.

2.4.3 Relationship Prediction

Relationship Prediction $\hat{u}_R$ is an approximation of the utility function u_R:

$$\hat{u}_R : S \times O \to [0, 1] \times E, \tag{2.2}$$

where E is a set of textual explanations.

For a given subject-object pair, $\hat{u}_R$ gives:

Expectancy: the predicted probability of occurrence of the given relationship type between the subject and the object. It's a number between 0 and 1. The recommended objects are usually presented ordered by the expectancy. The exact expectancy value doesn't have to be presented.

Explanation: a textual explanation why the object has been recommended. For some recommender algorithms explanations aren't available.

Our goal is to estimate the utility function u_R by the expectancy given by $\hat{u}_R$, so that $proj_1 \circ \hat{u}_R \approx u_R$, where $proj_1$ is a projection that discards the explanation from the function $\hat{u}_R$.

For instance, our recommender predicts that *Bob*'s listening to *Beck* utility is 0.8, because *Beck* is similar to *Sonic Youth*, a band that *Bob* has listened yesterday. In our notation: $\hat{u_R}(Bob, Beck)$ = (0.8, "Bob has listened to Sonic Youth band, which is similar to Beck").

Our expectancy concept is similar to the one used in the Duine framework [7]. The prediction techniques in Duine give a *Prediction*, *Validity indicators* and an *Explanation* for each subject-object pair. *Prediction* is an estimation of subject's rating of the object. *Validity indicators* determine the *confidence* of the technique when predicting the rating. Low values mean, that the technique isn't confident about the *Prediction* for the given subject-object pair, e.g. it doesn't have enough information to make a valid prediction. *Explanation* has the same meaning as the one in our recommender.

In a universal recommender we can't assume there will be a rating relationship between subjects and objects. Moreover, we are supposed to be able to predict any given relationship between subjects and objects, not just the preference expressed by the rating. In a Universal recommender, we only predict a 0/1 indicator: there is the predicted relationship between the given subject and object, or there isn't. Hence we can shrink the prediction and confidence concepts into one - expectancy.

Let confidence be a function $D : S \times O \rightarrow [0, 1]$ determining how sure the recommender is about a positive or negative recommendation of object o to subject s. For converting confidence to expectancy, we use the function $\Pi : True, False \times [0, 1] \rightarrow [0, 1]$, given by the formula 2.3, where d is the confidence value, and p is the positiveness, which is $True$ for positive recommendations and $False$ for negative.

$$\Pi(p, d) = \begin{cases} \frac{1}{2} + \frac{d}{2}, & \text{if } p = True \\ \\ \frac{1}{2} - \frac{d}{2}, & \text{if } p = False \end{cases} \tag{2.3}$$

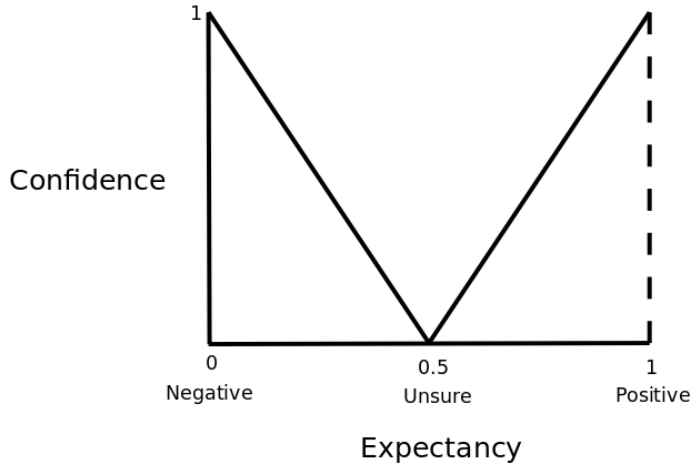


Figure 2.9: The dependency between expectancy and confidence.

Let's show the conversion on an example. With respect to the fact, that *Bob* has listened to *Sonic Youth* whose music is similar to *Beck*'s, our recommender gives a positive recommendation of *Beck* to *Bob*. From the similarity between *Beck* and *Sonic Youth* it determines the confidence 0.6 of the recommendation. The positiveness is $True$ because the recommender recommends *Bob* to

listen to *Beck*. To convert the confidence to expectancy, we use the Π function: $\Pi(True, 0.6) = 0.8$.

The dependency between relationship expectancy and prediction confidence is illustrated on the figure 2.9. Expectancy values just above zero indicate that the recommender is quite sure that there won't be the given relationship between the given subject-object pair, i.e. *Bob* wouldn't like to listen to *Rihanna*. Expectancy values near 0.5 mean low confidence for the prediction. Especially 0.5 means that according to the recommender, the subject is indifferent to the object. Expectancy values close to 1 mean that recommender is quite sure, that there will be the predicted relationship between the given subject-object pair E.g. *Bob* will like listening to *Beck*.

Expectancy isn't much intuitive for specifying strength of the relationships in a system, so the concepts of positiveness and confidence are used there (see 5.2.1, 5.2.1).

In our universal recommender we don't support rating prediction directly. But as subject-object rating means a level of subject-object preference, we can estimate rating using our relationship expectancy. We only need to define mapping from relationship expectancy values to rating. For example, we can express *Bob*'s expectancy 0.8 to *Beck* as rating 4 stars on a 0 to 5 star scale.

2.4.4 Recommendation

For the given subject, a *Recommender* performs a *Recommendation* $\rho^{\hat{u}_R}$ – it chooses at most N objects with the highest expectancy for the subject and relationship prediction function $\hat{u}_R$:

$$\rho^{\hat{u}_R} : S \to (o_1, \ldots, o_N),\ o_1, \ldots, o_N \in O \tag{2.4}$$

The recommender chooses N objects that are likely to be interesting for the user, i.e. they maximize the $\hat{u}_R$ function for s. The chosen objects are presented to the subject ordered by expectancy.

For instance, an artist recommender presents N artists to *Bob*. Artists are sorted by the expectancy.

2.5 Operations in Recommender Systems

In a recommendation system, some common operations are:

Recommender build After a configuration of the recommender is done, the recommender has to prepare itself for recommending objects. This operation may be pretty time consuming, as this doesn't have to be performed very often.

Recommendation Recommendation corresponds to evaluating the function $\rho^{\hat{u}_R}$ for a given subject. Recommendation is the most common operation for a recommender system. Choosing objects with the highest expectancy for the given subject among all objects can be a very expensive operation. That's why most recommender systems introduce some kind of index, enabling the recommender to recommend objects real-time. This operation should be

performed very fast, ideally in a constant time independent of the number of items in the system.

Prediction Prediction corresponds to evaluating the function $\hat{u}_R$ for a given subject-object pair. Prediction is used for estimating the subject's preference in a particular object. It can be used for direct displaying when browsing through the objects or for ordering displayed objects when browsing in an object catalogue. Usually it is not possible to store predictions for all subject-object pairs, so the prediction has to be computed on-line. Therefore it has to be pretty fast.

2.6 Related Work

The recommender research area has been strongly influenced by the Netflix prize [10]. In the years 2006-2009, the Netflix DVD rental company held a competition on improving their recommender, awarded by the grand prize of US$1,000,000. To the competition participants, Netflix exposed a data set of over hundred millions of ratings. Over 48,000 teams from 182 different countries participated, the competition had a huge response in both scientific and mainstream media. A paper describing the winning solution was publicly released [11]. The competition brought benefits for the Netflix company as well as for the whole recommender research world [12]. Unfortunately, the second Netflix prize was canceled after user privacy issues [13].

2.6.1 Domain Independent Recommenders

The idea of a domain-independent recommender isn't completely new. The most notable project in this field is an open-source project called *AURA* (The Advanced Universal Recommendation Architecture) [15]. The project was supported by Sun and later by Oracle, but since the end of 2009 there's been no activity on the source code repository. The experimental music recommender based on AURA, *The Music Explaura* is inactive. The project aimed to create a universal hybrid recommender system, combining the two most used approaches: collaborative filtering and content-based recommendation (see 3.2, 3.1). The system should have been working above a data store. More on the project can be found in [16].

Another notable work on universal recommender is a US software patent application *Universal system and method for representing and predicting human behavior* [17]. The text contains a mixture of marketing proclamations, ideas on the recommender architecture, description of recommender and machine-learning algorithms and ideas on capturing implicit user ratings. In our opinion, the recommending methods presented in the text could be working well only on very small sets of data, for larger data sets, the amount of time needed for performing some basic operations wouldn't be acceptable. The final paragraph, trying to claim everything around, from vector object representation to capturing the action of user adjusting the volume on his music player, gives a negative example of where the idea of software patents can lead. As far as we know there's no functional recommender system based on this patent.

An implementation of a domain-neutral component for recommending in e-shops was a part of the thesis [18]. The component was also dealing with gathering implicit feedback from users. Recommending was done using some of the common recommender algorithms like collaborative filtering. The component was tested on two e-shops.

The Universal Recommender White Paper [19] describes a recommendation engine working on semantic data sets. The proposed recommender is operating on the full semantic representation of domain data, where entities are connected by various relationships and according to the paper it generalizes all common recommender algorithms. The paper identifies problems that have to be overcome in order to implement such a recommender, such as normalization of the represented relationships. We use some ideas presented in the paper in the section 3.6.1 and simplify the approach presented in the paper.

3. Recommender Algorithms

In the chapter we list and describe algorithms that are commonly used for recommending. For each algorithm we describe its capability to be domain-independent, and the its capacity to use multiple data sources as an input.

3.1 Content-based Methods

Content-based filtering methods are one of the oldest and most popular methods for recommending. The principle of these methods is recommending objects that are similar to some objects, the user liked in past. The similarity among the objects is determined from the values of their characteristics.

These methods are widely used in text-based applications, for recommending documents or web sites [9]. One of the implementations of a content based based recommender is the Music Genome Project [20]. A musician analyzes each song in the system, giving a value for each of the musical characteristics.

The figure 3.1 shows an example of a content based recommender. The known relationships are marked as full arrows, the calculated or inserted object similarity by the dotted arrow and the predicted relationship by the dashed arrow.

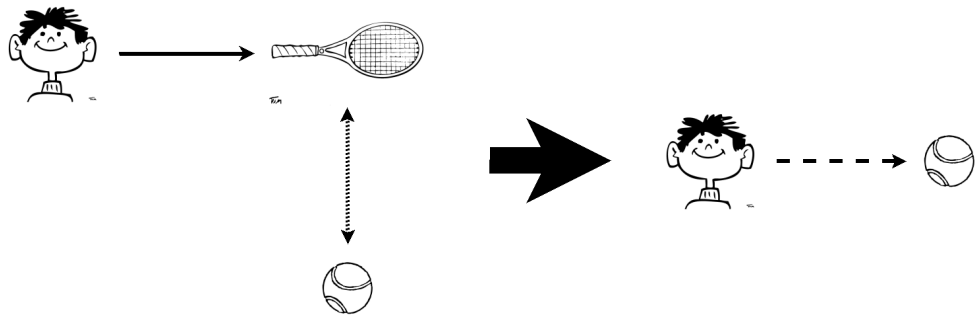


Figure 3.1: Content-based Methods

More formally, according to [9], the utility function approximation is done as depicted in the formula 3.1, where $sim : O \times O \rightarrow [0, 1]$ is a measure of object similarity, based on the object attributes. The sim function is applied on objects o' that are already in the relationship R with the given subject s. The sum can be replaced by a more sophisticated functions in some implementation. The method can be enhanced by modeling subject preferences for object attributes, as described in [21].

$$\hat{u}_R^{CB}(s, o) = \sum_{(s,o') \in R} sim(o, o') \tag{3.1}$$

Benefits and Drawbacks

The accuracy of the provided recommendations strongly depends on the similarity determination. If the items are characterized properly and the subjects tend to

like uniform sets of objects, the recommender can be pretty successful, as Genome [20] mentioned above. However there are several drawbacks of the method, as mentioned in the survey [9]:

Limited content analysis The set of characteristics assigned to each object is always limited and is never able to fully characterize the object. Moreover the values of the characteristics have to be determined either manually which is time-consuming or automatically which currently works well only for text documents.

Overspecialization The method always recommends objects that are similar to some that the subject liked in the past. Therefore the recommender never broadens subjects' horizon by recommending diverse objects.

New user problem When a new subject emerges in the system and has no relationships to objects, the system cannot make any recommendations to the subject.

Suitability for The Universal Recommender

In a domain-independent recommender we cannot assume that there are sufficient object characteristics available. However we would like to use the concept of recommending objects that are similar to the ones that were already liked. The concept has to be implemented so that the similarity rules are given in the adaptation phase. Therefore the recommender wouldn't rely on any specific object attributes and would remain domain-independent.

The advantage of the content-based method is its capability of using multiple attributes for determining object similarity.

3.2 Collaborative Filtering

Collaborative filtering is currently one of the most widely used technique for recommending. Market leaders as Google [4], Amazon [5] or Yahoo! [6] use recommenders at least partially based on collaborative filtering.

For recommending an object to a subject, the method uses relationships between other subjects and objects. For a given subject s, the algorithm finds subjects that like most of the objects that s likes. Recommended objects are then taken from other objects liked by the found subjects.

In the figure 3.2 there is a simplified example of a recommendation made by collaborative filtering. The full arrows represent already known relationships. The two subjects both have a relation to one item and therefore they are treated as similar. The recommendation is marked by the dashed arrow. An object to be recommended is chosen from the similar subject's related objects so that the subject isn't related to the object.

More formally, classic user-based collaborative filtering methods use a function $\hat{u}_R^{CCF}$ for estimating the utility as depicted in the formula 3.2. In the formula, $v_R : S \to \mathbb{R}^{|O|}$ is a function giving a subject vector to a given subject s, according the objects that are in R with s. *cos* is cosine of two vectors, as defined e.g. in [9].

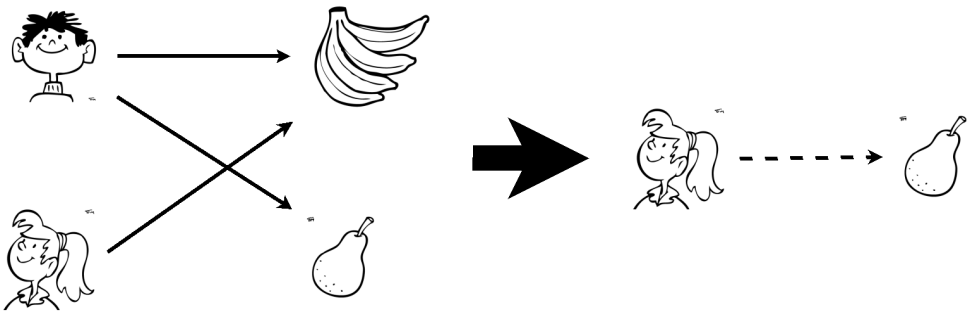


Figure 3.2: Collaborative Filtering

$$\hat{u}_R^{CCF}(s,o) = \sum_{(s',o)\in R} cos(v_R(s), v_R(s')) \tag{3.2}$$

The collaborative filtering methods can be divided into several groups. The overview of the groups can be seen in the figure 3.3, names of the techniques and algorithms belonging to the given groups are in italics.

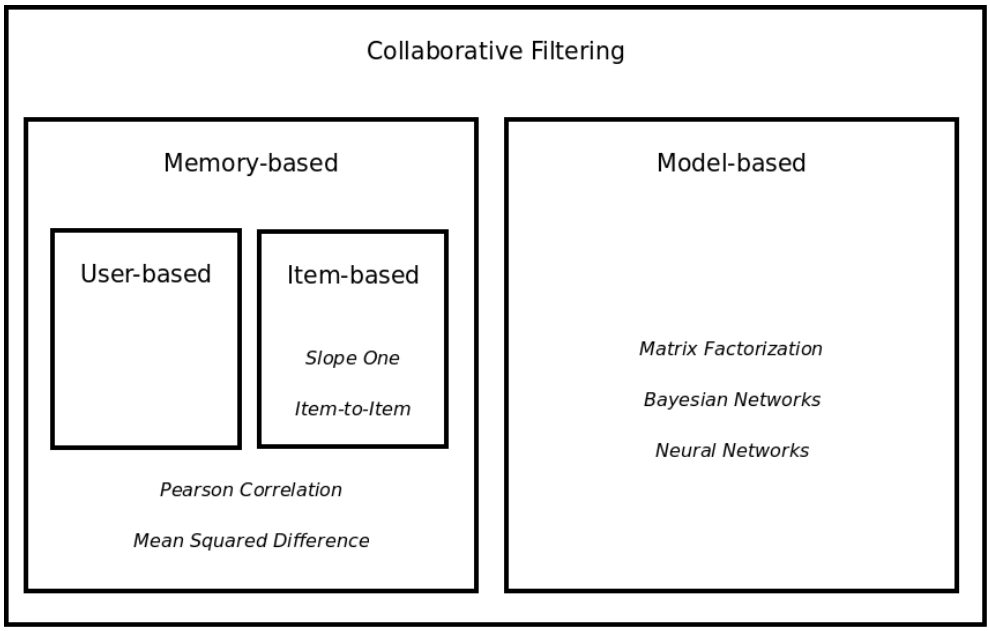


Figure 3.3: Classification of the collaborative filtering algorithms

By the manner in which the unknown relationships are predicted, collaborative filtering methods can be divided into two groups: *Memory-based* and *Model-based* [9].

3.2.1 Memory-based Collaborative Filtering

Memory based (or heuristic-based) methods are historically the first collaborative filtering methods. In memory-based prediction methods, the predictions of

possible relationships are counted as an aggregate of the known subject-object relationships. The aggregate function can be a simple average or some more sophisticated measure using relative differences to average ratings or inter-subject similarity.

Neighbourhood methods belong to this category. For recommending an object to a subject they use relationships of the neighbouring subject or object (depending on the recommender being subject-based or object-based). The neighbour is defined as the subject/object having the highest similarity to the given subject/object. The similarity can be counted in many ways, including the Pearson correlation coefficient or Cosine measure [9].

The memory-based methods can further be divided into two groups by the direction which is used for recommending: *Subject-based* or *Object-based*.

User-based Collaborative Filtering

Subject-based (or user-based) methods are centered to the subjects. In this traditional variation, the similarity of subjects is computed. More formally, according to [9], the utility function $u_R(s, o)$ of subject s, object o and relationship R is estimated based on the utilities $u_R(s_j, o)$ assigned to object o by the subjects $s_j \in S$ who are similar to subject s. Usually the similarity between subjects is determined by comparing their relationships to objects.

Item-based Collaborative Filtering

Object-based (or item-based) methods have been popularized by Amazon.com [5]. The principle of the algorithm remains the same, but the algorithm starts at the objects and similarity between objects is computed. Objects sharing the same related subjects are taken as similar. The objects recommended to a subject are taken from objects similar to those the subject is related to. The method can also be used for showing products that are related to a product that is viewed by an anonymous user, as can be seen in the Amazon.com product catalogue: "Users that bought product x also bought product y, z". The easiest implementation of the item-based memory-based collaborative filtering is the *Slope One* method.

Another way how to measure the similarity is the cosine measure, which is used in the *Item-to-Item* algorithm patented by Amazon.com [5]. The objects are represented as vectors. To count similarity between two objects, subjects that have a relationship to both of these objects are taken. Each such subject represents a dimension in the object vectors, the value is determined from the subject's rating or 0/1 (the subject has bought/viewed the object or not). The similarity between two objects is then counted as a cosine of the object vectors [5].

3.2.2 Model-based Collaborative Filtering

Model-based recommenders use machine-learning techniques to learn a *model* that predicts unknown relationships. The known relationships are used as training data for the model. The machine-learning techniques include *Bayesian networks*, *latent factor models*, or *artificial neural networks* [9].

Latent semantic models use vectors to represent subjects and objects [22]. For objects, the values in the vectors mean some characteristics of the object. This approach is similar to the one used in content-based filtering (3.1). The difference is in obtaining the vectors: the values in object vectors aren't submitted by a human expert, both subject and object vectors are learned from the known data by various techniques. Therefore the technique is domain-independent.

Subject and object vectors allow to project the subjects and objects into multidimensional space. Recommended objects are those that are "near" the given subject in the multidimensional space. Some common techniques for measuring the subject-object distance are vector cosine and dot product. Some dimensions can be coupled with some known object characteristics as genre for movies [22]. But the meaning of most of the dimensions can hardly be discovered, as they aren't designed by a human but a machine-learning technique. Consequently, the recommenders using matrix factorization usually aren't able to give reasons for their recommendations.

More formally, latent factor recommender estimates the utility by a function $\hat{u}_R^{LF}$ as displayed in the formula 3.3. $fv_R^s : S \to \mathbf{R}^f$, $fv_R^o : O \to \mathbf{R}^f$ are functions assigning each subject/object a vector of the given factor latent space dimensionality $f \in \mathbf{N}$. Vectors for the subjects and objects are obtained by a methods like *stochastic gradient descent* or *alternating least squares*, see [22].

$$\hat{u}_R^{LF}(s, o) = fv_R^s(s)^T fv_R^o(o) \tag{3.3}$$

Figure 3.4 shows some users and items projected to a simplified two-dimensional space ($f = 2$). Their nature determines their positions in the space. Items that are near each user are likely to produce a positive response from the user and hence they are good recommendations.

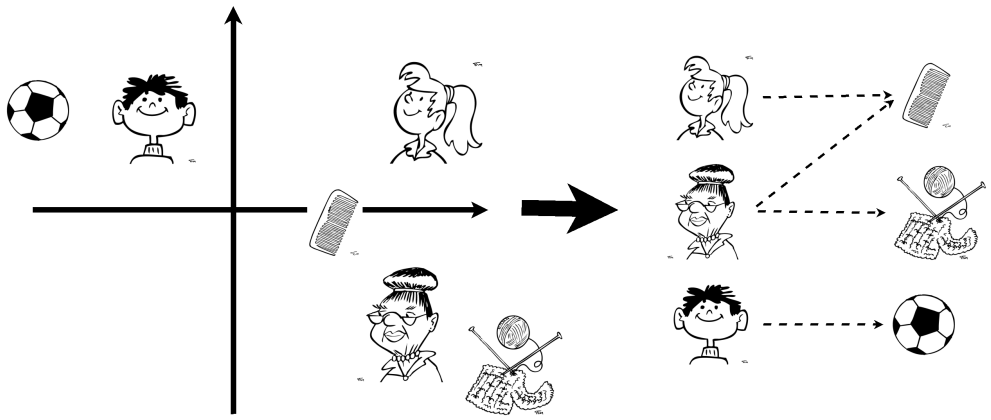


Figure 3.4: Latent factor model recommender

3.2.3 Benefits and Drawbacks

In general, collaborative filtering methods are currently the most used and the most successful methods for recommending. When properly used, they provide high accuracy and scalability for large amount of data. However they have some drawbacks too [9].

Cold start problem As for the content-based method, the system doesn't know any liked objects for a new subject. For a user being a subject, this problem can be solved by asking the user to enter some data. This approach is used in the Netflix system. A new user is given a questionnaire for rating some chosen movies [11]. Another solution to the problem is using a hybrid recommender (3.5) that would use an easier recommender (like giving the most popular objects) when predictions from collaborative filtering aren't available. A similar problem occurs when a new object is added to the system, it's not related to any subject and therefore it can never be recommended. This can be solved also by a hybrid recommender (3.5).

Sparsity The subject-object matrix is usually very sparse - the number of known relationships is very small compared to the number of relationships that should be predicted. Therefore, in most of the collaborative filtering methods, the objects related to few subjects are seldom recommended. This drawback is overcome by some of the model-based methods, e.g. matrix factorization [22].

Grey sheep problem This problem is mentioned in [23]. In the system, there might be subjects whose preferences aren't consistently similar to other subjects. Therefore they don't belong to any preference subject group and they can't get any accurate recommendations.

Suitability for The Universal Recommender

One of the biggest advantage of collaborative filtering is its domain independence. Generally, the algorithms don't assume any properties of subjects or objects, they just use the relationships between subjects and objects to make recommendations. Of course, the recommender tuned for a particular domain can't be used directly for other domain, but the core algorithm is domain-independent.

The universal recommender proposed in [19] is based on the idea of model-based collaborative filtering techniques.

The disadvantage of collaborative filtering is that it's limited to a single subject-object preference data source - usually an explicit rating. For our purposes it would have to be adapted so that it can process multiple data sources.

3.3 Knowledge-based Recommenders

Although the main focus in the recommender research field is on collaborative filtering, possibilities of knowledge-based recommenders are still being studied.

There are many approaches to knowledge-based recommending. One of them is presented in [25]. The user interaction to the system is similar as in the case of performing a search. At first, the user specifies an item that he/she likes. Then the system iteratively presents similar items to the user. The user further specifies his/her preferences, like "I would like a more romantic/adventurous movie". This continues until the user is fully satisfied with the presented item.

One problem of the approach is that the recommendations aren't personal. There are no user profiles in the system, so the recommendations are based only on data given by the user during the search. The author describes how to reduce

this disadvantage in [26]. The items found by the knowledge-based recommender are sorted by the integrated collaborative filter. Another problem of this approach is bothering the user. The user has to fully specify his preferences in order to get some good recommendations. Although the system leads the user through the search, it can take a lot of time to find the right item.

The article [23] proposes a bit different approach. Their knowledge-based recommender is trying to solve the "cold start problem" (see 3.2). When a new user registers to the system, he/she is asked to choose an example of an item he/she likes in the item catalogue. Then the user is asked to compare the item to some other items, using a scale from zero to one, determining how much the item is preferred over the other. An underlying algorithm exploits this knowledge to the whole item catalogue and uses the data for making recommendations. This approach is more usable than the first described one, as it demands the user input only once - when he/she is registering. Nevertheless new users are more sensitive to being asked a lot of questions and there's a risk, that the user leaves for a rival system. The presented recommender is only a theoretical suggestion, there might be some problems with scalability to large numbers of users and items.

Recommendation by ordering is also discussed in [27]. The authors of the article describe an advanced algorithm for exploiting the known relative preferences to the whole domain. The performance evaluations of the presented algorithm look very promising. Nevertheless, obtaining reliable relative preferences without bothering the user is difficult and requires some non-trivial domain knowledge.

Another approach to knowledge-based recommender was used in the *RACOFI* system [28]. The system consist of a collaborative filtering engine (*COFI*) and a rule applying agent (*RALOCA*). The primary recommendation is done by COFI. The rules are then used for adjusting the recommendation rate or removing objects from recommendations. The rules are applied on *objective* data we know about the subjects and objects, like age and genre respectively.

The [28] report shows that rules can handle boundary cases by removing inappropriate objects for recommendations. They can also refine recommendations by applying some rules that were found empirically. We find the idea of applying rules for recommending interesting, especially when there's not enough data for making other types of recommendation. The other benefit of rule-based recommending is the ability to give good explanations why the objects were recommended.

More formally, knowledge based methods use functions like $\hat{u}_R^{KB}$ defined in the formula 3.4 for estimating the utility. RL_{so} is a set of rules that can be applied to a particular subject-object pair (s, o), where each rule $rl : S \times O \to [0, 1]$ is a function determining utility of the object o to the subject s.

$$\hat{u}_R^{KB}(s, o) = \sum_{rl \in RL_{so}} rl(s, o) \tag{3.4}$$

In the illustration 3.5 we use a rule that uses information about the conditions where the subject is situated in order to determine the utility of the object to the subject.

According to [23], there are three types of knowledge, a recommender system can deal with:

Catalogue knowledge provides information about objects and their features.

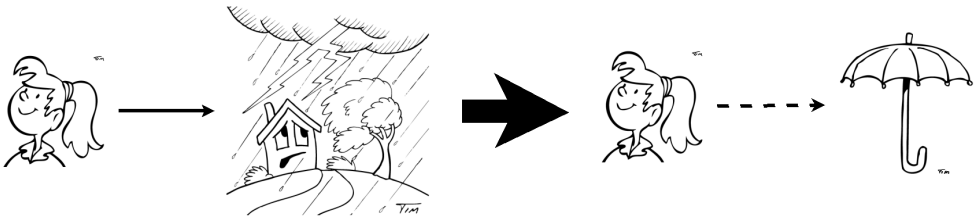


Figure 3.5: Knowledge-based Recommender

Functional knowledge provides information about how objects meet subjects' needs.

User knowledge contains information about subjects' needs.

Suitability for The Universal Recommender

Business knowledge can give an added value to a recommender, as some rules are well-known and often cannot be easily extracted from the underlying data. Therefore, if a way how to enter some basic domain-specific rules is found, the knowledge based recommender can be included into the universal recommender.

3.4 Other Domain-specific Methods

There are a lot more algorithms for recommending, but all of them rely on specific data that must be available about the subjects or objects for the recommender to work well. As these aren't acceptable for the universal recommender, we list only a few of them.

3.4.1 Social Networks and Link Prediction

In social networks, there are links between users, such as the *friends* link. If these are available, the recommender can recommend an object according on what user's friends liked. A schematic picture of the a social network recommender can be seen in the figure 3.6.

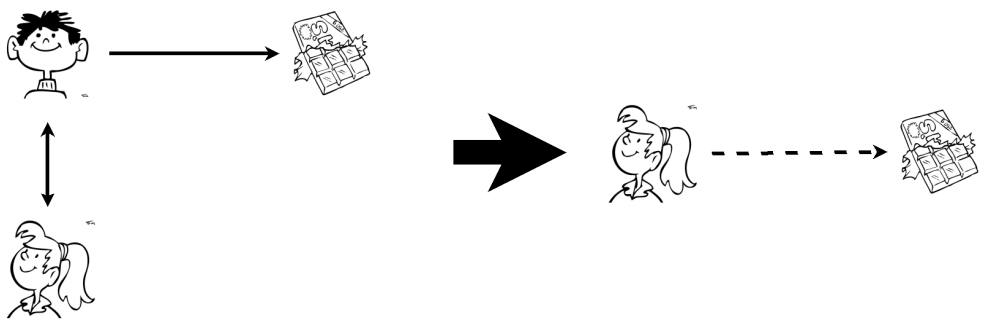


Figure 3.6: Recommending in social networks

More formally, the $\hat{u}_R^{SN}$ estimates the utility, as depicted in the formula 3.5, where $FR \subseteq S \times S$ is a symmetric friendship relation and $int : S \times S \to [0, 1]$ is a function giving an intensity of the friendship between two users.

$$\hat{u}_R^{SN}(s, o) = \sum_{(s,s') \in FR \ \& \ (s',o) \in R} int(s, s') \tag{3.5}$$

3.4.2 Demographic Filtering

When we have some additional information about users such as nationality, residence, age or occupation, we can use it for determining similarity between the users. A demographic filtering method can be used as an additional measure of user similarity when there's little known about the user's taste. An example of a recommendation made by a demographic filtering recommender can be seen in the figure 3.7.

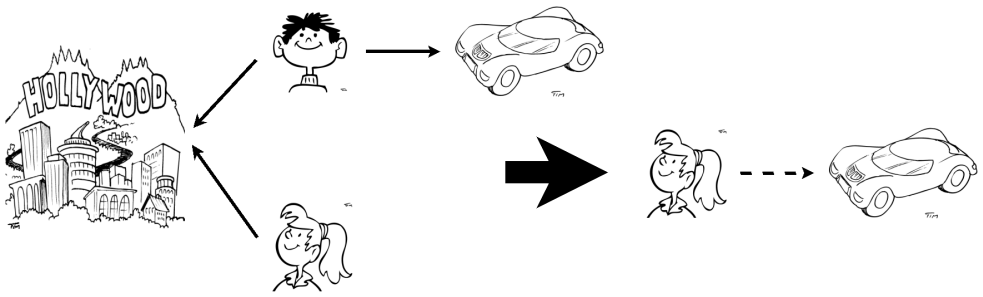


Figure 3.7: Demographic Filtering

More formally, the $\hat{u}_R^{DF}$ estimates the utility, as depicted in the formula 3.6, where $dsim : S \times S \to [0, 1]$ is a function giving similarity between users according to their demographic data.

$$\hat{u}_R^{DF}(s, o) = \sum_{(s',o) \in R} dsim(s, s') \tag{3.6}$$

Even though these methods require some specific data to be present in the domain, we would like our universal recommender to work with such data. E.g. in the domain where demographic data is available, we should be able to use them for making recommendations.

3.5 Hybrid Recommenders

Hybrid recommenders combine two or more recommending methods in order to improve the produced recommendations. There are several ways how to combine them.

For enhancing the success of produced recommendations, we can use a hybrid recommender that uses all its underlying methods when producing any recommendation. The combined methods can be of various types, using various relationships among subjects and objects. A combination of content-based and

collaborative filtering is quite popular, possibilities of combining the two methods into a single hybrid recommender are listed in [9]. The combination of memory-based and model-based collaborative filtering is often used in contemporary commercial recommenders, as in [4]

A hybrid recommender can help to overcome some drawbacks of the recommendation methods, as the *Cold start problem* (see 3.2.3). In this case, when little data is known about a new subject and object, some subsidiary recommendation method can be used. This subsidiary method wouldn't be suitable for performing the most of the recommendations, but it can make the best of the little data that is available. This approach is used in the default strategy of the Duine recommender framework [7].

More formally, a linear weighted hybrid recommender, as described in [24], uses $\hat{u}_R^H$ for estimating the utility, as depicted in the formula 3.6. U is a set of functions belonging to the contained recommenders, $\hat{u'}_{iR} \in U$ for $i = 1, \ldots |U|$. $w_i \in [0, 1]$ is a relative weight of the utility function $\hat{u'}_{iR}$.

$$\hat{u}_R^H(s, o) = \sum_{i=1}^{|U|} \hat{u'}_{iR}\ w_i \tag{3.7}$$

Figure 3.8 shows a recommendation for a new user, performed by a hybrid recommender combining social links and collaborative filtering.

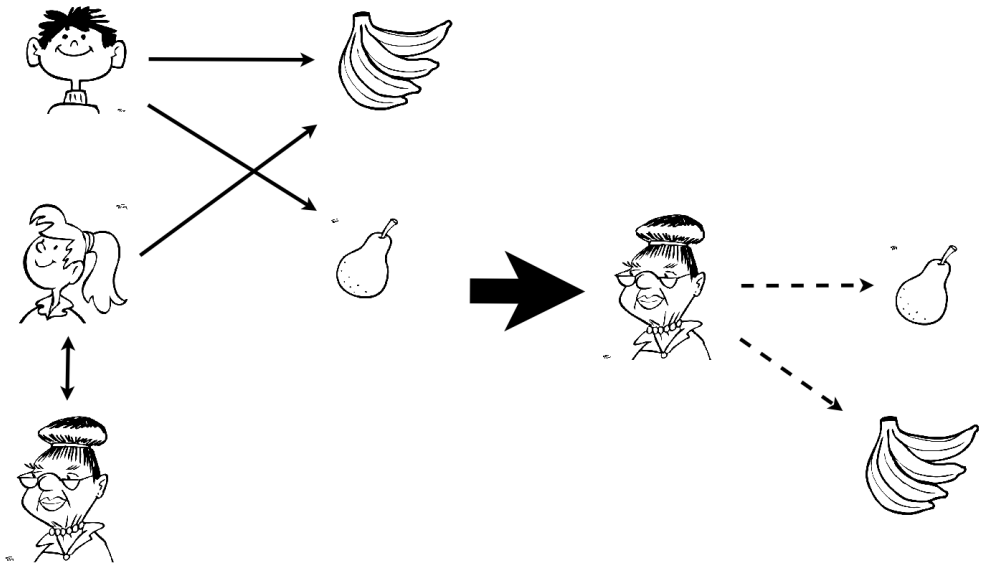


Figure 3.8: Hybrid Recommender

Suitability for The Universal Recommender

As presented in [19], the specific combination of methods used in a hybrid recommender depends on the data available in the given domains. Therefore a hybrid recommender can't be generic. Finding the right combination of recommenders for the specific domain can be non-trivial. However a hybrid recommender composed of two domain-independent methods can be used.

3.6 Algorithm Selection

In the chapter we have described all common contemporary approaches to recommender systems, their suitability for domain-independent recommending has been evaluated. In a universal recommender we cannot assume that any specific relationships or subject and object properties exist, so our choice narrows a lot.

Our vision of using business rules for recommending intersects with the knowledge based recommender. 3.3. However there will have to be a domain-independent mechanism for representing and combining the business rules.

From the studied algorithms, the only one that is domain-independent by nature is *collaborative filtering*. All the others are dependent on some specific features of subjects or objects, or they suppose some specific relationships. Therefore, our universal recommender should be based on a collaborative filtering algorithm.

We have noted, that there's a gap between our requirements for a universal recommender system and collaborative filtering algorithms: We would like to make use of multiple data sources including object and subject similarity, and domain specific rules. The collaborative filtering in its typical implementation takes only a single source of data - subject-object rating. The gap between our requirements and the input of a collaborative filtering algorithm should be filled by the Unresyst application. It should process all the inputs it has, into a single subject-object preference prediction. This prediction can be then used directly for recommending or it can serve as an input for a common collaborative filtering algorithm.

There are a lot of types of collaborative filtering algorithms, as can be seen in the 3.2 section. The algorithms differ in many aspects as time complexity, accuracy of the predictions or possibility to reflect the updates in the domain. Each domain has its own requirements on the named aspects and therefore Unresyst shouldn't be bound to a particular algorithm. Rather than that it should be able to use an arbitrary collaborative filtering algorithm in its algorithm layer. When implementing a recommender to a system we should be able to choose whether to use the predictions directly for recommending or to choose the right algorithm to fit the domain properties and needs.

3.6.1 Combining Knowledge-based and Collaborative Filtering Recommenders

The only method that wasn't generalized by the universal recommender proposed in [19] is the knowledge-based method. As we have shown in 3.3, simple domain-specific rules can enhance the recommendation accuracy, especially when there's little preference data available. Hence, incorporating a possibility to enter some simple domain-specific set of rules when adapting the universal recommender to a domain, would be interesting. We found a few options how to merge the evaluation of knowledge rules with collaborative filtering:

A hybrid recommender for knowledge-based and collaborative filtering

In this variation there are independent knowledge based and collaborative filtering recommenders. This approach was used in the RACOFI system [28]. For a rela-

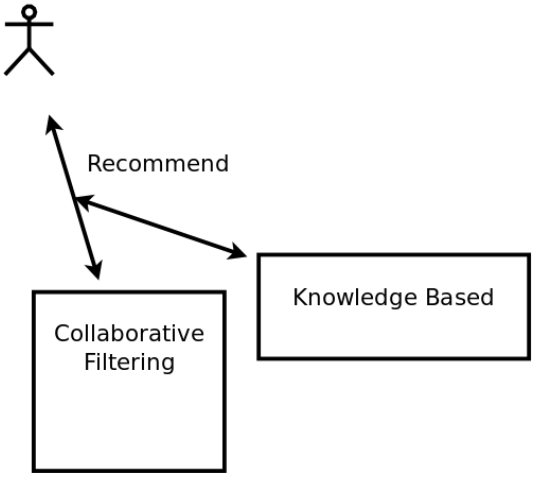


Figure 3.9: Combining Collaborative Filtering and Knowledge Based Recommender as a Hybrid Recommender

tionship prediction, if a rule is available, it is used for predicting the relationship together with the underlying collaborative filtering algorithm. If not, only the collaborative filtering is used. Although this is a possible combination of the recommenders it has some significant drawbacks. The method introduces a tension between the two recommenders. The results of the collaborative method, which should be universal are in some cases distorted by the knowledge recommender. This problem would lead to incorrect learning in the model-based algorithms, as some manipulations are done outside the model and therefore the method cannot fully affect the resulting predictions.

More formally, in this approach we use a function $\hat{u}_R^{HKC}$ for estimating the utility, as depicted in the formula 3.8. $\hat{u}_R^{HKC}$ is a composition of a knowledge based recommender utility estimation function $\hat{u}_R^{KB}$ and a collaborative filtering utility estimation function $\hat{u}_R^{CF}$, which can be replaced by any of the functions related to algorithms mentioned in the section 3.2.

$$\hat{u}_R^{HKC} = \hat{u}_R^{KB} \circ \hat{u}_R^{CF} \tag{3.8}$$

Using the knowledge-based predictions as an input for collaborative filtering

Alternatively, using the business rules given during the adaptation to a specific domain, we can generate a prediction (a kind of rating) that is later used in the underlying collaborative filtering algorithm. The prediction is available for subject - object pairs, for which some rules can be applied. The rules are only used during the initial recommender build to provide an input for the collaborative filtering. The recommendations then are done solely by the collaborative filtering algorithm. The advantage of the approach is, that the information from the knowledge-based algorithm is available in the initial phase and can be used for the collaborative filtering model build.

More formally, in this approach we use a function $\hat{u}_R^{KIC}$ for estimating the

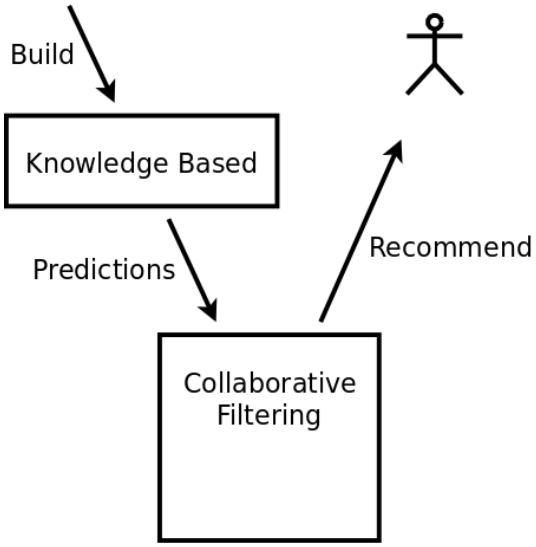


Figure 3.10: Using predictions from Knowledge based recommender as an input to Collaborative Filtering

utility, as depicted in the formula 3.9. $\hat{u}_R^{KIC}$ uses a function $\hat{u}^{CF}$, which can be replaced by any of the functions related to algorithms mentioned in the section 3.2. Instead of using bare predicted relationship R it additionally uses $R^{KB} \subset S \times S$, a relation obtained by applying the rules of the knowledge-based recommender to subjects $s \in S$ and $o \in O$. Formally $R^{KB} = \{(s,o)|\exists rl \in RL_{so}\}$, where RL_{so} is a set of rules that can be applied to the subject object pair (s,o), defined in the section 3.3.

$$\hat{u}_R^{KIC} = \hat{u}_{R \cup R^{KB}}^{CF} \tag{3.9}$$

For the given reasons we choose the second option. Another advantage of this approach is that the recommendation time will be dependent only on the recommender algorithm. The implementation of the chosen approach is thoroughly described in the chapter 5.1.

4. Relationships in Studied Systems

In the chapter we analyze selected systems where the Unresyst recommender could be used. As we don't have a direct access to the system databases, we use data sets[1]. The first two data sets - Last.fm and Flixster are publicly available. The third one (Czech travel agency e-shop data set) was made accessible to the thesis author by the courtesy of the system administrator, who didn't wish to disclose his name.

4.1 The Last.fm Dataset

Last.fm is a music recommendation service [1]. The system users post the information about what tracks they're listening to through plugins installed in their music players. Last.fm provides a freely available API[2] for developers who can build applications using Last.fm data.

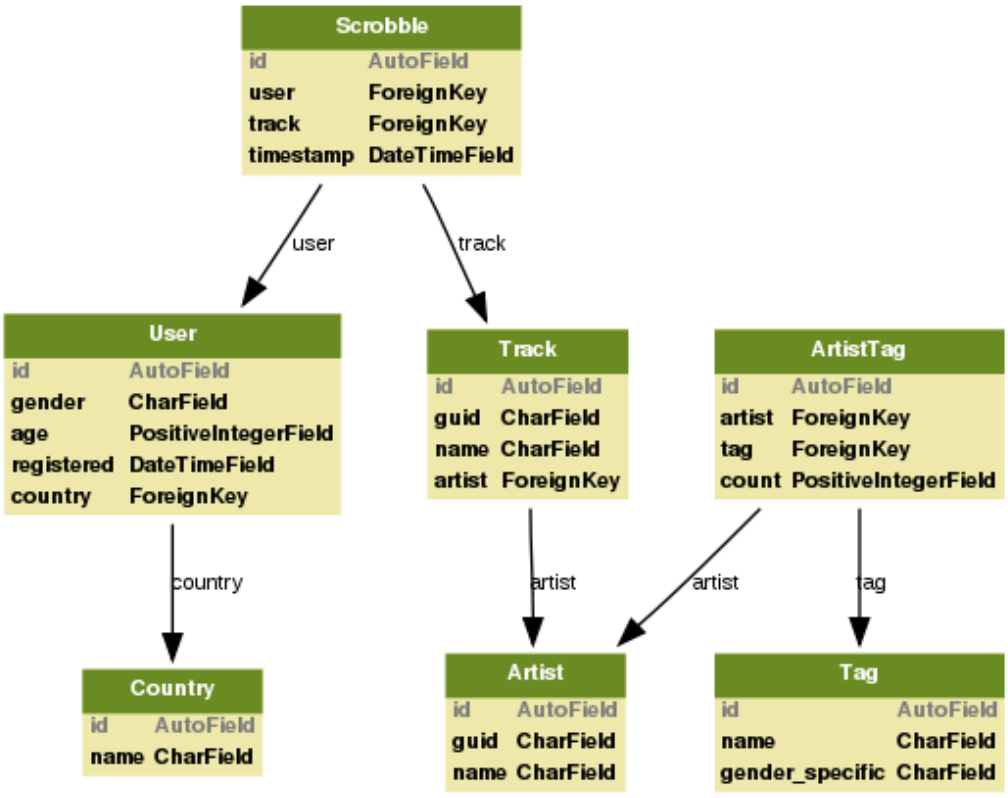


Figure 4.1: The data model of the Last.fm data set

The API was used for crawling the data sets we used for our study. The first

[1]Data set is a set of text files containing some part of data included in the system database
[2]See `http://www.last.fm/api`.

used data set [29] contains users and their listening history. Each user has listened and posted (*scrobbled*) some tracks, each track was recorded by an artist. For some of the users there's some basic personal information available (age, gender and home country), each scrobble has a timestamp.

The data set was integrated with the Last.fm social tag data set [30]. Last.fm users can tag artists, describing the genre the artist is playing (like *Pop*, *Rock* or *Techno*), the feeling of the music (like *slow*, *loud*) or any other property. For some of the artists we were able to assign tags, the users gave the artist.

The mentioned data sets were selected because together they contain basic attributes of both subjects and objects, which can be used for similarity measures. It contains implicit feedback (the scrobbles) that discloses some positive preferences of the users. The data model of available data is displayed in the figure 4.1

4.2 The Flixster Data Set

Flixster is a community server for sharing movie reviews and ratings[3]. Each contained movie has a profile page where registered users can comment and rate the movie. Users can become friends with each other. The Flixster data set [31] doesn't contain any properties for movies nor for users. The only data available is the user-movie rating and links between users who are friends. The data model is displayed in the figure 4.2.

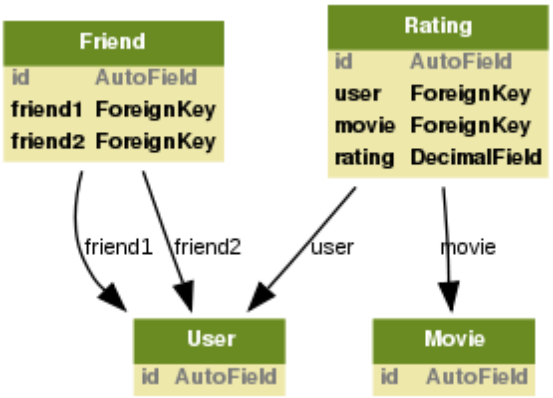


Figure 4.2: The data model of the Flixster data set

The data set is a of a type that is often used for collaborative filtering research as it contains an explicit user-item feedback that can be predicted. In the data set there is additional social information - the links between friends. A version of the data set was used for a research on trust propagation in social networks [32].

[3]See `http://www.flixster.com/`.

4.3 The Travel Agency Data Set

The travel agency data set was collected by an administrator of a travel agency website. During a few months he was recording different kinds of implicit feedback through a script. Users were browsing through the website, clicking on links, viewing the tour profiles, moving over elements on the web pages. Some of the users asked questions about tours, through an online form. Some users ordered tours from the website. The tours have some basic properties: the country and the type (tours to hotels at the sea, tours visiting interesting places around the country, etc.).

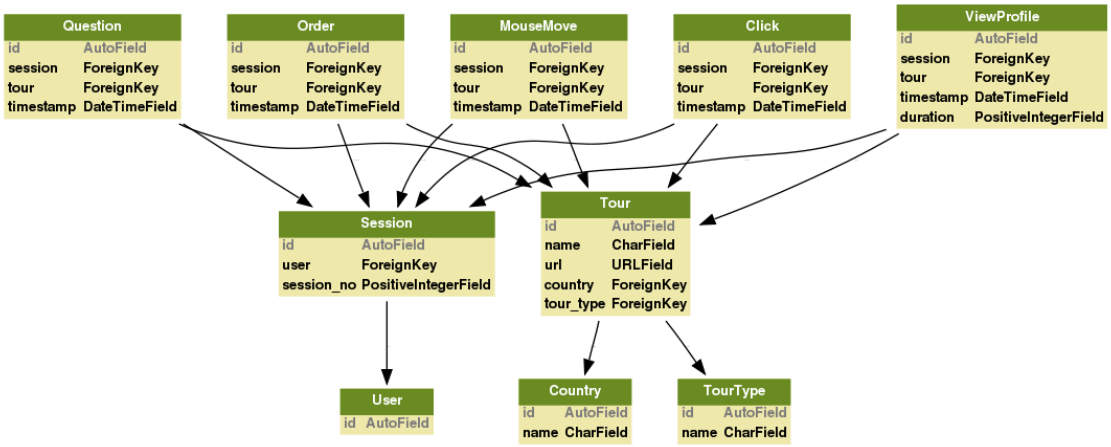


Figure 4.3: The data model of the Travel Agency data set

The data set contains different kinds of implicit feedback that is to be combined in order to provide good recommendations. Moreover the objects (tours) have some properties that can be used for determining similarity.

4.4 Summary

The studied systems were selected so that they cover a variety of different domains, where we can recommend. The data sets fulfill the recommender applicability conditions described in the section 2.1. The systems were selected so that we can prove the usefulness of the Unresyst recommender concept for processing both implicit and explicit feedback. Some of the data sets have attributes for their subjects and objects, that can be used for similarity measures.

The rules, relationships and biases used in Unresyst for the described data sets are presented in the Chapter 6.

5. Universal Recommender Design and Implementation

The chapter describes how the Universal Recommender was implemented. Firstly we give an overview of the architecture, then we describe the interfaces through which Unresyst communicates with the parent system. Then we describe the algorithm used for applying rules. High-level architecture of the Universal Recommender, the layers of the application, are studied in the following section. Finally we give a more detailed overview of the inner structure of Unresyst.

5.1 Overview of the Universal Recommender Architecture

The Universal Recommender System (Unresyst) is an independent application working with database. The parent system uses Unresyst for creating recommendations through a set of interfaces.

The Universal Recommender is designed so that it's independent of the domain, working only with abstract relationships. Several instances of the recommender can be run on a single system, so the Universal Recommender can run with various configurations for one parent system.

The recommender doesn't have to be directly a part of the parent system, its database is logically independent of the parent database system. The parent system and the recommender communicate only through the interfaces. There are no implicit links on the application nor on the database level. Hence the recommender could be easily run on an independent server.

Unresyst provides two interfaces to the parent system: the *Adaptation Interface* and the *Runtime Interface*. The Adaptation Interface is used only during the system setup and maintenance. See the section 2.3.3 for a more detailed description of the actions taken during the adaptation. On the other hand, the Runtime Interfaced is used continuously by the parent system to provide recommendations.

The figure 5.1 shows how Unresyst can be adapted to a domain on the example of the music domain. In the example we create two recommenders for a domain. A domain expert firstly creates the desired recommenders. In the example it's a *Novel Artist Recommender* and a *Artist Radio Recommender*. Both recommenders suggest artists to listeners. The Novel Artist recommender tries to broaden listener's music horizons by suggesting artists the listener haven't heard yet. Whereas the Artist Radio Recommender suggests artists to a listener no matter if he/she has already heard it. Such a recommender can be used for a personalized internet radio.

Each recommender has two interfaces: *Adaptation Interface* and *Runtime Interface*. The domain expert uses the Adaptation interface to adapt Unresyst to the music domain. The adaptation is done by defining basic recommender properties and a set of rules. The design and the usage of the Adaptation Interface is later described in the section 5.2.1.

In the figure 5.2 there is a schema of how a system that is using Unresyst

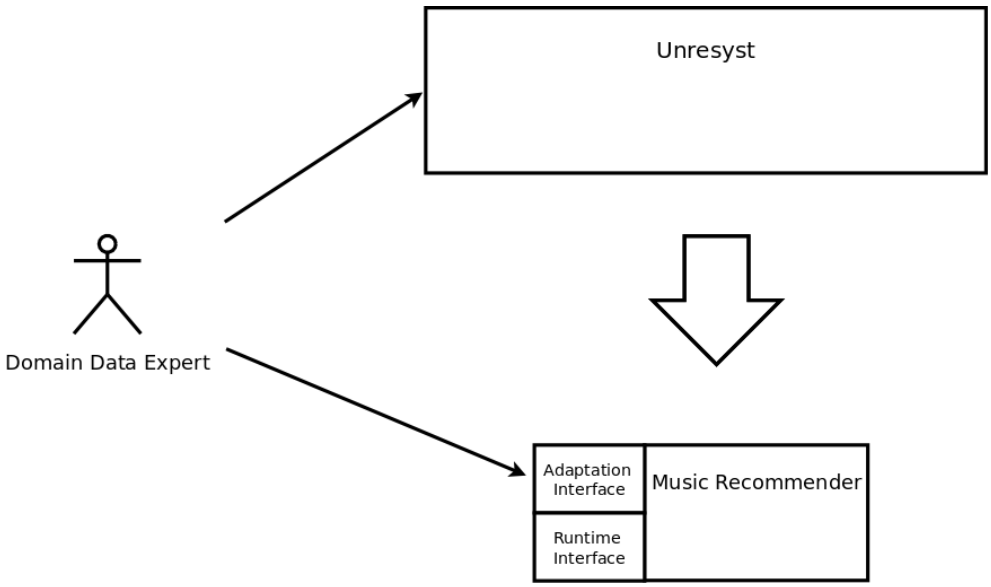


Figure 5.1: Adapting Unresyst to the music domain done by a domain expert.

operates. The user accesses the system as usually through its graphical interface (GUI). In the interface the parent system shows the recommendations obtained by using Unresyst. The parent system uses the Runtime Interface of both recommenders to get recommendations and to provide information needed for making recommendations. Both recommenders transfer the calls to the Unresyst module. The design and usage of the Runtime Interface is later described in the section 5.2.2.

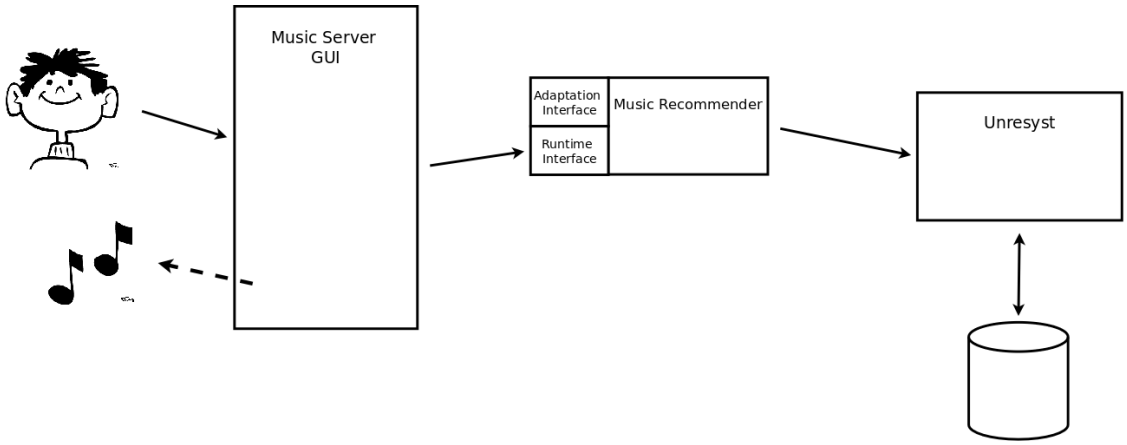


Figure 5.2: Unresyst common usage.

5.2 Unresyst Interfaces

In order to make Unresyst independent of the parent system we have defined interfaces through which the parent system and Unresyst communicate. Unresyst

has a set of two interfaces: the Adaptation interface and the Runtime interface.

5.2.1 Adaptation Interface

A well defined adaptation interface is critical for the success of the recommender. The interface has to be general enough to cover most domains and data sources they contain. At the same time it has to be clear and easy to use.

The interface is used in the *adaptation* phase of Unresyst setup. All initial data that is needed by Unresyst to provide recommendations are passed through this interface. Through the adaptation interface the following parts are given:

Subjects and Objects When adapting Unresyst to a domain, we have to define what and to whom we would like to recommend. In our implementation, this is done through an ORM Manager class covering the tables where subjects and objects respectively are stored. This corresponds to defining the O and S domains for the recommender (see section 2.4). In our example we use managers pointing to the *User* and *Artist* table.

The Predicted Relationship We have to define which relationship in the parent system data model is the preference. This is done in the way described in the section 5.2.1. This corresponds to defining the R relation in the recommender (see section 2.4) In our example we use the relationship "User played Artist's track" as the predicted relationship.

Rules and Relationships that influence recommendations When we wish to use business rules or relationships for making recommendations we have to define them through the adaptation interface. The way to define rules and relationships is described in section 5.2.1. This corresponds to specifying the RL set of a knowledge-based recommender (see section 3.3). An example of a subject-object rule is the mentioned "Music having the given tags shouldn't be recommended to female users". An example of an object similarity rule influencing the prediction is "Artists sharing tags are similar". An example of a subject similarity relationship is the mentioned "Users coming from the same country are similar". The difference between a rule and a relationship is the following: For rules we'd like to give a specific strength for each covered pair, i.e. for artists sharing some tags we can give a percentage of how much tags they're sharing. Whereas in relationships, we only know there's some connection, that can't be weighted for the particular pair, i.e. if the users are coming from the same country, they are similar without any further strength differentiation for different pairs of users coming from the same country.

Biases Some objects are more likely to be liked than others, independently of the subject to whom we're recommending - these are called object biases. Subject biases indicate a higher tendency for a subject to like an object, independently of the particular object. Such facts can be included to recommendations by defining biases. More about representing biases can be found in section 5.2.1. An example of a positive object bias is the mentioned "Artists that have released a new album recently".

In our implementation, the adaptation interface is used by deriving a class from the predefined Unresyst *Recommender* class. In the example we define an *ArtistRecommender* class, that holds all domain specific data. The following piece of *Python* code shows the definition of the music recommender from the example. Subjects of recommending are system users, recommended objects are artists. The definition of the rules, relationships and biases is discussed in the sections that follow.

```
from unresyst import Recommender
from models import User, Artist

class ArtistRecommender(Recommender):
    """A recommender recommending artists (musicians) that
    the user can like.
    """

    name = "Artist Recommender"
    """The name"""

    subjects = User.objects
    """The subjects to who the recommender will recommend."""

    objects = Artist.objects
    """The objects that will be recommended."""
```

Representing Rules

In the section we discuss the possibilities of implementing the rule representation, then we list the properties of the representing class. Finally we present a code of the rules of our example recommender for our prototype implementation

During the adaptation phase, the domain expert can introduce business rules that will be used for creating recommendations. As in [28], our rule consists of a *condition* determining when the rule should be applied, and an *action* defining what should be done if the condition is satisfied. The representation should use an easy human-editable notation, so that the rules can be easily added and later edited. Opposed to business relationships, for rules we have some kind of strength that is specific for each covered pair, i.e. for artists that share some tags we have a percentage of shared tags.

The RACOFI framework [28] uses two types of actions: *Modify* that modifies the predicted subject's object rating by a given constant and *Not Offered* that excludes an object from subject's recommendation.

Our rules aren't modifying predictions, but they serve as an input for the predicting algorithm (see the section 3.6.1 for the reasons why this way of integrating rules to collaborative filtering was selected). Therefore we will use absolute values, not relative values as in RACOFI *Modify* rules. The *Not Offered* rule type can be represented by a rule giving minimal value, so we don't need to have a special rule type for it.

In our rules we would also like to represent inter-entity similarity. Hence we will use the following types of rules:

1. *Subject-object rules* defining subject-object interactions (preferences) that can be used for recommending objects to subjects. We denote RL_{pref} a set of subject-object rules for a recommender. An example of a negative subject-object rule is the mentioned "Music having the given tags shouldn't be recommended to users of the given gender".

2. *Subject-subject rules* defining similarity between subjects. We denote RL_{ssim} a set of subject similarity rules for a recommender. An example of a positive similarity rule is "Users of similar age are similar".

3. *Object-object rules* defining similarity between objects. We denote RL_{osim} a set of object similarity rules for a recommender. An example of a positive similarity rule is the mentioned "Artists sharing some of their tags are similar".

There are several ways how to represent rules. The first studied one is *RuleML (Rule Markup Language)*[1], a XML-based markup language for representing rules. The language is very powerful, having the same expressiveness as declarative programming languages[2]. RuleML was used for entering rules into the RACOFI recommender system [28].

Creating and editing rules in RuleML isn't very convenient as creating and editing any other XML-based document. That is partially solved by a shortened notation [33], but even that isn't very comfortable to a user that isn't experienced in declarative programing. Another drawback of using RuleML for our purpose is its generality. The set of conditions and actions we'd like to represent is very limited and therefore we don't need such a powerful language.

Another drawback of such a general notation is the efficiency of evaluating rules. The classical substitution of all possible pairs to the condition wouldn't be feasible for larger domains. See 5.5 for details on how evaluating rules was implemented in a more efficient way.

Another option would be to directly use a logic programming language as *Prolog*. This option has the same drawbacks in over-generality and user-unfriendliness as RuleML. Additionally we would have to incorporate a logic programming language interpreter to Unresyst. The following lines of code show our example preference rule written in the Prolog declarative language. The code is meant only as an example and shows the possible actions without defining all necessary predicates. The example doesn't consider the strength of the rule conclusion.

```
% don't recommend artists with male-specific tags to females
dont_recommend(U, A) :-
    user(U),
    artist(A),
    female(U),
```

[1] See `http://ruleml.org/`.

[2] There are several converters between the *Prolog* programming language and RuleML, e.g. W^4 *RuleML Compiler*, see `http://centria.di.fct.unl.pt/~cd/projectos/w4/ruleml/index.htm`.

```
artist_tagged(A, T),
male_specific(T).
```

An option better fitting our needs is to implement our own *Object-oriented rule representation*, where we represent rules as instances of a rule class. In the adaptation phase, the domain expert instantiates the predefined classes to create business rules.

The *expectancy* concept (2.4.3) is powerful but not very intuitive for defining the impact of the rules on predicting relationships. By defining a rule with expectancy we would be able to indicate positive as well as negative impact on predicting the given relationship because low values of expectancy have negative impact on preference and high values positive. Defining directly the expectancy of a rule would result in complicated functions and could lead to confusion and misinterpretation. Hence, for rule definition, we use a less general but more intuitive concept of *confidence* and *positiveness*.

Each rule is either defined as *positive* or *negative*. Positive rules increase the prediction of the given relationship, negative rules decrease it. Each rule additionally has a confidence function defining how strong (positive resp. negative) the rule is for the given pair of entities. Events that can have both positive and negative impact should be represented by two rules.

Another advantage of using confidence and positiveness is a more intuitive approach to defining positive and negative rules, as the recommender uses only the part (positive or negative) that we have defined. I.e. the negative rule of not recommending a specifically tagged artists to female listeners doesn't necessarily mean we would like to recommend these artists to all male listeners (positively). If we define the negative rule, only the negative part will be taken. If we wanted to incorporate the positive part too, we would define a new positive rule.

In the examples, we will be predicting the *User listens to artist's tracks* relationship, representing the preference of the user to the given artist.

For our object-oriented rule representation we define a class which instances hold conditions, actions and some additional data. As we have showed, the only possible actions are determining the confidence of the subject-object relationship and determining inter-entity similarity. Therefore actions will be represented as functions giving a confidence of the preference or similarity. The rule objects contain:

Condition A boolean function determining whether the rule can be applied to a given entity pair. The function is defined depending on the rule type:

- **Subject-object condition** $C_{pref} : S \times O \to \{False, True\}$ for rules from RL_{pref}. A function takes a subject-object pair and determines whether the rule can be applied to the pair. In our example 1 the condition function is: If the user s is a female and artist o is tagged by one of the given tags, the condition evaluates to *True*. Otherwise it's *False*
- **Subject-subject condition** $C_{ssim} : S \times S \to \{False, True\}$ for rules from RL_{ssim}. A function takes a subject-subject pair and determines whether the rule can be applied to the pair. In our example 2 the condition function is: If we know the age of both of the users, the condition is *True*, otherwise it's *False*.

Object-object condition $C_{osim} : O \times O \rightarrow \{False, True\}$ for rules from RL_{osim}. A function takes an object-object pair and determines whether the rule can be applied to the pair. In our example 3 the condition function is: If the artists share some tags, the condition is *True*, otherwise it's *False*.

Is Positive A boolean constant (*True* or *False*) for determining whether the rule is positive for the predicted relationship or negative. I.e. the rules 2 and 3 are positive, the rule 1 is negative.

Confidence A function determining the strength of the subject-object preference in means of the predicted relationship or subject-subject/object-object similarity in a positive or negative way.

Subject-object relationship confidence $D_{so} : S \times O \rightarrow [0, 1]$. A function takes a subject-object pair and gives a confidence, that the *predicted relationship* appears (when positive) or doesn't appear (when negative). High values mean strong confidence. Low values mean that the rule is unsure about the impact (positive or negative) on the preference. This makes a similar effect as if the condition C wasn't satisfied for the subject-object pair. In our example 1 we can use a linear confidence function. The more of the given tags the artist has, the higher is the resulting confidence.

Subject-subject similarity $D_{ss} : S \times S \rightarrow [0, 1]$. A function takes a subject-subject pair and determines similarity between the subjects. The higher the confidence, the more similar (or dissimilar if negative) the subjects are, according to the rule. In our example 2 we use a confidence function that returns a normalized difference between the age of the two users.

Object-object similarity $D_{oo} : O \times O \rightarrow [0, 1]$ A function takes an object-object pair and determines similarity or dissimilarity between the objects. The higher the confidence, the more similar (or dissimilar when negative) the objects are, according to the rule. In our example 3 the confidence function returns the number of common tags normalized by the overall number of tags.

Weight $w \in [0, 1]$ A constant between 0 and 1 determining the strength of the rule. Weight, unlike confidence, is independent of the particular entities on which the rule is evaluated. The weight should be set so that the more significant rules get higher weight. The exact weight values should be verified experimentally.

Description A text explaining the meaning of the rule. Descriptions can be used when presenting recommendations to users. In the text there can be gaps that that will be filled with a textual representation of entities. Our example rule 2 can have a description "The users {subject1} and {subject2} are similar because their age is similar."

The following pseudo-code, shows evaluating a rule $r \in RL_{pref}$ for a subject s and object o. If the condition is evaluated to $True$, the confidence is counted

and positive or negative preference is stored according to the positiveness of the rule.

```
If r.condition(s, o) == True Then

    conf := r.confidence(s, o)

    If r.is_positive == True Then
        save_positive_preference(s, o, conf, r.weight)
    Else
        save_negative_preference(s, o, conf, r.weight)
    End If

End If
```

Let's show a representation of our example rule $r_1 \in RL_{pref}$ *Don't recommend male-specific artists to females.*

The formula 5.1 depicts the condition of the example rule. $F \subseteq S$ is a set of female subjects, $MST \subseteq T$ is a set of male specific tags, a subset of the set of tags T and $AT \subseteq O \times T$ is a relation between artists and tags, meaning the artist was tagged by the given tag.

$$C_{pref}^{r_1}(s,o) = \begin{cases} True & \text{if } s \in F \text{ and } \exists t \in MST, (o,t) \in AT \\ \\ False & \text{otherwise} \end{cases} \tag{5.1}$$

The formula 5.2 depicts the confidence function of the rule. The confidence is the proportion of the male-specific tags to all tags given to the artist.

$$D_{pref}^{r_1}(s,o) = \frac{|\{(o,t)|\ t \in MST\}|}{|\{(o,t)|\ t \in T\}|} \tag{5.2}$$

The following lines of code show the example rule $r_1 \in RL_{pref}$ written in our Python prototype recommender implementation. Weight is set to a chosen constant, it can be replaced by a more appropriate one as described in the section 7.3.4. The character sequences `%(object)s` and `%(subject)s` are placeholders for the name of the object and subject respectively. When presenting recommendations to a user, these placeholders are filled with the actual object resp. subject name.

```
from unresyst import *
from models import *

AGE_DIFFERENCE = 38 - 17
"""The age difference between the oldest and the yongest user"""

class ArtistRecommender(Recommender):
    """A recommender recommending artists (musicians) that
    the user can like.
```

```
"""

# ...

# the class contains definitions for business rules
rules = (

    # don't recommend artists with male-specific tags to females
    SubjectObjectRule(
        name="Don't recommend male music to female users.",

        # the user is a female and the artist was tagged by
        # a male-specific tag
        condition=lambda user, artist: user.gender == 'f' and \
            artist.artisttag_set\
            .filter(tag__gender_specific='m').exists()

        # it's a negative rule
        is_positive=False,

        weight=0.5,

        # the more male-specific tags the artist has, the higher
        # is the rule confidence. Normalized by the artist tag
        # count
        confidence=lambda user, artist: float(
            artist.artisttag_set\
                .filter(tag__gender_specific='m').count())/ \
                artist.artisttag_set.count(),

        description="Artist %(object)s isn't recommended to" + \
            " %(subject)s,  because the artist" + \
            " is considered male-specific."
    ),
)
```

Such rules can represent each of knowledge types described in [23], described in the section 3.3. Catalogue knowledge can be represented by rules accessing object properties in their Condition and Confidence functions. Functional and user knowledge can be representing by rules using the relationships between subjects and objects.

Representing Relationships

Apart from defining business rules, the domain expert can define which relationships are relevant for performing the prediction. This is done when adapting a system to Unresyst.

In relationships, unlike rules, we only know there's some connection, that can't be weighted for a particular entity pair, i.e. if the users are coming from the same country, they are similar without any further strength differentiation for a pair of users coming from the same country.

We would like to represent relationships of the following types. The relationships don't have to be direct - on the path from subject/object to subject/object there can be an entity of any type.

As for the rules, there can be both positive and negative relationships. Positive relationships increase the subject-object preference or similarity for the relevant entity pairs, negative relationships decrease it.

As for the rules, we have the following relationship types

Subject-object relationships indicate a subject's preference for the given object in means of the predicted relationship.

Subject-subject relationships indicate similarity among subjects. In our example, we have a relationship "Users that are from the same country are similar". The relationship is positive, as it increases the resulting similarity.

Object-object relationships indicate similarity among objects.

Obviously, the requirements on the relationship representation are very similar to the requirements we have set on rule representation. Hence, we can use the rule representation for relationships with some corrections:

Condition The boolean function means whether the given subjects/objects are in the relationship. The forms of the function remain the same as described in the section 5.2.1. In our example, the condition function is *True* if both users have filled the country they're from and they are from the same country, otherwise it returns *False*.

Is Positive: A boolean constant for determining whether the relationship is positive for the predicted relationship or negative. Works analogously to the Is Positive value in rules.

Confidence The function won't be needed for representing relationships. If the Condition function returns $True$ for a subject/object pair the pair surely is (when positive) or isn't (when negative) in a relationship. Hence, such a function would always return 1 for relationships.

Weight, Description The properties an analogous meaning to rule weight, see 5.2.1

The relationship evaluation is the same as for rules, just the confidence isn't counted.

Let's show the relationship representation on our example relationship $r_2 \in RL_{ssim}$ *If two users are from the same country, they are similar.*

The formula 5.3 depicts the condition function of the rule r_2, where *country* : $S \to C \cup \{\varnothing\}$ is a function that gives a country from the set C to a given user, or $\varnothing$ if the user hasn't filled his/her country.

$$C^{r_2}_{ssim}(s_1, s_2) = \begin{cases} True & \text{if } country(s_1) \neq \varnothing \text{ and } country(s_2) \neq \varnothing \\ & \text{and } country(s_1) = country(s_2) \\ False & \text{otherwise} \end{cases} \tag{5.3}$$

The following lines of code show the relationship r_2 written in our Python prototype recommender implementation.

```
from unresyst import *
from models import *

class ArtistRecommender(Recommender):
    """A recommender recommending artists (musicians) that
    the user can like.
    """

    # ...
    relationships = (
        # if two users are from the same country, they are similar
        SubjectSimilarityRelationship(
            name="Users living in the same country",

            # both users have given their country and it's the same
            condition=lambda user1, user2:
                user1.country and \
                user2.country and \
                user1.country == user2.country,

            # it's relationship positive to similarity
            is_positive=True,

            weight=0.5,

            description="Users %(subject1)s and %(subject2)s" + \
            " are from the same country.",
        ),
    )
```

Such a representation can use semantics of a system independently of its data model. There doesn't have to be a direct link between subjects/objects in the data model to define a relationship. Even though there isn't a direct connection in the data model, we can represent such a relationship in our system. Moreover, the relationships can be taken from various sources, e.g. multiple databases, data stores, or log files. The only thing that has to be provided to Unresyst is an implementation of a function returning whether the given subjects/objects are in the relationship.

This representation is a reduction of the full semantic representation used in [19]. The authors propose using all system entities and relationships for recommending. In this approach there are significant challenges. Firstly, it's not clear how to assign weights for relationships between entities that are not subjects or objects. Such relationship can have a varied meaning in the context of different relationships. In our example, if we had to use the links from artists to tags, we would have to assign a weight to them. However, we're using these links in two contexts: similarity of artists and the rule deprecating artists marked by the given tag for female users. Evaluating such relationship once in a positive meaning and once in a negative would be much complicated. The second challenge of the [19] approach is that most of the main-stream recommender algorithms use only subjects, objects and relationships between them. Therefore the choice of a recommender algorithm is limited, or new algorithms have to be developed for the recommender.

The relationships between entities that are not subjects or objects can be used in our representation as well, but only as parts of the described rules. They do not have a particular representation in Unresyst.

To our relationship definition we can include a relationship of any type:

One-to-one relationships can be included by directly traversing the relationship.

One-to-many relationships can be included for example by testing the equality of the connected entities. In our example a relationship user - country is a one-to-many relationship. A user can be from at most one country, while one country can be a home of many users.

Many-to-many relationships can be included too. In our example the artist tags are many-to-many relationships. One artist can be tagged by many tags and one tag can be assigned to many artists.

Representing Predicted Relationship

During the adaptation phase, the domain expert has to define which relationship should be predicted. In our example, we would like Unresyst to recommend artists to users.

The predicted relationship definition is is done in a way similar to defining relationships that are relevant for recommendations. The only attribute that is omitted is the *weight*. The condition and the description remains as described in the section 5.2.1.

The formula 5.4 depicts the condition function of the predicted relationship pr, where $scr : S \times O \to \mathbf{N}$ is a function that for a user s an artist o gives the number of scrobbles (how many time the user played artist's songs).

$$D^{pr}_{pred}(s,o) = \begin{cases} True & \text{if } scr(s,o) > 0 \\ \\ False & \text{otherwise} \end{cases} \tag{5.4}$$

The following lines of code show the predicted relationship pr written in our Python prototype recommender implementation.

```
from unresyst import *
from models import *


class ArtistRecommender(Recommender):
    """A recommender recommending artists (musicians) that
    the user can like.
    """

    # ...

    predicted_relationship = PredictedRelationship(
        name="User listens to artist's tracks.",

        # gives true for user, artist pairs where the user
        # have listened to the artist
        condition=lambda user, artist: \
            user.scrobble_set.filter(track__artist=artist).exists(),

        description="User %(subject)s listens to the " + \
            "%(object)s's tracks."
    )
    """The relationship that will be predicted"""
```

Representing Biases

Apart from rules and relationships the domain expert can define the so-called biases. Object bias indicates a tendency of an object to be liked more or less than others, depending if the bias is positive or negative. Object bias is a property of an object and is independent of the subject to whom we're recommending. We denote a set of object biases for a recommender B_{obj}. Subject bias is an analogous concept for subjects. They indicate a higher tendency for a subject to like any object. Subject biases are useful only when the absolute expectancy matters, as for predicting object rating (see 4.1). We denote a set of subject biases for a recommender B_{subj}.

As for rules and relationships, we use an object-oriented representation of biases. Each bias instance contains the following attributes.

Condition A boolean function determining whether the bias can be applied to a given entity. The function is defined depending on the rule type:

Object bias condition $C_{bobj} : O \to \{False, True\}$. A function takes an object and determines whether the object is influenced by the bias. In our example the condition function is: If the artist o is played more than N times in a given recent period, the condition evaluates to *True*. Otherwise it's *False*

Subject bias condition $C_{bsubj} : S \to \{False, True\}$. A function takes a subject and determines whether the subject is influenced by the bias.

Is Positive A boolean constant (*True* or *False*) for determining whether the bias is positive for the preference or negative. Our example bias is positive.

Confidence A function determining the strength of the bias in a positive or a negative way.

Object bias confidence $D_{bobj} : S \to [0, 1]$. A function takes an object and gives a confidence, that the object will be preferred by any given subject (or not preferred when negative). High values mean strong confidence. Low values mean that the rule is unsure about the impact (positive or negative) on the preference. In our example we take a confidence function returning the artist play count divided by the maximum play count for an artist in the given period.[3]

Subject bias confidence $D_{bsubj} : S \to [0, 1]$. A function takes a subject and gives a confidence, that the subject will prefer any given object (or not prefer when negative).

Weight, Description The properties has the same meaning as in rule definition (see 5.2.1).

Let's show the bias representation on an example bias $b \in B_{obj}$ *Artists that were much played recently are likely to be played.*

The formula 5.5 depicts the condition function of bias b, where $scr_P : S \times O \to \mathbf{N}$ is a function that for a user s an artist o gives the number of scrobbles in the given time period P (how many time the user played artist's songs in the period), $mincount \in \mathbf{N}$ is a constant determining the lower bound for applying the bias.

$$C^b_{bobj}(o) = \begin{cases} True & \text{if } \sum_{s' \in S} scr_P(s', o) > mincount \\ \\ False & \text{otherwise} \end{cases} \tag{5.5}$$

The formula 5.6 depicts the confidence function of bias b, where $maxcount \in \mathbf{N}$ is a constant.

$$D^b_{bobj}(o) = \frac{\sum_{s' \in S} scr_P(s', o)}{maxcount} \tag{5.6}$$

The following lines of code show the example bias written in our Python prototype recommender implementation.

```
import datetime

from django.db.models import Count

from unresyst import *
from models import *

MAX_PLAY_COUNT = 542
"""The maximum play count for an artist in the period"""
```

[3]For the sake of simplicity we take an absolute play count and a fixed time period. For better results it would be better to take the number of plays relative to an average artist play count in a given period. Our example function would probably give high biases to a constant set of popular artists.

```
N_MIN_PLAY_COUNT = 100
"""The minimum play count for an artist to apply the bias"""

PERIOD_START_DATE = datetime.date(2010, 9, 1)
PERIOD_END_DATE = datetime.date(2010, 12, 31)


class ArtistRecommender(Recommender):
    """A recommender recommending artists (musicians) that
    the user can like.
    """

    # ...
    biases = (
        ObjectBias(
            name="Artists whose tracks have been listened a lot" + \
                " recently.",

            description="%(object)s has been listened much recently.",

            # take only artists with more than the minimal play count
            # in the given period
            condition=lambda artist: \
                artist.track_set\
                    .filter(scrobble__timestamp__range=
                        (PERIOD_START_DATE, PERIOD_END_DATE))\
                    .aggregate(Count('scrobble')) > N_MIN_PLAY_COUNT

            weight=0.5,

            # it's a positive bias
            is_positive=True,

            # the number of scrobbles for the artist divided by
            # the maximum
            confidence=lambda artist: \
                float(artist.track_set\
                    .filter(scrobble__timestamp__range=
                        (PERIOD_START_DATE, PERIOD_END_DATE))\
                    .annotate(scrobble_count=Count('scrobble'))\
                    .aggregate(Sum('scrobble_count')))/MAX_PLAY_COUNT
        ),
    )
```

5.2.2 Runtime Interface

Unresyst runtime interface serves for performing common operations in the recommender. The interface is made easy to use, so that Unresyst can be incorporated directly in the parent system without needing to implement an adapter.

The runtime interface contains the following methods.

build() The build method does all computations and prepares all necessary data structures of the recommender in order to start recommending. The state of the system database in the time of build is used for recommending. Changes made in the parent system database will take effect in recommending after either calling the update method or the build method. The build method can take an indispensable amount of time and system resources and therefore it should be called only by an entrusted person. It shouldn't be accessible to a common system user in any way. Build must be called before using any other runtime interface method.

predict_relationship(*subject*, *object*) The function predicts the preference in the means of the predicted relationship, of the given subject to the given object. The return value is an instance of the *RelationshipPrediction* class (see the next paragraph for details). The function is meant to be called very often. Therefore the execution should be pretty fast, independent of the number of entities in the parent system.

get_recommendations(*subject*, *count*) The function returns the given number of recommendations for the given subject. The return value is a list of instances of the *RelationshipPrediction* class (see the next paragraph for details). The function takes advantage of the structures precomputed during the execution of the build method. For reasonable values of the count parameter it should be executed in a constant time.

update(*entity*) A method that is called after an update in the parent system database in order to propagate the change to recommendations and predictions. The entity parameter is either a subject or an object. The method recomputes all structures we have created during the build method for the given entity. For some of the algorithms the update method might be as complex as performing build. The update method isn't implemented in our recommender prototype.

The *RelationshipPrediction* class, which instances are returned by the runtime methods *get_recommendations* and *predict_relationship*, has the following attributes.

subject The subject of the prediction, typically the system user. The attribute contains a parent system class instance, no conversion is needed.

object The object of the prediction, analogously to the *subject* attribute.

expectancy The estimated probability of the predicted relationship occurring between the subject and the object, which is a rate of subject-object preference. The attribute contains a float number between from the $[0, 1]$ interval.

explanation A short textual explanation of the provided prediction. The attribute contains a short text that can be presented to the user. For some algorithms this attribute may be unavailable.

In our prototype implementation, all runtime interface methods are placed directly on the Recommender base class. When using the interface, we work with our class defined in the adaptation phase. In all runtime methods we work directly with the parent system objects, no conversion is needed.

The usage of the runtime interface of our Python prototype implementation can be seen on the following lines. Firstly we import all necessary classes and build the recommender. The `User` model points to a table in the `lastfm` application containing system users, the `Artist` model table contains artists. These are domain-specific models belonging to the application. Using the recommender method `get_recommendations` we obtain one artist that is according to the recommender most likely to be liked by the user 44. The recommender proposes artist *Cat Stevens*, with expectancy 0.75. The recommender explains, that the user 44 is similar to user 2, because they're both females, and the user 2 has already listened to Cat Stevens.

```
>>> from lastfm.models import *
>>> from lastfm.recommender import *
>>>
>>> NovelArtistRecommender.build()
>>> u = User.objects.get(id=44)
>>> NovelArtistRecommender.get_recommendations(u,1)
[< user_44 <- Cat Stevens: 0.750000,
Similarity: user_44's gender is f. user_2's gender is f.
And: User user_2 listens to the Cat Stevens's tracks. >]
```

The next lines of code show predicting the relationship. Here we ask about an artist, who the user has already listened. Therefore the expectancy is 1.0.

```
>>> u2 = User.objects.get(id=78)
>>> a = Artist.objects.get(id=32)
>>> NovelArtistRecommender.predict_relationship(u2, a)
< user_78 <- Boards Of Canada: 1.000000, User user_78 listens
to the Boards Of Canada's tracks. >
```

5.3 Applying Rules

In the section we describe the techniques used for applying the given rules, relationships and biases on the domain data in order to compute relationship predictions that can be used directly or as an input for a collaborative filtering algorithm. Firstly we create a domain neutral representation for all recommender subjects and objects, then we create links between these according to the configuration given in the adaptation phase. For entities and entity pairs where multiple

similarity relationships or biases meet, we aggregate them. Finally we perform a BFS[4] with a very limited maximum depth.

5.3.1 Matching Rule Conditions to Entities and Entity Pairs

After creating a domain-neutral representation of subjects and objects, we continue by creating links between entities, for which a rule, relationship or bias can be applied. These links are called rule/relationship/bias *instances*. Both these actions are performed in the Abstractor application layer (see 5.4.2).

Matching the entities and entity pairs to the given conditions is done by substituting all possible entities and entity pairs one by one to the given conditions. As the complexity of this action is $O(mn)$ (m is the number of subjects, n the number of objects), in our prototype implementation we enable defining a *generator* function for each rule, relationship or bias (see 5.5 for details).

In our example, we create a similarity rule instance between Alice and Bob as the age of the two is similar, another similarity rule instance between Arctic Monkeys and Modest Mouse as these share the "Indie" tag.

The data model is designed so that data that is shared among all rule matches, like weight and name, are kept on one place (see 5.5.1 for details).

5.3.2 Aggregating Similarities and Biases

For entity pairs to which multiple similarity links were applied, and for entities to which multiple biases apply, we need to *aggregate* them to provide at most one similarity instance for each entity pair and at most one bias instance for each entity. This is done on the top of the Algorithm application layer (see 5.4.3)

When aggregating, we also transform the confidence and positiveness to the general expectancy rate (see 2.4.3 for details on confidence and expectancy concepts). The transformation is done by the formula 5.7 where s is an instance of a rule/relationship/bias, w_s is the weight of the rule/relationship/bias and $c(s)$ is the confidence of the instance.

$$ex(s) = \begin{cases} \frac{1}{2} + \frac{w_s c(s)}{2} & \text{if } s \text{ is an instance of a positive rule/relationship/bias} \\ \\ \frac{1}{2} - \frac{w_s c(s)}{2} & \text{if } s \text{ is an instance of a negative rule/relationship/bias} \end{cases} \tag{5.7}$$

In our example, if Alice and Bob were of the similar age and moreover they were from the same country, we would aggregate those two similarity instances to one aggregated instance.

The combination of multiple instances into one aggregate is done in a separate module called *Combinator*, see 5.3.4. The combining is done outside the Aggregator level in order to have a possibility to switch combination strategies without having to modify the code of the Aggregator layer.

[4]BFS stands for Breadth-First Search, a technique used for traversing graphs in the order according to their node level.

5.3.3 Compiling Rules to Preference Predictions

Later on we compile all information we have to provide predictions of subject-object preferences in means of the predicted relationship.

For providing a prediction for a given subject-object pair, the following preference sources are taken into account.

Subject-object rules Preference rules that apply to the given pair are taken directly as a preference source. An example of such rule is our negative rule not recommending artists tagged by a male-specific tags to female. For Cindy and "Black Sabbath" this rule will negatively impact the resulting preference prediction. Formally if there's a single rule to be applied to the *s-o* pair, we set $\hat{u}_R(s,o) = D_{so}(s,o)w_{r_1}$ if $C_{pref}(s,o) = True$ for a given rule $r_1 \in RL$.

Known relationships and aggregated similarity Another way to create a preference prediction is to take an already known preference and recommend a similar object to the one that is already preferred by the subject, or to recommend the preferred object to a subject that is similar to the one who preferred the object. In our example we recommend Modest Mouse to Alice, as she has listened to Arctic Monkeys band which is known to be similar to Modest Mouse. Formally for object similarity if there is a single rule applied to the *s-o* pair, we set $\hat{u}_R(s,o) = D_{oo}(o,o') * w_{r_2}$ if $C_{osim}(o,o') = True$ for a given rule $r_2 \in RL$. Moreover we recommend U2 to Alice, because Bob has listened to U2 and his age is similar to Alice's. Formally for subject similarity, if there is a single rule applied to the *s-o* pair, we set $\hat{u}_R(s,o) = D_{ss}(s,s')w_{r_3}$ if $C_{ssim}(s,s') = True$ for a given rule $r_3 \in RL$.

Aggregated Bias The subject or object bias is also taken into account for relationship predictions. If the subject or the object (or both) have a bias, it's included to the prediction. In our example The Tape has a bias which influence all predictions for pairs where The Tape is the object. Formally for object bias, if there is a single rule applied to the o object, we set $\hat{u}_R(s,o) = D_{bobj}(o)w_{r_4}$ if $C_{bobj}(o) = True$ for $\forall s \in S$ and a given rule $r_4 \in RL$, analogously for the subject bias.

For some of the subject-object pairs more than one of the preference sources can be available, e.g. we have a subject-object rule for the pair and the object is one of the biased ones. For such pairs we use the combinator module (see 5.3.4) to combine these sources to a single prediction.

During the build, we don't compile the predictions for all possible subject-object pairs, as this would be time and space consuming. Instead of that we select the given number of *promising* objects for all subjects in the domain in order to provide fast recommendations. See the section 5.5 for details.

5.3.4 Combining

In some cases there can appear multiple pieces of information for similarity, bias and preference and we need to combine them to get a single result. The combination is done in a module called *Combinator*. The combination method is kept on

one place so that the same method can be used for combining all pieces of information in all places. The piece of information to combine is called a *combination element*.

Multiple combination elements can appear for some entities and entity pairs during the following actions.

Aggregating similarities and biases During the aggregation, multiple similarity instances can appear on a single subject or object pair. Multiple biases can appear on a single subject or object. See 5.3.2 for details on aggregation.

Compiling rules to preference predictions During the compilation, multiple combination elements can appear on a single subject-object pair, see 5.3.3 for the list of all possible preference sources. The similarity preference sources can produce multiple combination elements for one subject-object pair.

Intuitively, when combining two combination elements we would like the result to fulfill the following requirements:

1. If one element is positive and the other negative, the result should be neutral.
2. If both elements are positive, the result should be more positive than both of the elements.
3. If both elements are negative, the result should be more negative than both of the elements.

For combining the combination elements we propose functions described in following subsections.

Twisted Average

The first idea for multiple element combination is to use weighted average. But as the weighted average function doesn't meet the requirements 2 and 3, we will have to alter it so that it meets the requirements.

The idea on how the altered average function should look like is on the figure 5.3. The straight line is the average between the expectancies, the curved line is our idea of how our function should look like, when the positive or negative effect is confirmed by multiple rules. For positive effects (higher than 0.5) it gives slightly higher values than the average, for negative effects (lower than 0.5) it gives slightly lower values. For modeling the function we use as a base function $2x^2$, which is to be used for the negative effects (lower than 0.5). For positive effects, we transform the base function so that the resulting function is symmetric around the point $(0.5, 0.5)$. The resulting function for combining two rules is 5.8.

$$CMB(e_1, e_2) = \begin{cases} 2(\frac{e_1+e_2}{2})^2 & \text{if } \frac{e_1+e_2}{2} \in [0, \frac{1}{2}] \\ 1 - 2(\frac{e_1+e_2}{2} - 1)^2 & \text{if } \frac{e_1+e_2}{2} \in (\frac{1}{2}, 1] \end{cases} \tag{5.8}$$

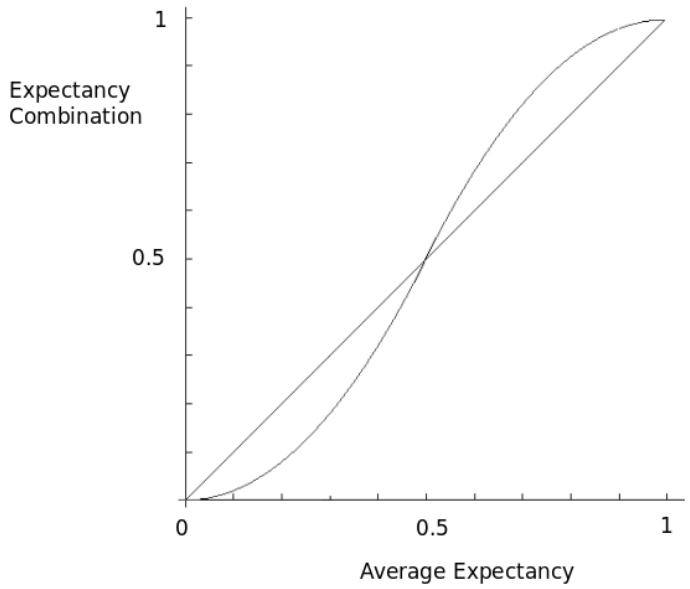


Figure 5.3: Combining expectancy by the Twisted Average function.

To generalize 5.8 for multiple rules, we define 5.9, where a denotes the average of the combined expectancies and n denotes the absolute difference between the number of positive and negative expectancies.

$$f(a) = \begin{cases} 2^n a^{n+1} & \text{if } a \in [0, \frac{1}{2}] \\ \\ 1 - |2^n (a-1)^{n+1}| & \text{if } a \in (\frac{1}{2}, 1] \end{cases} \tag{5.9}$$

Certainty Factor Calculus

The problem of combining multiple pieces of evidence having different certainties was a subject of interest of early expert systems like Mycin [34]. In the Mycin expert system, the certainty-factor calculus was used. The certainty-factor model [35] uses a numerical measure of certainty which values range from -1 and 1, where negative values indicate disbelief in the given hypothesis and positive values indicate the belief in the hypothesis.

In Unresyst our hypothesis can be formulated as *The subject prefers the object* for subject-object rules, or *The subjects/objects are similar* for similarity rules or *The object is likely to be preferred by any subject/ The subject is likely to prefer any object* for biases. As our expectancy measure ranges from 0 to 1, we have to transform the certainty-factor combination function, so that it fits our expectancy concept. The function for transforming expectancies to certainty factor is 5.10, the inverse function is 5.11.

$$t(x) = 2x - 1 \tag{5.10}$$

$$t^{-1}(x) = \frac{x+1}{2} \tag{5.11}$$

The factor combination formula described in [35] is 5.12, where cf_1, cf_2 are certainty factor values of the combined pieces of evidence.

$$CFC(cf_1, cf_2) = \begin{cases} cf_1 + cf_2(1 - cf_1) & \text{if } cf_1, cf_2 > 0 \\ \frac{cf_1 \ cf_2}{1 - min(|cf_1|, |cf_2|)} & \text{if } -1 < cf_1 cf_2 \leq 0 \\ cf_1 + cf_2(1 + cf_1) & \text{if } cf_1, cf_2 < 0 \end{cases} \tag{5.12}$$

Finally, our resulting function that combines two expectancies to a single expectancy $CMB : [0,1] \times [0,1] \rightarrow [0,1]$ is in 5.13.

$$CMB(e_1, e2) = t^{-1}(CFC(t(e_1), t(e_2))) \tag{5.13}$$

As noted in [35], the function is commutative and associative in its second argument, so the order of the combination has no effect on the result. The resulting function apparently fulfills the requirements for our combination function.

5.4 Unresyst Application Layers

The oncoming sections describe the proposed architecture of Unresyst. The overview of the layers is illustrated in the figure 5.4.

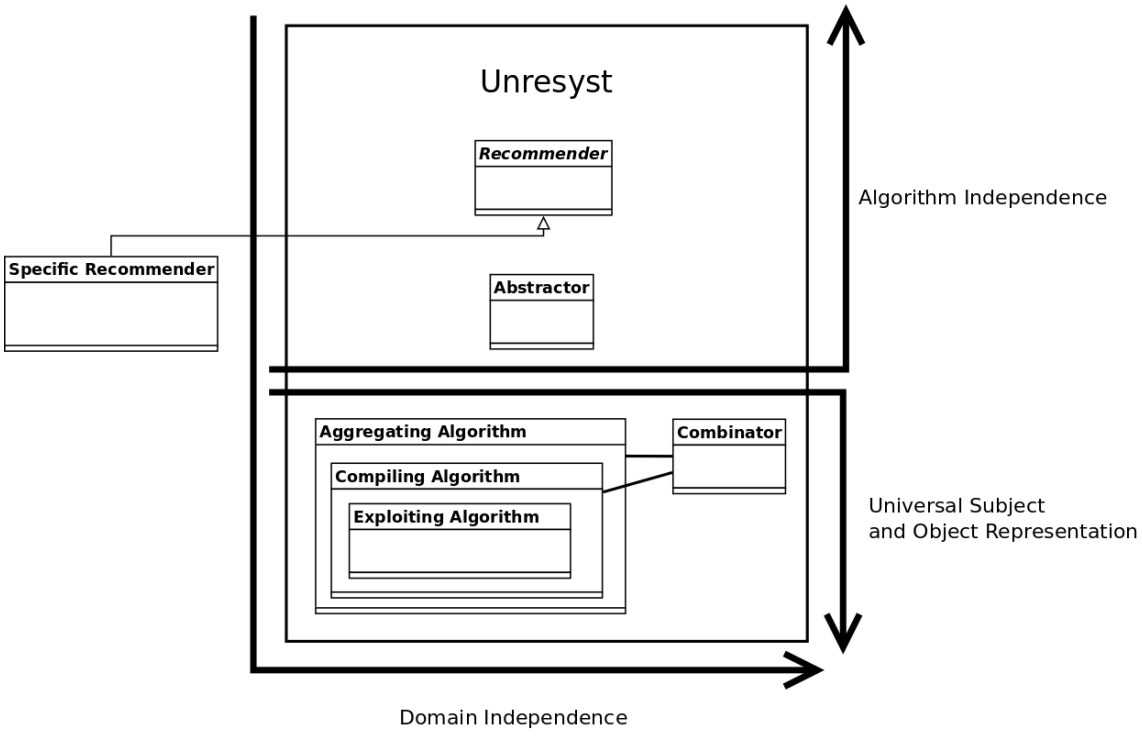


Figure 5.4: The Unresyst architecture layers

5.4.1 Recommender Layer

The top most level (*Recommender*) provides the runtime interface (see 5.2.2) to the whole recommender, the adaptation data is stored in the specific recommender subclass (see 5.2.1).

5.4.2 Abstractor Layer

The second layer (*Abstractor*) cares for converting the domain-specific objects and relationships to a domain-neutral representation. In Abstractor, the domain specific subjects, objects, rules, relationships and biases are converted to an abstract, domain-neutral representation which is later used by the underlying algorithm. Therefore all actions done in layers under the Abstractor layer can be implemented as domain-neutral.

Our prototype implementation follows the proposed architecture. The classes of the recommender follow the proposed layers. The final configuration of classes that will be used as the proposed layers is done right in the *Recommender* base class, so that it can be overridden in its specific subclasses.

5.4.3 Algorithm Layer

Under the Abstractor there is a domain-neutral *Algorithm* layer. The algorithm is further structured to levels so that a new algorithm can be plugged to any of the levels. The proposed levels of the algorithm layer are the following.

Aggregating algorithm level The level cares for aggregating multiple bias and rule instances during the build and update, see 5.3.2 for details. For combining multiple rule instances it uses the Combinator module (5.4.3). The input for the level are rule, relationship and bias instances, the output are aggregated instances. For each entity pair there can be at most one aggregated relationship and for each entity there can be at most one aggregated bias.

Compiling algorithm level The level compiles the subject-object rules, aggregated similarities and aggregated biases to relationship predictions during the build and update, see 5.3.3 for details. For combining multiple sources of prediction it uses the Combinator module (5.4.3). The input of the level are subject-object rules, aggregated similarities and aggregated biases. The output of the level are compiled subject-object preference predictions.

Exploiting algorithm level The level takes the predictions obtained from the higher level and later exploits them for other pairs. For this layer, we can use a classical collaborative filtering algorithm as described in 3.2. In the prototype implementation we use only a simple algorithm returning predictions it has obtained from the compiling level. The input of the level are the predictions, the output are the obtained predictions, optionally exploited to some previously unpredicted subject-object pairs.

The levels provide a possibility to include a new algorithm at any level. E.g. if we find an algorithm that can produce predictions from similarities, aggregated preferences and aggregated biases, we can put the new algorithm in the place of our Compiling Algorithm. The leveled structure even enables us to create a new algorithm with other layers than the proposed ones.

In our prototype implementation all algorithm level classes are derived from the same base class, having the same interface for building the algorithm and providing predictions and recommendations. The complete interface description can be found in the generated documentation on the attached CD.

Combinator

As the combination of multiple sources of information is done on two places in different algorithm levels, the *Combinator* module is excluded from the algorithm and defined as a separate module.

The module cares for combining multiple combination elements into a single preference, similarity or bias. The methods proposed to be used in the Combinator can be found in 5.3.4. The proposed methods are implemented in our prototype as separate Combinator subclasses.

5.5 Unresyst Prototype Implementation

In the section we disclose some implementation details of our recommender prototype. We describe the data model of our implementation, then we give some details about problems that had to be solved during the implementation.

5.5.1 Unresyst Prototype Data Model

As the data model of the Unresyst application is extensive, we had to divide the model to smaller ones: Basic data model, Biases data model, Aggregator data model and Algorithm data model. The figure displaying the complete data model can be found on the attached CD.

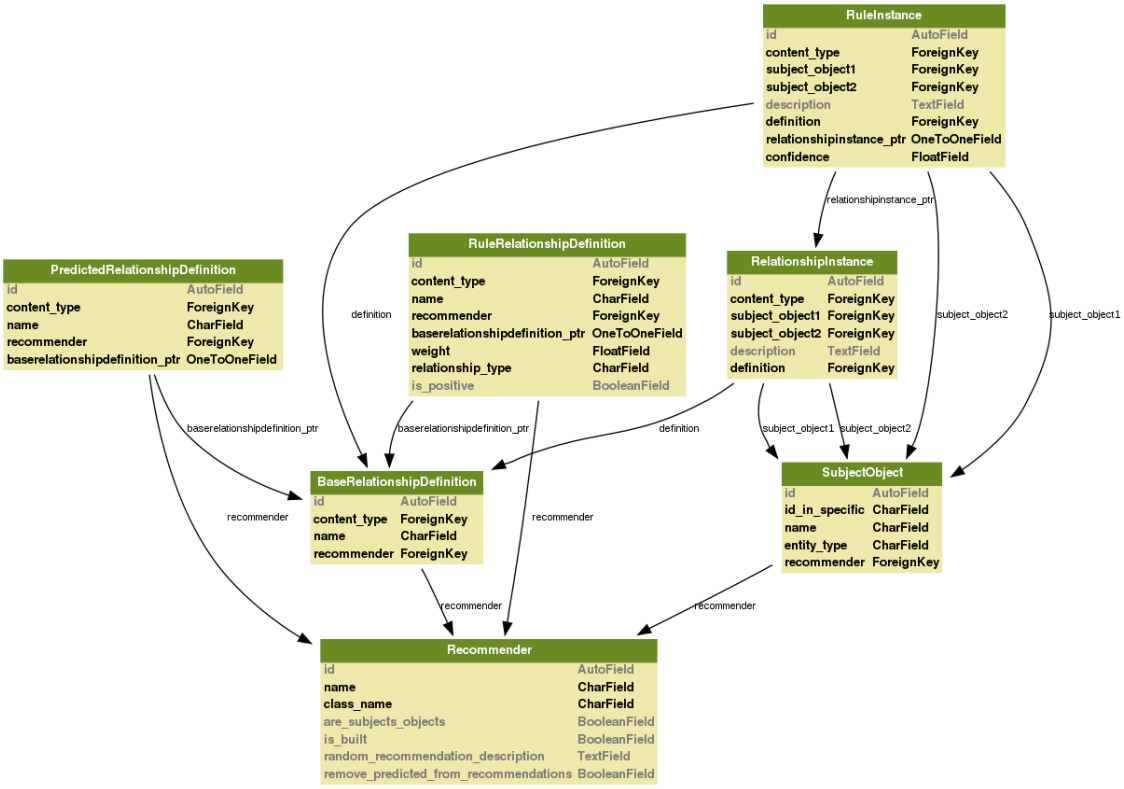


Figure 5.5: The basic data model of Unresyst

The basic data model of the Unresyst application consists of the models displayed in the figure 5.5. All these models are filled by the Abstractor layer (see 5.4.2).

As there can be multiple recommenders in one parent system, we need a representation of the recommender in the database. This is the *Recommender* model. One recommender model in database corresponds to one specific recommender. All other models for recommendations are linked to the recommender model they belong to.

Note that domain neutral representations of subjects a objects are stored in one model. Thanks to that, we can use a single model for all kinds of relationships (preference and similarity). Each rule defined in the adaptation phase has one *definition* and for each pair it applies to, it has one *instance*. Definitions contain all data that are common to all instances, like weight, name and the link to the recommender. Instances connect the affected pairs of entities and hold the confidence for the pair (just for rules).

The models related to biases are displayed in the figure 5.6. As for rules and relationships, each defined bias has one definition and an instance for each entity it applies to. Bias definitions and instances are also filled by the Abstractor layer.

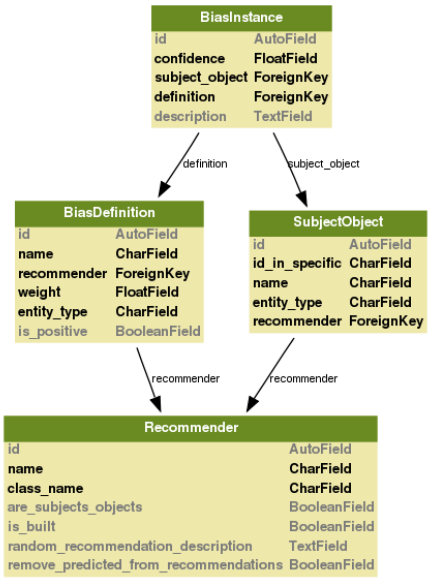


Figure 5.6: The data model of entity bias

The Aggregator data model on the figure 5.7 contains models filled by aggregator level of the algorithm layer (see 5.4.3). The aggregated instances group together rule, relationship and bias instances.

The data model for our simple algorithm is in the figure 5.8. As our implementation only returns the results of the levels above, it only contains a model for preference prediction. These models aren't generated for all possible pairs as this wouldn't be feasible for larger domains. However, as we need to generate recommendations for a given subject fast, we do some pre-computation. During the build we select the given number of promising objects for each subject. The promising objects are chosen from all rules, relationships and biases that

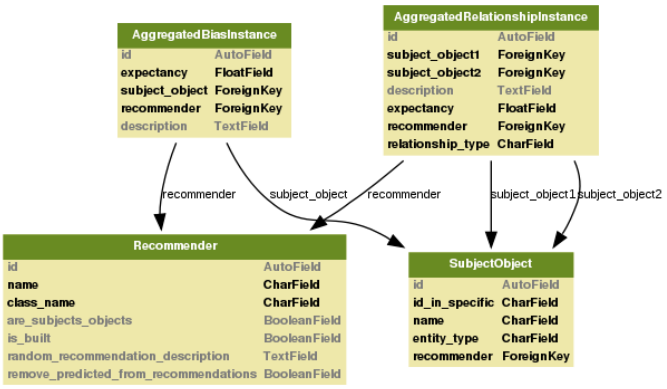


Figure 5.7: The Aggregator data model

are available. For these objects, all known preference sources are inspected (see 5.3.3) and the predictions for the pairs are saved.

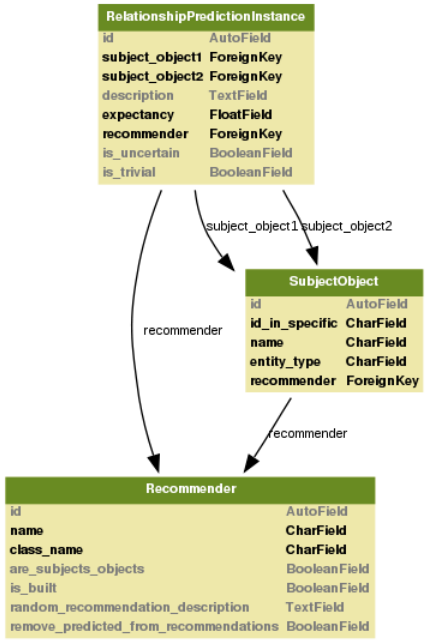


Figure 5.8: The data model of our simple algorithm implementation

5.5.2 Implementation Details

The section presents details about some of the problems that had to be solved during the Unresyst prototype implementation.

The representation of similarity relationships was in some cases ineffective, for example the similarity relationship between users coming from the same country from our introductory example. When building the recommender with such relationship definition, we would have to connect by a relationship instance all pairs

of users coming from the same country - the number of relationship instances would be quadratic to the number of users.

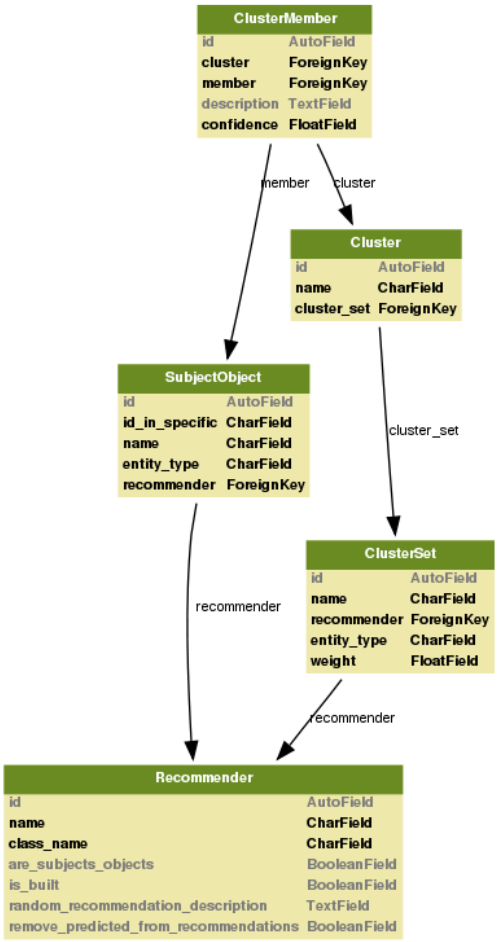


Figure 5.9: The data model of our cluster implementation

For higher effectivity, we introduced the *cluster* concept. I.e. for each country we create a cluster and we connect each user with the cluster appropriate to the country he/she is from. The similarity can then be obtained by checking whether the pair of users is connected to the same cluster. In this case we reduce the number of links so that it's at the most equal to number of users. The domain-neutral representation of models related to clusters is in the figure 5.9.

In the Flixster data set there is an explicit feedback relationship that can be both positive and negative. For these kinds of relationships we introduced a new type of rule - an Explicit Rule. In this kind of rule there is no positiveness given, as the feedback can be both positive and negative. Instead of that, in the rule we define directly the expectancy measure. The data model for explicit rules is displayed on the figure 5.10.

In the implementation of different levels of the algorithm layer (see 5.4.3), we had to design the classes so that one class can contain an instance of a similar type of class (a subclass of the same class). This directed us to the Composite design pattern [36], which was slightly modified for our needs. The diagram of

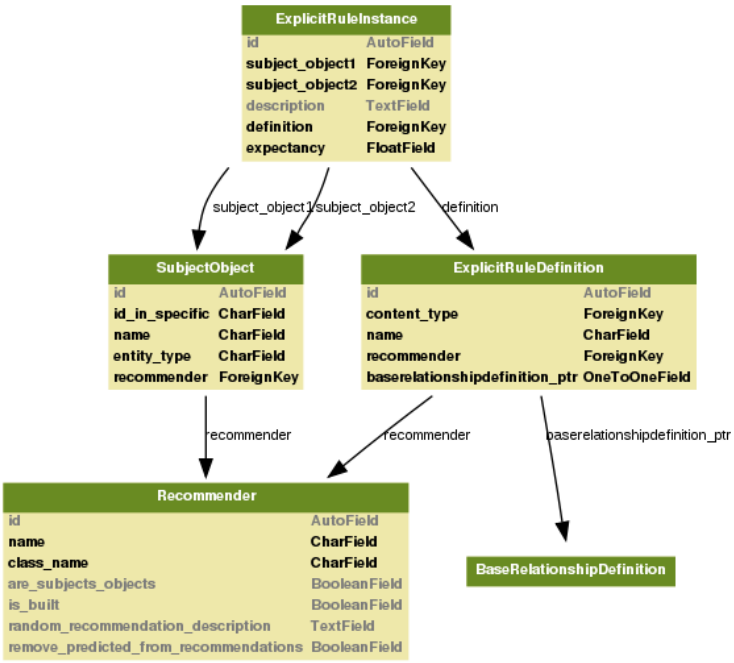


Figure 5.10: The data model of the explicit rule definitions and instances

the classes in the algorithm layer is displayed in the figure 5.11. The predicting and recommending methods are implemented only in the inner-most class - the simple algorithm.

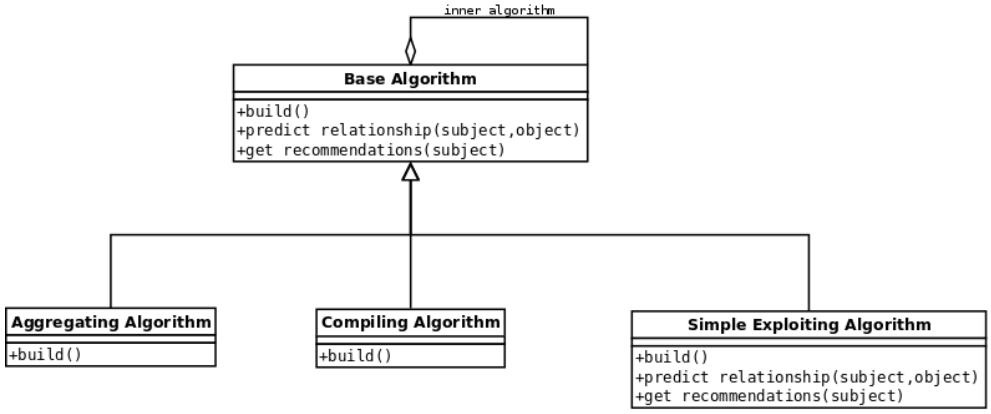


Figure 5.11: The diagram of the classes in our implementation of the algorithm layer

As the prototype was to be adapted to real data sets with thousands of subjects and objects we had to make some technical enhancements to make the prototype recommender build in a reasonable time. Firstly, in the rule, relationship and bias classes we added a possibility to define a *generator* instead of the condition. Generator is a function returning entities or entity pairs that satisfy the condition. The use of the generator in rule abstraction is much faster because

we don't have to traverse all entities or entity pairs to test the condition, instead of that we directly evaluate the confidence on the entities/entity pairs returned from the generator.

As can be noted in the data models 5.5 and 5.7, all relationship instances are symmetric, as we use a single model for subjects and objects. When saving a relationship instance between the entities (A, B), we have to check whether the relationship (B, A) isn't already saved. The same check has to be performed when searching for a relationship between the entities. But as the check for the opposite direction adds an extra database query, we introduced some regulations for saving the relationship entities so that the opposite order doesn't have to be checked. The regulations are the following

1. If the relationship is between a subject and an object, we save the subject to the first position, object to the second

2. If it's a similarity relationship we save the entity with the lower private key to the first position and the entity with the higher private key as the second.

These regulations guarantee the order of the entities in the relationships and therefore we can perform a save or retrieval of a relationship within a single database query.

6. Verifying the Universal Recommender on Real-World Data

In the chapter we describe the process of adapting the Unresyst prototype to the selected data sets (see Chapter 4).

In the following sections we give an overview of relationships we have predicted using Unresyst. We give an overview of the rules we used for the Unresyst configuration. As each data set contains more data than our prototype can manage, we had to reduce the data sets in the manner described in the following subsections. In addition, we give some basic statistics of the counts of subjects and objects in the reduced data sets.

The data models and descriptions of the attributes that we can use for recommending, can be found in the Chapter 4. The complete source code of the presented rules can be found on the attached CD.

6.1 Last.fm

We decided to predict the relationship *User listens to Artist*, rather than predicting the listened tracks. That's because the range of played tracks from each artist is pretty wide and different users tend to listen different tracks.

We have set up two recommenders for the Last.fm data set: a novel artist recommender, recommending only artists that the user hasn't heard before and a radio recommender recommending all artists no matter if they have been heard or not. The latter recommender could be used e.g. for selecting music to play in a user's personalized internet radio.

In Unresyst configuration for the Last.fm data set we used the following rules:

- Similarity of users according to their age. We take only pairs of users whose age difference is lower than a given constant. The confidence of similarity raises with lowering age difference. We suppose, the users of similar age are likely to listen to the same artists.

- Similarity of users according to their registration date. We used the same mechanism as for user age. We suppose that in some period of time, the users listening a given kind of music could become members of the last.fm, e.g. when an article about Last.fm was published in a magazine specialized in the given kind of music.

- Similarity of users according to their gender, home country. The rules were implemented through clusters (see 5.5.2).

- Artist similarity according to their social tags. We take only pairs of users that have more than a given constant of tags in common. The more tags they have in common the higher is the confidence. First we tried to implement the similarity using tags as clusters, but in this application clusters

were less efficient than similarities, as the number of tags for each artist (where available) was pretty high.

- Preference derived from the number of user's scrobbles for the artist. The confidence depends on the percentage of the artist scrobbles in the overall user's scrobble count. The rule is used only in the radio recommender. We suppose that artists heard many times before are likely to be liked.

- Positive bias for popular artists. Artists that have been listened by many users (more than a given constant) get a positive bias. We suppose that artists, that were listened by a high number of users, are likely to be liked by any user.

- Negative preference for artists tagged by a gender-specific tag. As shown in the introductory example (1.2.1) we classified some of the most popular tags as gender-specific. We have marked tags like "Hard Rock" or "Death Metal" as male-specific, and tags like "sweet" or "emotional" as female-specific. We suppose the gender-specific music is often disliked by the opposite gender.

As the data set originally contains millions of scrobbles, we had to reduce it to make it feasible for our recommender prototype and the available hardware. We took the first c. hundred users from the data set and then we selected every seven hundredth scrobble for each user. After that we obtained a data set with entity counts listed in the table 6.1. Assigning tags for the listened artists was done by matching their MusicBrainz [1] identifiers. We succeeded in finding tags only for c. 14% of the artists, but for those there was a high number of tags, in average 89 unique tags for an artist.

2348	artists
100	users
5057	tracks
5532	scrobbles
9120	tags
28588	artist-tag links
322	tagged artists

Table 6.1: Statistics for the reduced Last.fm data set

6.2 Flixster data set

The Flixster data set is a classic collaborative filtering data set with explicit user-movie rating and without any attributes for users or movies. Additionally, Flixster contains links between users who are friends. As we had the only user-movie interaction relationship - the rating - we took as the predicted preference

[1] MusicBrainz is a community music metadatabase that attempts to create a comprehensive music information site. Each contained artist has a unique identifier. See `http://musicbrainz.org/`.

relationship a rating that was higher than a given constant. If the user gave the movie the given number of stars, he/she probably likes it.

For Unresyst setup we used the following rules.

- The explicit rule for user - movie ratings. As rating can give both positive and negative feedback we introduced a special rule type for it, see section 5.5.2 for details. The five-stars scale was normalized to a float number between 0 and 1.
- Similarity of users according to the friend relationship. We suppose that users who are friends have a similar taste so they like similar movies.
- Object bias for high- and low-rated movies. We assume movies that were given exceptionally high/low ratings are supposed to be liked/disliked by many users.
- Subject bias for high- and low-rating users. As mentioned in the section 5.2.1, the subject biases are relevant only in cases when we care about the exact expectancy number. This is the case of the Flixster data set as we try to predict the exact user - movie rating. Some users tend to rate all movies high whereas some sceptic users give low ratings to most of the movies. We include this remark to predictions by defining a subject bias.

The data set originally contained over eight million ratings and over seven million friend links. When reducing the data set we took a subset of users so that the friend links were preserved. For these users we found the appropriate ratings.

418	users
4237	movies
14195	ratings
1051	friend links

Table 6.2: Statistics for the reduced Flixster data set

6.3 Travel agency

The travel agency data set was created through a dump of the tables where the collected implicit feedback was stored. Firstly we had to filter out feedback that wasn't related to a particular tour, as feedback was collected on the whole website, not just tour profiles. Then we categorized the feedback by saving the data to the data model displayed in the figure 4.3. The strongest sign of preference is ordering the tour, so we decided to predict the *User ordered a tour* relationship.

For Unresyst setup we used the following rules

- Implicit preference relationships - question, mouse move, click. When the user asked a question about the tour, has been actively moving the mouse over the tour profile, or has clicked on tour details or photos, we suppose he/she is interested in the tour.

- View profile time. As in the data set there were recorded the events of opening and closing a page, with timestamps, we used the data to determine time the user spent on the tour profile. The longer the time, the higher was the preference according to the rule.

- Similarity of tours according to their type and destination country. The rules were implemented through clusters (see 5.5.2).

- Positive object bias according to the amount of implicit feedback. Tours that were much actively visited were supposed to be more likely to be preferred by any user.

After processing the data set in the described way it had the parameters described in the table 6.3

193	users
753	tours
50	destination countries
227	tour orders
1551	tour profile views
672	user clicks to tour profiles
4570	mouse moves
5	questions

Table 6.3: Statistics for the Travel agency data set

7. Evaluating Recommender Results

In the chapter we describe how we evaluated Unresyst on the presented data sets. Firstly we describe the methodology of evaluating, then we discuss the choice of the evaluation metric for each data set, we give an overview of evaluation results. Finally we discuss possibilities of improving the Unresyst evaluation results.

7.1 Evaluation Methodology

As we only had access to the presented systems through the data sets, we couldn't perform a live experiment with real system users. Instead of that, we used an offline analysis as described in [37]. We ran Unresyst on a subset of available data (*training data*) and evaluated the results on the rest (*testing data*). In the database on which we performed the recommender build, we kept only the training data. Later on we took the testing data and measured the performance of the evaluator using metrics described in the section 7.2.

For a comparison, we ran the evaluation also on an external recommender that was using a common recommender algorithm. As an external recommender we used the Mahout recommender framework [8]. Our usage of Mahout is illustrated on the figure 7.1. We created the External Recommender class having the same interface as the Unresyst recommender. The recommender was using for its build the Mahout recommender, communicating via CSV[1] files. The input files contained train data for the Mahout recommender. The output files contained predictions or recommendations on the test data.

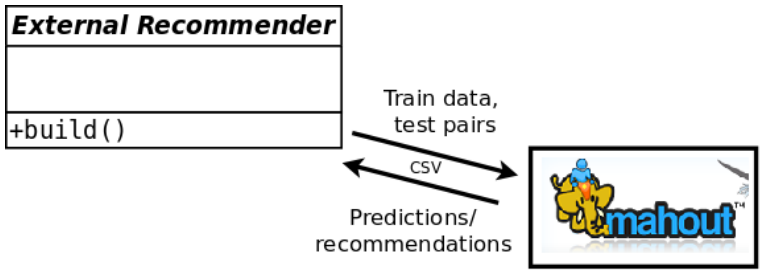


Figure 7.1: The usage of the Mahout recommender for comparison

7.1.1 Selecting Test Data

We used a constant ratio of testing to training data (1 : 4) for all data sets. In data sets where the timestamp in the feedback relationships was available (Last.fm and Travel agency), we selected the given part of the newest relationships as training data. Then the evaluation was done as if we "replayed" the scrobbles and orders

[1] CSV stands for Comma-separated values, a simple format of text files where the values (in our case identifiers of subjects, objects and optionally rating) are separated by commas.

recorded in the data set. In the Flixster data set where the timestamp wasn't available, we selected the ratings to the test set randomly.

In the travel agency data set we also removed all implicit feedback data that was newer than the newest contained tour order. This corresponds to a simulation of a live user test. In a given period we collect user feedback (orders, questions, clicks, etc.). In the next period we perform a test where we try to predict which tours a user orders, using only the data collected in the previous period.

7.2 Evaluation Metrics

For evaluating recommender accuracy there are a lot of various metrics [37]. In our case we had to deal with two different cases: explicit feedback where the exact preference value is available and implicit feedback where we have only positive signs of preference.

7.2.1 Explicit Feedback Metrics

For evaluating explicit feedback from the Flixster data set we have chosen RMSE, although the metric is an object of recent criticism [38]. RMSE stands for Root Mean Square Error and has been the most widely used metric for evaluating accuracy recommender system, since it was used in the Netflix prize for evaluating the participants' results [10]. Apart from being widely used it has an advantage of being simple to compute and it punishes large errors much more than the small ones.

The formula 7.1 computes RMSE for the whole recommender. In our case p_{so} is the preference of s to o taken from the testing data, $\widetilde{p}_{so}$ is a prediction of the preference of s to o, created by the recommender.

$$RMSE = \sqrt{\frac{\sum_{(s,o)\in TestPairs}(p_{so} - \widetilde{p}_{so})^2}{|TestPairs|}} \tag{7.1}$$

7.2.2 Implicit Feedback Metrics

The more complicated case is evaluating a recommender running on a data set where only implicit feedback is available. Last.fm and Travel agency data sets have only positive implicit feedback and the lack of feedback can't be taken as a negative preference. I.e. there can be several reasons why a user hasn't listened to an artists, maybe he/she doesn't like the artist or he/she only doesn't know about the artist.

Therefore using the implicit feedback from test data to evaluate the recommender by a metric like RMSE isn't much sensible, as we only have positive examples with the highest possible expectancy value 1. If we used such an evaluator, an ideal recommender would give 1 for all the items, which isn't the recommender we're looking for. We can't even use a slightly negative feedback for missing feedback, as there are much more objects with missing feedback, than with present feedback and there's no reasonable way to select objects without feedback to predict preference for.

In order to overcome the limitations of having only positive implicit feedback, we have to either restrict the number of objects the recommender can propose or we have to take relative values of the predicted expectancy. The metrics measuring coherency of the recommendation list and list from test data, as NDCG [39] aren't much helpful, as we don't have any ordering in the list from test data.

The paper [37] proposes using the *Precision/Recall* metric, which is heavily used in the field of machine learning and information retrieval. For our application, we count the metric for each user and count an average to get the numbers for the complete recommender. Objects that have an implicit feedback from the user are taken as *relevant*, others are irrelevant. We try to predict relevant objects for a subject via our *get recommendations* method. In the formulas for counting precision (7.2) and recall (7.3) we accommodate the Precision/Recall concepts to our needs. $Prec_s$ is precision of recommendations for subject s, Rec_s is recall of recommendations for subject s, R_s is a set of objects recommended to s, T_s is a set of objects in test set for s, N is the number of recommended objects.

$$Prec_s = \frac{|R_s \cap T_s|}{N} \tag{7.2}$$

$$Rec_s = \frac{|R_s \cap T_s|}{|T_s|} \tag{7.3}$$

We experimented with different values of N, but for all tested recommender algorithms both precision and recall were extremely low. The problem of this approach is, that it's taking objects with no implicit feedback as irrelevant, which isn't completely appropriate as the lack of positive feedback doesn't imply a negative preference. Therefore we had to seek another metric.

Authors of [40] used a metric called $\overline{rank}$ for evaluation of their recommender working with implicit feedback. The definition of the metric is the formula 7.4, where T denotes the set of testing subject-object pairs. $rank_{so}$ is a rank for the given subject-object pair, defined in 7.5. m denotes the number of objects in data set, i_{so} the index of the object o in a list of all objects ordered reversely by the expectancy for the subject s (the first object gets a 0). $rank_{so}$ has a value of 0 for the object o that is predicted to be the most desirable for the subject s, and it has the value of 1 for the object o that is predicted to be the least preferred by the subject s.

$$\overline{rank} = \frac{\sum_{(s,o)\in T} rank_{so}}{|T|} \tag{7.4}$$

$$rank_{so} = \frac{i_{so}}{m-1} \tag{7.5}$$

As the metric requires counting all expectancy of all subject-object pairs, we have performed it only for a number of randomly selected subjects. To count the $\overline{rank}$ metric we created an ordered list of all objects for a given subject. For each object o we had in the test set for the subject s, we obtained the index i_{so} and counted $rank_{so}$. For comparison, a random recommender would score around 0.5, as the objects from the test set would be randomly distributed in the list, giving the average 0.5.

7.3 Evaluation Results on the Datasets

When evaluating Unresyst on the studied data sets, we experimented with various recommender configurations. The results of our evaluation are in the following sections. Finally we give some ideas on improving the recommender results. For comparison to standard collaborative filtering methods we use a Slope One recommender [41], implemented in the Mahout [8] framework. Slope One is a relatively simple but efficient item-based collaborative filtering method.

7.3.1 Last.fm

For the Last.fm data set recommender evaluation we used the $\overline{Rank}$ metric (see the section 7.1, as we didn't have any explicit feedback. But as counting the metric for a single subject requires predicting his/her relationship to all objects in the system, we had to count the metric only for randomly selected users.

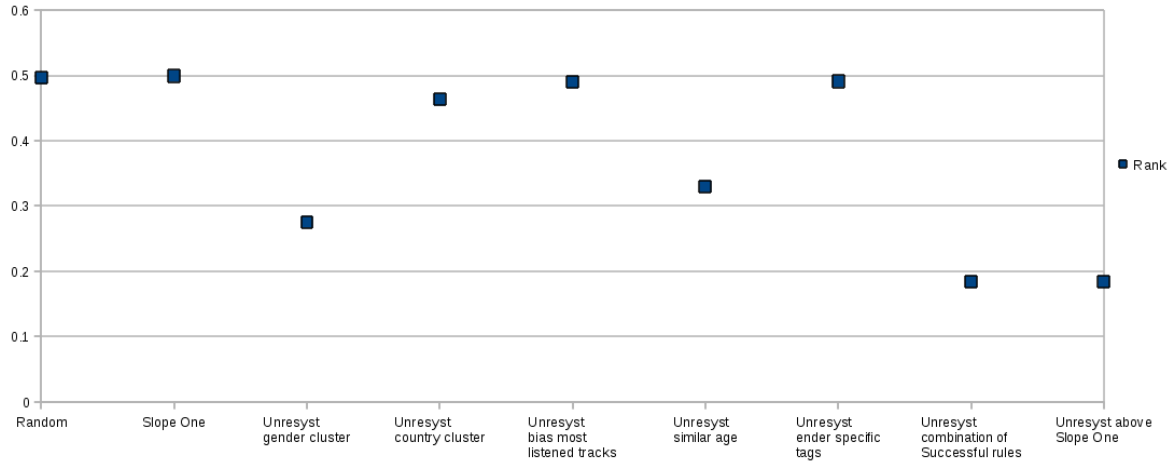


Figure 7.2: The evaluation results on the Last.fm data set. Smaller Rank values are better.

For the Last.fm data set we have introduced two recommenders: one for predicting novel (previously unheard) artists and a radio recommender for predicting any artists, no matter if the user has listened to them or not. As the number of repeatedly played artists in the dataset was relatively low, the evaluation results of the radio recommender were very close to the novel artist recommender results. Therefore we present only the results of the novel artist recommender.

The baseline random recommender gave a rank of 0.5 as expected. The Slope One recommender wasn't much successful at the data set, its result was almost the same as the result of the random recommender. A breakthrough was done by adding clusters for user gender. Obviously, some music is popular within groups of users of the same gender. The country cluster performed just a bit better then the random recommender.

Recommending the most listened tracks wasn't much successful, giving results just below the random recommender. On the opposite, recommending artists listened by users with similar age was much more successful. The proposed rule

with gender-specific tags didn't bring much success, obviously a better analysis should be performed.

The combination of all the successful rules gave a very good result, which wasn't improved by using the Slope One recommender below Unresyst.

7.3.2 Flixster

For the Flixster data set, we used the RMSE metric, as there's explicit feedback available. The results can be seen on the chart 7.3. As a baseline we took a random recommender, that predicted for all pairs expectancy value 0.5. All tested configurations were better than this baseline.

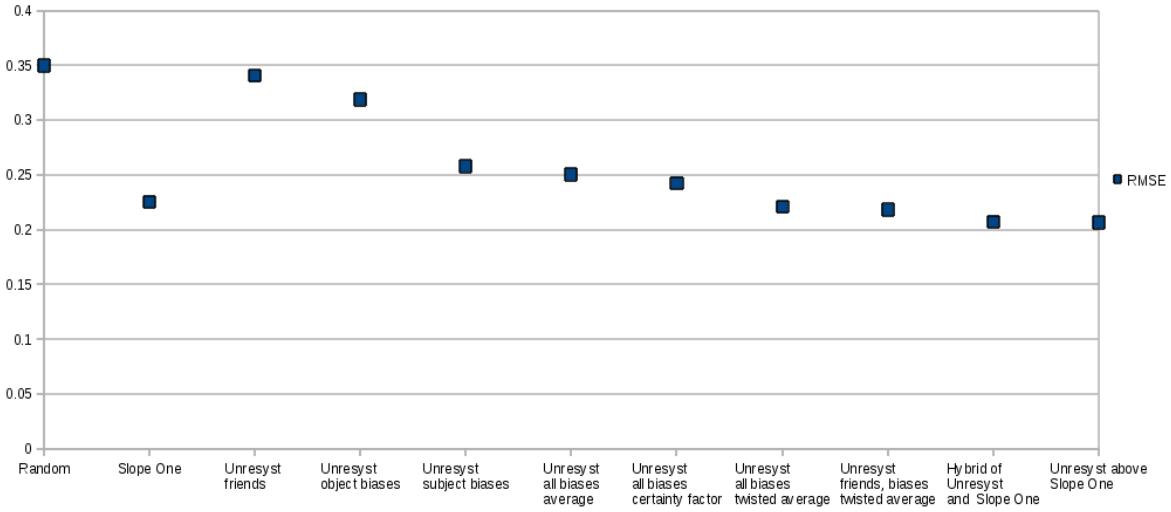


Figure 7.3: The evaluation results on the Flixster data set. Smaller RMSE values are better.

The external Slope One recommender scored pretty well at the data set, the RMSE was significantly better than the random and also outperformed most of the Unresyst configurations. A good result from the Slope One recommender was expected as the Flixster data set is a classic data set with explicit user-item ratings, for which collaborative filtering methods are designed.

Then we evaluated Unresyst, while adding different data sources. Using the friend relationship didn't bring much accuracy benefit, the result was just a bit better than the random. Object biases (giving higher expectancy to highly rated movies and lower to low-rated) were slightly more successful. Much more important were subject biases, obviously some users tend to give constantly high rating to some movies, while some others give constantly low ratings to all movies.

Then we experimented with combining biases using the three combination functions: Average (as a baseline), Certainty Factor and Twisted average, as described in the section 5.3.4. Twisted average outperformed the remaining ones. The twisted average bias combination was even better than the Slope One recommender.

Then we added the friends relationship, which didn't make much difference. Finally we experimented with mixing the Unresyst knowledge based recommender

with the Slope One recommender. The hybrid recommender used Slope One predictions for pair where no rules could be applied. This approach slightly improved the mere Unresyst recommender. Lastly, we tested the knowledge based recommender above Slope One as proposed in the chapter 3.6.1. The evaluation results confirms that the predictions of the knowledge based recommender can be successfully used as an input of a collaborative filtering algorithm.

A table of all evaluation results for the Flixster data set can be found in the Appendix table 8.2.

7.3.3 Travel Agency

For the travel agency data set we used the $\overline{Rank}$ metric as described in the section 7.1. The baseline random recommender result was around 0.5 which was the expected value.

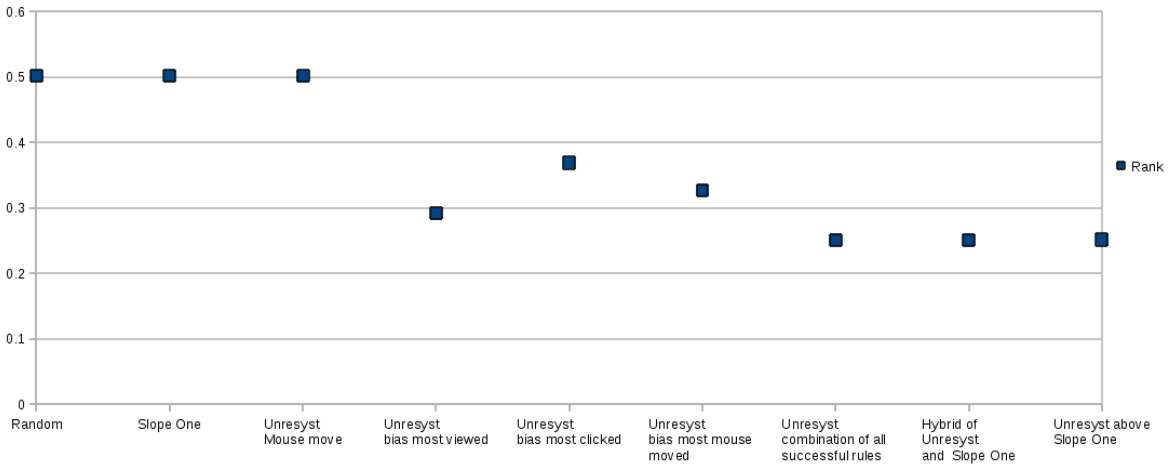


Figure 7.4: The evaluation results on the Travel data set. Smaller $\overline{Rank}$ values are better.

The Slope One recommender wasn't successful on the data set, the obtained result was the same as for the random recommender. The reason was that the number of orders was pretty low and therefore the user-item matrix was very sparse.

Then we tested Unresyst with single data sources. Clusters for countries and tour types weren't successful, giving values around 0.5, nor were the rules obtained from mouse moves, questions and profile views. The first successful attempt were object biases, most successful of them being bias that increased the expectancy for tours that were viewed the most.

The combination of all successful rules further improved the $\overline{Rank}$. We tried using different combination functions to combine rules, but the effect wasn't obvious - all implemented combination functions gave very similar results. We suppose it's because there were little subject-object pairs where multiple rules would meet.

Adding Slope One to a hybrid recommender as well as using Slope One below Unresyst, didn't bring any significant improvement. Obviously the collaborative

filtering methods aren't much useful for data sets with sparse data, even when we try to increase the density by a knowledge based recommender.

A table of all evaluation results for the travel agency data set can be found in the Appendix table 8.2.

7.3.4 Improving the Recommender Results

An obvious way to improve Unresyst results on a given data set is to set weights of the rules, so that the important rules that heavily influence the recommendations get high weights and vice versa.

Notably, weight values are important when predicting explicit feedback values, as in the Flixster data set. When we predict a value of an explicit feedback we care about the exact expectancy value, not just the relative order of the object predictions for a subject.

When we evaluated Unesyst on the data sets, we firstly tried adding a single rule to the recommender and according to the evaluation result, we set the rule weight in the final setup.

Another option would be to observe the coherence between the rule instances and the known predicted relationship pairs. A positive rule that covers many predicted relationship pairs and few others should get a high value, e.g. if user's viewed tours cover all ordered tours and not many additional ones, the rule should get high weight. The rule shouldn't be suspended for additional covered pairs as much as for missing coverage, because the uncovered pairs may become being in the predicted relationship, e.g. a user can order a tour he/she has previously viewed. On the contrary negative rules should cover many pairs uncovered by the predicted relationship and few covered.

Similarly, the positive bias rules should cohere with the number of predicted relationships in which the subject or object is. E.g. if we use highly viewed tours as a bias, it should cohere with highly ordered tours.

For similarity rules and clusters, a high-weighted rule should cover objects (or subjects if it's a subject similarity rule), that are in the predicted relationship with a single subject (or object). I.e. artist similarity should cover artists that are often played together by a single user. Such similarity should obtain high weight.

8. Conclusion and Future Work

The chapter summarizes the thesis and gives ideas for future development of the Unresyst recommender.

8.1 Conclusion

In the thesis we have analyzed the problem of a universal recommender. We have proposed design and a prototype of Unresyst, a universal recommender system, taking multiple sources of preference and similarity as an input. Unresyst combines a knowledge-based recommender with collaborative filtering in a way that is according to our belief novel. The Unresyst interface isolates the business knowledge from recommender algorithms. The proposed architecture for Unresyst is highly modular and enables using various algorithms under the business knowledge layer.

We have designed an interface for entering business rules that can be used for both explicit and implicit user feedback. For entering the strength of the rules we have introduced the expectancy concept, which is a way of representing confidence of both positive and negative rules. We have analyzed methods for combining multiple rules that can be applied to a single entity or entity pairs.

The design concepts were verified by implementing a prototype that was adapted to data sets from various domains. We proposed ways of measuring Unresyst recommendation accuracy and used it for our prototype implementation.

The evaluation results show that a knowledge based recommender can outperform a common collaborative filtering algorithm. When used with in a combination with collaborative filtering, such recommender can give even more accurate results.

8.2 Future Work

We propose a set of actions to continue the work done in the thesis. Additionally to the implementation, we propose some more activities in the universal recommender research.

Implementation The Unresyst recommender should be implemented in a fast and scalable way, so that it can be used in a real world system. The implementation should deal with technical matters like a deeper integration of Unresyst with a collaborative filtering framework. We should be able to integrate the Unresyst implementation to systems running on a wide range of platforms or it should be able to run as a recommender service.

More Domain Experiments Even though we have experimented with Unresyst on a range of various domains, some more experiments are needed, including collecting user feedback to provided recommendations. Some experiments

beyond the traditional user-item recommendations can be performed, like recommending to user groups, or recommending multiple kinds of items. With enough hardware resources, tests on larger data sets can be provided, with a complete evaluation.

More Rule Combination Methods The rule combination methods presented in the section 5.3.4 can be replaced by more sophisticated ones as Bayesian networks.

Recommender Algorithm Modifications The traditional collaborative filtering algorithms can be modified so that they take as an input the data sources provided by the Unresyst interface. The modified algorithms can work on one of the levels proposed in the section 5.4.3.

Learning Weights of the Rules The concept of automatic rule weight setting presented in the section 7.3.4 can be further studied and expanded, so that the weight setting is done during the build of the recommender. Experiments with more sophisticated learning methods can be performed.

Learning New Rules Additionally to learning weights of the existing rules, data mining methods can be used for finding new rules in parent system data. The found rules can be discussed with domain experts and eventually added to the Unresyst configuration.

Bibliography

[1] *About Last.fm recommendation service*. Last.fm Ltd.[cit. 2011-04-11] Available at URL:<http://www.last.fm/about>

[2] Kumar, Ravi; Raghavan, Prabhakar; Rajagopalan Sridhar; Tomkins Andrew. *Recommendation systems: a probabilistic analysis*. 39th Annual Symposium on Foundations of Computer Science 1998. Available at URL:<http://www.tomkinshome.com/site_media/papers/papers/KRR+98.pdf>. ISBN 0-8186-9172-7.

[3] Schafer, Ben; Frankowski, Dan, Herlocker, Joe; Sen, Shilad. *Collaborative Filtering Recommender Systems*. The Adaptive Web, LNCS 4321, 291 - 324, 2007. Available at URL:<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.130.4520&rep=rep1&type=pdf>.

[4] Abhinandan, Das; Mayur, Datar; Ashutosh Garg; Rajaram, Shyam. *Google News Personalization: Scalable Online Collaborative Filtering*. WWW 2007 / Track: Industrial Practice and Experience 2007. Available at URL:<http://iws.seu.edu.cn/resource/Proceedings/WWW/2007/papers/paper570.pdf>. ACM 978-1-59593-654-7/07/0005.

[5] Linden, Greg; Smith, Brent; York, Jeremy. *Amazon.com Recommendations Item-to-Item Collaborative Filtering*. Published by the IEEE Computer Society 2003. Available at URL:<http://www.win.tue.nl/~laroyo/2L340/resources/Amazon-Recommendations.pdf>. IEEE Computer Society 1089-7801/03.

[6] Park, Seung-Taek; Pennock, David M. *Applying collaborative filtering techniques to movie search for better ranking and browsing*. KDD '07 Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, 2007. Available at URL:<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.8109&rep=rep1&type=pdf>. ISBN 978-1-59593-609-7.

[7] *The Duine Recommender Framework*. Telematica Instituut/Novay. [cit. 2011-04-11] Available at URL:<http://duineframework.org/index.html>

[8] *Overview of Mahout*. Apache Software Foundation.[cit. 2011-04-11] Available at URL:<https://cwiki.apache.org/confluence/display/MAHOUT/Overview>

[9] Adomavicius, Gediminas; Tuzhilin, Alexander. *Towards the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions*. IEEE Transactions On Knowledge and Data Engineering, vol. 17, no. 6, June 2005. Available at URL: <http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.107.2790&rep=rep1&type=pdf> IEEE Computer Society 1041-4347/05

[10] Bennett, James, Lanning, Stan. *The Netflix Prize*. KDDCup'07, August 12, 2007. Available at URL:<http://citeseerx.ist.psu.edu/viewdoc/

download?doi=10.1.1.117.8094&rep=rep1&type=pdf> ACM 978-1-59593-834-3/07/0008

[11] Koren, Yehuda. *The BellKor Solution to the Netflix Grand Prize*. Netflix prize documentation, 2009. Available at URL:<http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf>

[12] Bell, Robert M.; Koren Yehuda. *Lessons from the Netflix prize challenge*. ACM SIGKDD Explorations Newsletter 2007, 10.1145/1345448.1345465. Available at URL:<http://www.sigkdd.org/explorations/issues/9-2-2007-12/6-Netflix-1.pdf>

[13] Hunt, Neil. *Netflix Prize Update*. Netflix Blog 2010 [cit. 2011-04-11]. Available at URL: <http://blog.netflix.com/2010/03/this-is-neil-hunt-chief-product-officer.html>.

[14] *The Google Prediction API*. Google [cit. 2011-04-11]. Available at URL:<http://code.google.com/apis/predict/>.

[15] *The Advanced Universal Recommendation Architecture (AURA)*. Oracle Labs [cit. 2011-04-11]. Available at URL:<http://labs.oracle.com/projects/dashboard.php?id=196>.

[16] *The AURA project wiki*. Oracle Corporation [cit. 2011-04-11]. Available at URL:<http://kenai.com/projects/aura/pages/Home>.

[17] Hueter Geffrey J.; Quandt, Steven C.; Hueter, Noble H. *Universal system and method for representing and predicting human behavior*. #20090248599, United States Patent Application Publication, 2009. Available at URL:<http://www.freepatentsonline.com/20090248599.pdf>.

[18] Peška, Ladislav. *User preferences in the domain of web shops*. Master thesis (Mgr.), Charles University in Prague, Faculty of Mathematics and Physics, 2010.

[19] Kunegis, Jerome; Said, Alan; Umbrath, Winfried. *White Paper: The Universal Recommender*. Arxiv preprint arXiv:0909.3472, 2009. Available at URL:<http://uni-koblenz.de/~kunegis/paper/kunegis-universal-recommender.pdf>.

[20] Glaser, William T.; Westergren, Timothy B.; Stearns, Jeffrey P.; Kraft, Jonathan M. *Consumer item matching method and system*. United States Patent 7003515. Available at URL:<http://www.freepatentsonline.com/7003515.pdf>.

[21] Eckhardt, Alan; Horváth, Tomáš; *Vojtáš*, Peter. *PHASES: A user Profile Learning Aproach for Web Search*. WI '07, 780-783. IEEE Computer Society Washington. Available to subscribers at URL:<http://portal.acm.org/citation.cfm?id=1331740.1331829> ISBN 0-7695-3026-5

[22] Koren, Yehuda; Bell, Robert; Volinsky, Chris. *Matrix factorization techniques for recommender systems*. Published by the IEEE Computer Society, 0018-9162/09, 2009. Available at URL:<http://research.yahoo.com/files/ieeecomputer.pdf>

[23] Martínez, Luis; Pérez, Luis G.; Barranco, Manuel J.; Espinilla, M. *A Knowledge Based Recommender System Based on Consistent Preference Relations*. Studies in Computational Intelligence, 2008, Volume 117/2008, 93-111, DOI: 10.1007/978-3-540-78308-4_6 Available for subscribers at URL: <http://www.springerlink.com/content/m64g474m7p0t4r26/>.

[24] Burke, Robin. *Hybrid Web Recommender Systems*. The Adaptive Web, LNCS 4321, pp. 377 - 408, 2007. Available at URL: <http://josquin.cs.depaul.edu/twiki/pub/Csc594/ArticleList/burke-hybrid.pdf>

[25] Burke, Robin. *Knowledge-based recommender systems*. Department of Information and Computer Science, University of California, Irvine 200, To Appear in the Encyclopedia of Library and Information Science. Available at URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.3078&rep=rep1&type=pdf>

[26] Burke, Robin. *Integrating Knowledge-based and Collaborative-filtering Recommender Systems*. AAAI Technical Report WS-99-01, 1999, AAAI. Available at URL: <http://www.aaai.org/Papers/Workshops/1999/WS-99-01/WS99-01-011.pdf>.

[27] Cohen, William W.; Schapire, Robert E.; Singer, Yoram. *Learning to order things*. Journal of Artificial Intelligence Research 10 (1999) 234-270. Available at URL: <http://www.jair.org/media/587/live-587-1790-jair.pdf>

[28] Anderson, Michelle; Lemire, Daniel; Ball, Marcel; Boley, Harold; Greene, Stephen; Howse, Nancy; McGrath, Sean. *RACOFI: A Rule-Applying Collaborative Filtering System*. Proc. IEEE/WIC COLA'03, Halifax, Canada, October 2003. Available at URL: <http://www.daniel-lemire.com/fr/documents/publications/racofi_nrc.pdf> NRC 46507

[29] Celma, Oscar. *Last.fm Dataset - 1K users*. [cit. 2011-04-11]. Available at URL: <http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html>.

[30] Lamere, Paul. *LastFM-ArtistTags2007*, Dataset. [cit. 2011-04-11]. Available at URL: <http://musicmachinery.com/2010/11/10/lastfm-artisttags2007/>.

[31] Jamali, Mohsen. *Flixster Data Set*. [cit. 2011-04-11] Available at URL: <http://www.cs.sfu.ca/~sja25/personal/datasets/>.

[32] Jamali, Mohsen; Ester, Martin. RecSys '10 Proceedings of the fourth ACM conference on Recommender systems *A matrix factorization technique with trust propagation for recommendation in social networks* Available for subscribers at URL: <http://portal.acm.org/citation.cfm?id=1864708.1864736> ISBN: 978-1-60558-906-0

[33] Boley, Harold. *POSL: An Integrated Positional-Slotted Language for Semantic Web Knowledge* Faculty of Computer Science University of New Brunswick, Working Draft 11 May 2004. Available at URL: <http://ruleml.org/submission/ruleml-shortation.html>

[34] Shortliffe, Edward Hance. *Computer-based medical consultation MYCIN.* America Elsevier Publishing Company, Inc., New York 1976. Available as a book at URL: <http://www.annals.org/content/85/6/831.1.extract> ISBN 978-0444001795

[35] Lucas, P.J.F. *Certainty-factor-like structures in Bayesian belief networks.* Knowledge-Based Systems 14, 2001, 327-335 Available at URL:<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.1853&rep=rep1&type=pdf> PII S 0950-705 1(00)00073-3

[36] Gamma, Erich; Helm, Richard; Johnson Ralph; Vlissides, John M. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional; 1 edition (November 10, 1994) ISBN 978-0201633610

[37] Herlocker, Jonathan L.; Konstan, Joseph A.; Terveen, Loren G.; Riedl, John T. *Evaluating collaborative filtering recommender systems.* ACM Transactions on Information Systems (TOIS), Volume 22 Issue 1, January 2004. Available at URL:<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.5270&rep=rep1&type=pdf>

[38] McNee, Sean M.; Riedl, John; Konstan, Joseph A. *Being accurate is not enough: how accuracy metrics have hurt recommender systems.* CHI 2006, April 22-27, 2006, Montreal, Canada. Available at URL: <http://www.grouplens.org/papers/pdf/mcnee-chi06-acc.pdf>

[39] Valizadegan, Hamed; Jin, Rong; Zhang, Ruofei; Mao, Jianchang. *Learning to Rank by Optimizing NDCG Measure.* Advances in Neural Information Processing Systems 22, 1883-1891, 2009. Available at URL:<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.8402&rep=rep1&type=pdf>.

[40] Hu, Yifan; Koren, Yehuda; Volinsky, Chris. *Collaborative Filtering for Implicit Feedback Datasets.* 2008 Eighth IEEE International Conference on Data Mining. Available at URL: <http://www2.research.att.com/~yifanhu/PUB/cf.pdf> ISBN: 978-0-7695-3502-9

[41] Lemire, Daniel; Maclachlan, Anna. *Slope One Predictors for Online Rating-Based Collaborative Filtering* SIAM Data Mining (SDM'05), Newport Beach, California, April 21-23, 2005 Available at URL: <http://arxiv.org/pdf/cs.DB/0702144> arXiv:cs/0702144v2

List of Abbreviations and Terms

API Application programming interface, a set of methods implemented by a software system that can be used by another system.

CSV Comma-separated values, a simple format of text files where the values are separated by commas.

GUI Graphical User Interface, a type of user interface that uses graphics when communicating with user, rather than textual commands.

ORM Object-relational mapping, a technique used for manipulating data directly in the application without using SQL.

RMSE Root Mean Square Error, a metric for evaluating accuracy of recommender systems.

Scrobble Scrobble is a verb used in the Last.fm system. To scrobble means to send information about what song a user is playing to the Last.fm server.

SQL Structured Query Language a language for data manipulation in a database.

Unresyst Universal Recommender System proposed in the thesis.

XML Extensible Markup Language, a format of text documents in machine-readable form.

Attachments

The text contains the following attachments:

1. Unresyst CD
2. List of Tools and Other Resources Used
3. Recommender Evaluation Result Tables

Unresyst CD

The CD is fixed to the thesis book cover. The full list of its contents and brief instructions on how to use it, can be found in the file `readme.txt` on the CD in the root folder. The CD contains:

1. Complete source codes of the Unresyst implementation prototype.
2. The source code of the Slope One recommender, using the Mahout framework, including the framework setup files and scripts for running the recommender on Unresyst exported data.
3. Scripts for reducing the data sets and saving data from them into the system database.
4. Generated programming documentation to the Unresyst implementation prototype.
5. Complete data models both for Unresyst and applications covering the data sets.
6. This text in the *PDF* format.
7. Other resources as diagrams and presentations.

All the named files can be found on the Unresyst project homepage at URL: `<http://code.google.com/p/unresyst/>`.

List of Tools and Other Resources Used

During the work on the thesis we used the following software tools and resources:

1. *Python 2.6.5* for building the prototype implementation.
2. The ORM layer of the *Django 1.2* web application framework, to operate with the database.
3. The *MySQL 14.14* database (can be interchanged with any other database supported by the Django framework).
4. The *gedit* text editor for writing the source code.
5. The *pdfTeXk 1.40.3* for building the PDF from LaTeXtext.
6. *Kile 2.0.85* for writing the thesis text in LaTeX.
7. The drawings were overtaken from the *Tim Tim picture database* of freely available drawings. The drawings are available at URL: `<http://www.timtim.com/>`.
8. For hosting the source codes in the *SVN* repository, we used the *Google Code* service. Instructions on how to checkout the repository are available at URL: `<http://code.google.com/p/unresyst/source/checkout>`.

All the tools are freely available, under open source licenses. The named services are free.

The source code of the prototype implementation is available under *Mozilla Public License 1.1*, see URL: `<http://www.mozilla.org/MPL/>`. Other contents as the thesis text are available under *Creative Commons 3.0 BY-SA*, see URL: `<http://creativecommons.org/licenses/by-sa/3.0/>`.

Recommender Evaluation Result Tables

The attachment contains complete lists of experiments done on the datasets. The interpretation of the results including charts can be found in the section 7.

Recommender Setup	$\overline{Rank}$
Empty random recommender	0.496207
External Slope One recommender	0.499361
Unresyst, gender cluster	0.275451
Unresyst, country cluster	0.463337
Unresyst, listened tracks bias	0.490121
Unresyst, similar age rule	0.330009
Unresyst, gender specific tag rule	0.512404
Unresyst, similarity based on tags	0.490963
Unresyst, combination of successful rules gender cluster (weight 0.5), country cluster(weight 0.1), similar age (0.4)	0.184017
Unresyst, combination of successful rules above Slope One gender cluster (weight 0.5), country cluster(weight 0.1), similar age (0.4)	0.184017

Table 8.1: Results for evaluating recommenders on the Last.fm data set.

Recommender Setup	*RMSE*
Empty random recommender	0.349860
External Slope One recommender	0.225575
Unresyst, friends (weight 0.5)	0.332549
Unresyst, friends (weight 1.0)	0.337328
Unresyst, friends (weight 0.2)	0.340405
Unresyst, friends (weight 0.65)	0.340405
Unresyst, object biases (weight 0.5)	0.325602
Unresyst, object biases (weight 1.0)	0.318892
Unresyst, object biases (weight 0.75)	0.319917
Unresyst, just positive object biases (weight 1.0)	0.318768
Unresyst, just negative object biases (weight 1.0)	0.349974
Unresyst, subject biases (weight 0.5)	0.294145
Unresyst, subject biases (weight 1.0)	0.257785
Unresyst, just positive subject biases (weight 1.0)	0.257919
Unresyst, just negative subject biases (weight 1.0)	0.349761
Unresyst, all biases (weights 1.0 pos., 0.25 neg.), average	0.349761
Unresyst, all biases (weights 1.0 pos., 0.25 neg.), certainty factor	0.242736
Unresyst, all biases (weights 1.0 pos., 0.25 neg.), twisted average	0.220870
Unresyst, all biases (weights 1.0 pos., 0.25 neg.), friends (weight 0.65), average	0.245660
Unresyst, all biases (weights 1.0 pos., 0.25 neg.), friends (weight 0.65), certainty factor	0.239204
Unresyst, all biases (weights 1.0 pos., 0.25 neg.), friends (weight 0.65), twisted average	0.227061
Unresyst, all biases (weights 1.0 pos., 0.25 neg.), friends (weight 0.1), twisted average	0.218237
Hybrid of Unresyst and Slope One, all biases (weights 1.0 pos., 0.25 neg.), friends (weight 0.1), twisted average	0.207439
Slope One below Unresyst, all biases (weights 1.0 pos., 0.25 neg.), friends (weight 0.1), twisted average	0.206578

Table 8.2: Results for evaluating recommenders on the Flixster data set.

Recommender Setup	$\overline{Rank}$
Empty random recommender	0.501684
External Slope One recommender	0.501684
Unresyst with tour type cluster	0.501684
Unresyst with tour country cluster	0.501684
Unresyst with mouse move rule	0.501186
Unresyst with view profile rule	0.501186
Unresyst with question rule	0.501684
Unresyst with click rule	0.501454
Unresyst with bias - most viewed tours	0.291246
Unresyst with bias - most clicked tours	0.368687
Unresyst with bias - most mouse moved tours	0.326599
Unresyst with bias - most mouse moved tours	0.326599
Unresyst with combination mouse move (weight 0.1), view profile (weight 0.1), click (weight 0.05), bias viewed (weight 0.3), bias clicked (weight 0.25), bias mouse moved (weight 0.2), average	0.250765
Unresyst with combination mouse move (weight 0.1), view profile (weight 0.1), click (weight 0.05), bias viewed (weight 0.3), bias clicked (weight 0.25), bias mouse moved (weight 0.2), certainty factor	0.250765
Unresyst with combination of mouse move (weight 0.1), view profile (weight 0.1), click (weight 0.05), bias viewed (weight 0.3), bias clicked (weight 0.25), bias mouse moved (weight 0.2), twisted average	0.250765
Hybrid of Unresyst and Slope One with combination of mouse move (weight 0.1), view profile (weight 0.1), click (weight 0.05), bias viewed (weight 0.3), bias clicked (weight 0.25), bias mouse moved (weight 0.2), twisted average	0.250765
Slope One below Unresyst with combination of mouse move (weight 0.1), view profile (weight 0.1), click (weight 0.05), bias viewed (weight 0.3), bias clicked (weight 0.25), bias mouse moved (weight 0.2), twisted average	0.251148

Table 8.3: Results for evaluating recommenders on the travel agency data set.