Problem Description

Study how to adaptively aggregate recommender systems on a per-user and per-item basis when combining results from complementing prediction methods. Create a flexible algorithm that combines multiple predictions into one coherent result. Utilize the resulting aggregation algorithm to provide personalized search in an information retrieval system.

Assignment given: January 17th, 2011
Supervisor: Asbjørn Thomassen

Adaptive Aggregation of Recommender Systems

By Olav Frihagen Bjørkøy
(olavfrih@stud.ntnu.no)

Supervised by Asbjørn Thomassen
(asbjornt@idi.ntnu.no)

Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway

Abstract

In the field of artificial intelligence, *recommender systems* are methods for predicting the relevance items to a users. The items can be just about anything, for example documents, articles, movies, music, events or other users. Recommender systems examine data such as ratings, query logs, user behavior and social connections to predict what each user will think of each item.

Modern recommender systems combine multiple standard recommenders in order to leverage disjoint patterns in available data. By combining different methods, complex predictions that rely on much evidence can be made. These aggregations can for example be done by estimating weights that result in an optimal combination.

However, we posit these systems have an important weakness. There exists an underlying, misplaced subjectivity to relevance prediction. Each chosen recommender system reflects one view of how users and items *should* be modeled. We believe the selection of recommender methods should be automatically chosen based on their predicted accuracy for each user and item. After all, a system that insists on being adaptive in one particular way is not really adaptive at all.

This thesis presents a novel method for prediction aggregation that we call *adaptive recommenders*. Multiple recommender systems are combined on a per-user and per-item basis by estimating their individual accuracy in the current context. This is done by creating a secondary set of error estimating recommenders. The core insight is that standard recommenders can be used to estimate the accuracy of other recommenders. As far as we know, this type of adaptive prediction aggregation has not been done before.

Prediction aggregation (combining scores) is tested in a recommendation scenario. Rank aggregation (sorting results lists) is tested in a personalized search scenario. Our initial results are promising and show that adaptive recommenders can outperform both standard recommenders and simple aggregation methods. We will also discuss the implications and limitations of our results.

Preface

This is a Master Thesis in the field of Artificial Intelligence, as part of my degree in Computer Science at the Norwegian University of Science and Technology (NTNU). My specialization is in the field of intelligent systems, at the Department of Computer and Information Science (IDI), in the faculty of Information Technology, Mathematics and Electrical Engineering (IME).

Parts of this thesis is based on my previous work in the same field, *User Modeling on The Web: An Exploratory Review*. A short version of this thesis is also available. See Appendix B for more information.

I would like to thank my supervisor, assistant professor Asbjørn Thomassen, for valuable guidance and feedback throughout the process. In addition, thanks are in order for my fellow students Kim Joar Bekkelund and Kjetil Valle, who helped me formulate my thoughts and provided feedback on the work represented by this thesis.

To limit the scope of an already extensive topic, this document assumes a basic knowledge of set theory, graph theory, linear algebra and fundamental concepts in artificial intelligence on behalf of the reader.

Trondheim, June 13th, 2011,
Olav Frihagen Bjørkøy

Contents

— 1 —
Introduction

As people get access to more and more information, a common problem often occurs. Having too much information can be as harmful as having no information at all. While having too little is an obvious problem, too much leads to information overload, where relevant content drowns in irrelevant noise. Our ability to make informed decisions is often the first thing to go (Davenport and Beck, 2001, p.1).

While people struggle with excessive information, many algorithms in *artificial intelligence* can increase their performance by accessing more information. Halevy and Norvig (2009) calls this the "unreasonable effectiveness of data". Perhaps surprisingly, more data often trumps more efficient algorithms. For example, Banko and Brill (2001, p.3) show how common algorithms in AI can substantially improve by giving them a lot more data to work with. As much as researchers chase elegant algorithms, finding more data to work with may be time better spent.

Few places is this difference of users and computers more apparent than in *recommender systems*. A recommender system is a technique in user modeling to estimate the relevance of an item to a user (see Figure 1.1). The items can be just about anything, for example documents, websites, movies, events or other users. These systems use data such as search query logs, ratings from similar users, social connections and much more to predict unknown relevance, as we shall see. Recommender systems are especially prolific on the Web, for example in personalized recommendations of news, books, movies, articles, social connections, search results, et cetera. Unlike search engines, which lets users search for known or desired information, recommender systems strive to present each user with previously unknown yet interesting items.

Modern recommender systems often embrace the unreasonable effectiveness of data by combining multiple algorithms that predict relevance in various ways. By considering different aspects of users and items when making predictions, aggregation methods provide quite complex predictions that rely on much evidence. For example, Bell et al. took this approach to its logical conclusion by combining 107 different recommender systems when winning the Netflix movie recommender challenge (see Bennett and Lanning (2007)).

While the name "recommender systems" might seem limiting, they are incredibly powerful tools. If we can accurately predict how users will react to items, we will have come a long way towards solving the information overload problem.

Despite their apparent power, recommender systems are often confined to simple tasks like creating small lists of recommended items or computing similar items to the ones being considered. Examples include recommending new social connections, or suggesting news articles based on previous reading. Seldom are the full potential of recommender systems reached by creating completely adaptive systems that work hard to mitigate any signs of information overload.

modeling:	data	$\longrightarrow$	recommender system	$\longrightarrow$	recommender model
predictions:	user, item	$\longrightarrow$	recommender model	$\longrightarrow$	predicted rating

Figure 1.1: A simplified view of recommender systems: Ratings of items by users are used to create a model. *This model is then used to predict unknown ratings between users and items. Note that many recommender systems work differently, as we shall see later in this thesis.*

We posit that traditional recommender systems have an important weakness. There exists an underlying, misplaced subjectivity to relevance prediction. We believe this fundamental weakness hinders their usefulness, as there is a mismatch between how recommender systems perform predictions, and how predictions actually should be made for each user and item.

Consider this: when an algorithm is selected for use in a recommender system, there is a concious decision of which predictive data pattern to use. Before any user modeling is performed, the researcher or developer selects one or more methods that is thought to best model every user and item in the system. While the methods themselves may perform well, their selection reflects how whoever created the system assumes how each user can and *should* be modeled. This underlying subjectivity is not desirable. We call this the *latent subjectivity problem*.

Examples are not hard to come by. For instance, while one user might appreciate social influence in their search results, another user might not. While one user might find frequency of communication maps well to relevance, another might not. One user may think the similarity of movie titles is a good predictor, while another might be more influenced by their production year. Some users may favor items rated highly on a global scale, while others are more interested in what users similar to themselves have to say.

The same problem exists for the items that should be recommended. For example, while one item might best be judged by its content, another might be better described by previous ratings from other users. One item's relevance may be closely tied to when it was created, while other items may be timeless. The exact differences are not important, only that they exist.

Another way to put this is that *recommender systems are dependent on the subjective assumptions of their creators*. A recommendation method use certain aspects of available data to make predictions, and these aspects are chosen by whoever creates the system.

Modern aggregation approaches face the same problem. Aggregation is done on a generalized, global level, where each user and item is expected to place the same importance on each modeling method. While the aggregation is selected to minimize some error over a testing set, the subjective nature remains. The generalized aggregation treats all users and items the same — hardly a goal of user modeling.

Should it not be up to the users to implicitly decide which method best describes their preferences? And, considering the vast scope of items we can come by, will the selected methods perform optimally for every item? We believe the priority of each algorithm should be implicitly and automatically based on how well they have previously worked for individual users and items. Without this adaptability, it may be hard for recommender systems to perform well in scenarios with widely differing users and items. The scope of users and items is simply too great for any one or generalized combination of methods to capture the nuanced nature of relevance prediction.

This thesis proposes a novel aggregation method that we call *adaptive recommenders*, where the selection of algorithms is implicitly made by the users and items. This provides an extra level of abstraction and personalization. The selection decisions are implicit, and happens in the background, without any extra interaction required. This leaves the subjective nature of selecting ways to model users and items where it should be. That is, in the hands of individual users, and dependent on specific items.

This adaptive selection has an important consequence. If an algorithm is contextually used based on how well it performs, any possibly useful recommender algorithm suddenly becomes a worthy addition. Algorithms of the system are only used in situations they work well, and those that do not will be used in other situations where they might be better suited.

As far as we know, this kind of adaptive prediction aggregation has not been done before. The main research question of this thesis is whether or not adaptive recommenders can outperform traditional approaches.

This thesis is structured as follows. Chapter 2 will present background theory and previous work for the information overload problem, recommender systems, prediction aggregation (combining scores) and rank aggregation (combining sorted lists). We will also briefly introduce the topics of information retrieval and personalized search, that will serve as a case study in later chapters.

Chapter 3 will further discuss the latent subjectivity problem, and build the *adaptive recommenders* approach from the ground up. We will show how this approach can be used for both prediction aggregation and rank aggregation.

Chapter 4 will test three hypotheses and experiment with our newly built method. We will experiment with prediction aggregating for singular items, and explore rank aggregation for personalized search. Finally, Chapter 5 will discuss the implications of our results, important limitations, contributions and suggest future work.

— 2 —

Background Theory

This chapter will introduce previous work and background theory needed to develop our approach to relevance prediction. We will first describe the information overload problem, before delving into how user modeling and recommender systems are currently used to solve this problem. This chapter will also introduce the field of personalized search, where adaptive recommenders will be especially applicable.

2.1 Information Overload

Information overload conveys the act of receiving *too much information.* The problem is apparent in situations where decisional accuracy turns from improving with more information, to being hindered by too much irrelevant data (Bjørkøy, 2010, p.13). This is a widespread phenomenon, with as many definitions as there are fields experiencing the problem. Examples include *sensory overload*, *cognitive overload* and *information anxiety* (Eppler and Mengis, 2004, p.2). When this happens, two common tasks become difficult:

1. Consumption of relevant content is hindered by irrelevant noise.
2. Discovering new and interesting content is difficult due to the amount of information.

Finding contemporary examples is not difficult:

- Missing important news articles that get drowned out by irrelevant content.
- Forgetting to reply to an email as new messages keep arriving.
- Consuming sub-par entertainment because the most relevant is never discovered.
- Reformulating search queries because the results include irrelevant items.
- Browse through much information to find what one is actually looking for.

Information overload is often likened to a *paradox of choice*, as there may be no problem acquiring the relevant information, but rather identifying this information once acquired. As put by (Edmunds and Morris, 2000, p.6): *"The paradox — a surfeit of information and a paucity of useful information."* While normal cases of such overload typically result in feelings of being overwhelmed and out of control, Bawden and Robinson (2009, p.5) points to studies linking extreme cases to various psychological conditions related to stressful situations, lost attention span, increased distraction and general impatience.

Kirsh (2000) argues that *"the psychological effort of making hard decisions about* pushed *information is the first cause of cognitive overload."* According to Kirsh, there will never be a fully satisfiable solution to the problem of overabundant information, but that optimal environments can be designed, in order to increase productivity and reduce the level of stress. This is achieved through careful consideration of each user's needs. To solve the problems of information overload, applications must be able to individually adapt themselves to individual users.

An insightful perspective on information overload comes from the study of *attention economy*. In this context human attention is seen a scarce commodity, offset by how much irrelevant noise is present at any given time. Davenport and Beck (2001, p.1) defines attention as *"[...] focused mental engagement on a particular item of information. Items come into our awareness, we attend to a particular item, and then we decide whether to act"*. To evade information overload means maximising the available attention, allowing more focus on the most important items in each situation.

Conceptual models used in interaction design can also help us see when and where information overload interferes with a user's experience. Norman (2002) advocates a model called *the seven stages of action*, that describes how each user goes through several states while using a system (see Figure 2.1, adapted from Norman (2002)). First, the user forms a goal and an intention to act. The user then performs a sequence of actions on the world (the interface) meant to align the perceived world and the goals. After performing a set of actions, the new world state is evaluated and perceived. At last, the user evaluates the perception and interpretation of the world in accordance with the original goal.

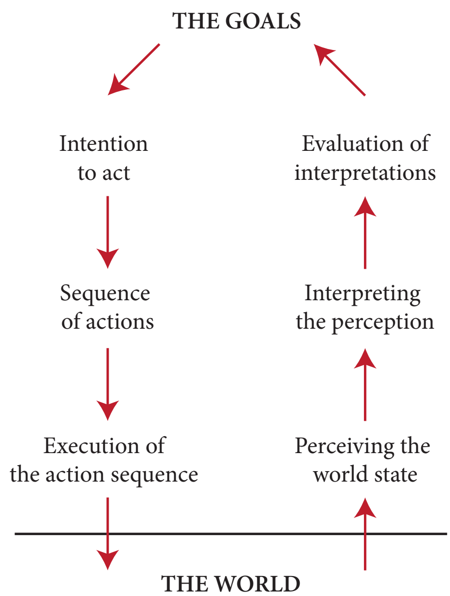


Figure 2.1: Stages of Action

As apparent from this model, information overload can interfere both before and after any action is taken. For example, if the application presents too much content, or presents content in a confusing manner, it can be difficult for the user to identify which actions that would help achieve the current goal. Likewise, after actions are taken, the new world state can suffer the same shortcomings of overwhelming scope or lack of presentations, leading to information overload. This precludes the user from properly evaluating the resulting application state.

In short, an application interface can fail both before and after a user tries to interact with it. Information overload can happen throughout the interaction process.

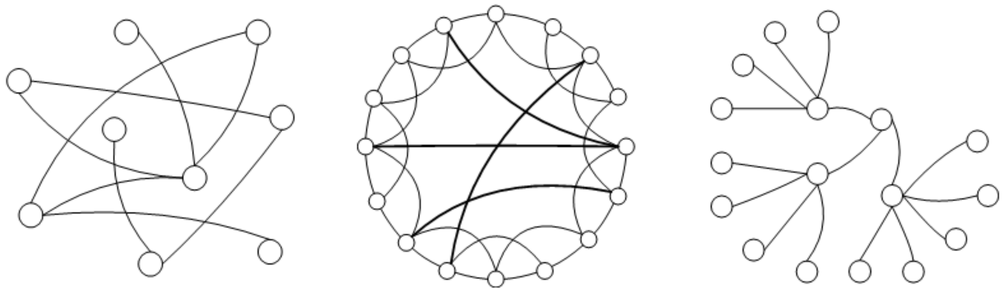


Figure 2.2: Complex Networks, from the left: a random *network, a* small-world *network and a* scale-free *network (which is a type of small-world network). Figure adapted from Huang et al. (2005, p.2).*

2.1.1 Online Overload

The Web is a common source of information overload, and a good example of how and why the problem occurs. Online information overload is especially pervasive when considering *content aggregating websites*, i.e. sites that combine information from multiple other sites and sources. Online information retrieval systems (search engines) are in this category, as are online newspapers, feed readers and portal sites.

The wealth and scope of data on the Web are natural culprits of information overload, as well as the varying qualities of websites publishing the information. However, the problem is also a result of the fundamental observed structure of the Web. Graph theory presents applicable models that characterize how people navigate between websites, and show how content aggregators form important hubs in the network. These models give a theoretical foundation for why information overload occurs on the Web.

In the Web graph, nodes correspond to websites and directed edges between nodes are links from one page to another. The *degree* of a node is defined as its number of edges. It has been observed that this graph has the properties of a *small-world network* ((Newman et al., 2000), (Huang et al., 2005, p.2)), a type of random graph, where most nodes are not neighbors, but most nodes are reachable through a small number of edges (See Figure 2.2). This is because of important random shortcuts differentiating the graph from a regular lattice. The graph is not random, but neither is it completely regular. As described by Barabási (2003, p.37), the average number of outbound links from a web page is around 7. From the first page, we can reach 7 other pages. From the second, 49 documents can be reached. After 19 links have been traversed, about 10^{16} pages can be reached (which is more than the actual number of existing web pages, since loops will form in the graph).

The high degree of the Web graph would suggest that finding an optimal path to your desired page is quite difficult. Yet, while it is true that finding the *optimal path* is hard, finding *a good path* is not that big a challenge. When people browse the Web, links are not followed blindly — we use numerous heuristics to evaluate each link, often resulting in quite a good path to where we want to go. So why is the Web still quite challenging to navigate?

As discovered by Albert et al. (1999), the Web also exhibits properties of a *Scale-Free Network* (SFN). They found that in some natural observed networks, there exists a small number of nodes with an extremely high degree. This is also true on the Web — some websites have a huge number of outbound links. For comparison, while a random network is similar to a national highway system, with a regular number of links between major cities, scale-free networks are more like an air traffic system, with central hubs connecting many less active airports (Barabási, 2003, p.71).

These highly connected nodes, called *hubs*, are not found in small-world networks or random graphs. As demonstrated by the presence of hubs, the degree distribution of a scale-free network follows a power law, $P(k) \sim k^{-\gamma}$, where $P(k)$ is the probability of a node having k connections and γ is a constant dependent on the type of network, typically in the range $2 < \gamma < 3$. Since the Web has directed edges, we have two power laws: $P_{in}(k) \sim k^{-\gamma_{in}}$ and $P_{out}(k) \sim k^{-\gamma_{out}}$.

Albert et al. (1999) describes a number of studies placing the γ values for the Web in the $[2, 3]$ range, with γ_{out} being slightly higher than γ_{in}. Both these probabilities exhibit power tails (or long tails). A few important nodes have a huge number of inbound and outbound links, i.e. the hubs.

Barabási (2003, p.86) proposed that hubs emerge in a scale-free networks because of two factors. *(1) Growth:* nodes are added to the network one by one, for example when new websites are added to the Internet. *(2) Preferential attachment:* when new nodes are created, they connect to existing nodes. The probability that the new node will connect to an existing node is proportional to the number of links the existing node has. Older, more established and central nodes are preferred neighbors.

This is called the Barabási-Albert model (Albert et al., 1999). The probability for a new node connecting to an existing node is given by $\prod k_i$, where k_i is the number of links pointing to node i, in the following equation:

$$\prod_i k_i = \frac{k_i}{\sum_j^N k_j}.$$

Search engines, social link aggregators, news portals, et cetera are all hubs of the Internet, emerging from the preferential link attachment of newly created nodes, that make navigating the Web less easy as it might appear. What does seem clear is that these content aggregating hubs are prime candidates for overwhelming their users with information.

The fundamental observed structure of the Web creates the need for information brokers that link the net together, and the need for techniques to display a lot of data, adapted to each user and each item. In other words, we need user modeling methods that can predict how relevant an item will be to a user.

2.2 User Modeling

The term *user modeling* (UM) lacks a strict definition. Broadly speaking, when an application is adapted in some way based on what the system knows about its users, we have user modeling. From predictive modeling methods in machine learning, to how interface design is influenced by personalization, the field covers a lot of ground.

It is important to differentiate between adapting the interface of an application and the content of an application. Many user modeling methods strive to personalize the interface itself, e.g. menus, buttons and control elements (e.g. Jameson (2009); Fischer (2001)). Adapting the content, on the other hand, means changing how and what content is displayed. For instance, interface adaption might mean changing the order of items in a menu, while content adaption might mean changing the order and emphasis of results in a web search interface (e.g. Xu et al. (2008); Qiu and Cho (2006); Rhodes and Maes (2000)).

In this thesis, we are interested in adapting the *content* of an application. We believe the information overload problem often stems from a mismatch between presented content and desired content. Examples of adaptive content include:

- Suggesting interesting items based on previous activity.
- Reorganizing or filtering content based on predicted user relevance.
- Translating content based on a user's geographical location.
- Changing the presentation of content to match personal preferences or abilities.
- Personalizing search results based on previous queries and clicks.

The fields of Artificial Intelligence (AI) and Human-Computer Interaction (HCI) share a common goal solving information overload through user modeling. However, as described by (Lieberman, 2009, p.6), they have different approaches and their efforts are seldom combined. While AI researchers often view contributions from HCI as trivial cosmetics, the HCI camp tends to view AI as unreliable and unpredictable — surefire aspects of poor interaction design.

In AI, user modeling refers to algorithms and methods that infer knowledge about a user based on past interaction (e.g. Pazzani and Billsus (2007); Smyth (2007); Alshamri and Bharadwaj (2008); Resnick et al. (1994)). By examining previous actions, predictions can be made of how the user will react to future information. This new knowledge is then embedded in a model of the user, which can predict future actions and reactions. For instance, an individual user model may predict how interesting an unseen article will be to a user, based on previous feedback on similar articles or the feedback of similar users.

HCI aims to meet user demands for interaction. User modeling plays a crucial role in this task. Unlike the formal user modeling methods of AI, user models in HCI are often cognitive approximations, manually developed by researchers to describe different types of users (e.g. Fischer (2001); Jameson (2009); Cato (2001)). These models are then utilized by interaction designers to properly design the computer interface based on a

Figure 2.3: Levels of Interface Autonomy: Interfaces range from those only customizable by the user, to intelligent systems takes the initiative on their own accord.

models predictions of its user's preferences. Totterdell and Rautenbach (1990) describes user modeling in interaction design as a collection of deferred parameters:

> *"The designer defers some of the design parameters such that they can be selected or fixed by features of the environment at the time of interaction [...] Conventional systems are special cases of adaptive systems in which the parameters have been pre-set."*

This thesis is concerned with the AI approach to user modeling, and in particular, the use of *recommender systems* (RSs).

2.2.1 Interface Autonomy

Using AI to adapt an interface raises important questions with regard to usability, privacy and usefulness. These questions are rooted in the autonomy expressed by each interface. An autonomous interface is one that takes initiatives on its own, regardless of whether the user has asked for it (Lieberman, 1997, p.2). Naturally, any application that automatically personalizes its content will be autonomous to some degree.

Adaptive interfaces can be classified into increasing order of autonomy (see Figure 2.3). At the order of least autonomous systems, we have *customizable interfaces*. These are interfaces that the user may customize themselves, but that do not take the initiative or change anything without explicit user action. For example, an interface might have a settings panel where users can change the order of items in a menu.

At the next level of autonomy, we have *adaptive interfaces* that suggest to the user possible changes or actions that might be beneficial. For example, an email application could suggest which folder an email should be moved to. At the most autonomous level, *intelligent interfaces* implicitly and automatically customize the interface or content based on passive observation of the user. This could for instance entail automatic filing of emails based on content classification and data mining of previous user actions with similar messages.

An application that personalizes content automatically will fall somewhere in the two last categories and present either an adaptive or intelligent interface, depending on the extent and transparency of its autonomy.

In this thesis, we are only interested in fully autonomous, intelligent interfaces. We will create a system that implicitly, and without any effort from each user, can adapt the content of an application based on previous behavior. Other examples of such implicit user modeling include Qiu and Cho (2006), Shen et al. (2005) and Carmel et al. (2009).

As our goal is to adaptively combine different RSs based on each user and item, we shall now describe what makes a recommender system, and introduce some of the many algorithms they employ.

2.3 Recommender Systems

Recommender systems are powerful and versatile approaches to user modeling. Whenever we wish to predict the relevance of an item to a user, recommender systems are the tools to use. Such systems are commonly used on the Web to provide a host of functionality, for example:

- Suggesting new social connections based on an existing social graph.
- Recommending new and unseen products based on past purchases.
- Ordering news articles by predicted individual relevance.
- Recommending items based the activity of similar or like-minded users.
- Personalizing search results based on the current user's preferences.

Note that although we use the terms *ratings*, *utility*, *preference*, *relevance* and *connection strength* depending on the context, they all refer to the same measure. The terms are measures for what a user thinks of an item, using the domain language of the application in question.

Common to our examples are a set of users, a set of items, and a sparse set of known ratings or relevance measures. The operations of a recommender system is best described through graph operations, although the underlying algorithms might not use this as the representation at all. Mirza and Keller (2003) explain how any RS can be expressed as a graph traversal algorithm. In this graph structure, items and users are nodes, while ratings, social connections et cetera are edges between the nodes. An RS performs predictive reasoning by estimating the strengths of hypothetical connections between nodes that are not explicitly connected.

For example, if a user has rated a subset of the movies in a movie recommendation system, algorithms can use these ratings to predict how well the user will like unseen movies. This inference can for instance be based on each movie's ratings from similar users. In social networks, recommender systems can be used to infer new social relations based on existing connections. The principle is the same. By evaluating current explicit connections, and the connections of similar users, new connections can be predicted.

As evident by the examples, recommender systems are powerful methods for user modeling, personalization and for fighting information overload. Their ability to infer unknown relevance between users and items makes them useful in many situations, as we shall see.

2.3.1 Aspects of Recommender Systems

Formally, a recommender system can be seen as a quintuple, $\mathrm{RS} = (I, U, R, F, M)$, where I is the set of items and U is the set of users. R is the set of known ratings or relevance scores, for example explicit preferences given by users for certain items, or connections in a social graph. We have explicit ratings whenever the user provides their own ratings (e.g. product purchases), and implicit ratings when the system infers ratings from behavior (e.g. query log mining). F is a framework for representing the items, users and ratings, for example with a graph or matrix. M is the actual user modeling algorithm used to infer unknown ratings for predicting a user's preference for an unrated item. This is where AI comes in.

In Adomavicius and Tuzhilin (2005, p.2), M is seen as a utility (the rating, in AI terms) estimation function $p : U \times I \rightarrow S$. Here, p (for prediction) is a function that maps the set of users and items into a fully ordered set of items S, ranked by their utility to each user. In other words, S is the completely specified version of R, where every user has either an explicit, implicit or predicted preference for every item in I. To predict the most relevant unrated item for a user, we simply find the item with the highest expected utility:

$$\forall u \in U,\ i'_u = \arg\max_{i \in I} p(u, i)$$

The utility function p depends on the modeling method being used. The *reason* for using a recommender system is that the utility is not defined for the entire $U \times I$ space, that is, the system does not know the utility of every item to every user. The point of a recommender system is then to extrapolate R to cover the entire user-item space. To be able to rank items according to user preferences, the system must be able to predict a user's reaction to items they have not yet explicitly or implicitly rated themselves.

Another common way of describing and implementing an RS is using a simple matrix. This is what we shall use in this thesis. In this matrix, one dimension represents users and the other represents items. Each populated cell corresponds to a known rating. This matrix corresponds to the framework variable F in our RS quintuple:

$$R_{u,i} = \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,i} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ r_{u,1} & r_{u,2} & \cdots & r_{u,i} \end{pmatrix}$$

Critically, these matrices are usually extremely sparse (most of the cells are empty). While there may be a large number of users and items, each individual user only rates, connects to or uses a few items. This is true for any scenario where users explicitly rate items, access items in search results, or connect to each other in a social network. For example, in the seminal Netflix Challenge movie recommender dataset, almost 99% of the potential user/item pairs have no rating (Bell and Koren, 2007a, p.1). These recommender systems had to be able to produce results from a matrix where only 1% of the cells had meaningful values.

This is often the defining characteristic of a recommender system. An RS is defined by its ability to extract meaningful patterns from sparse data, through dimensionality reduction, neighborhood estimation and many other methods, as we shall see. Naturally, much research looks at ways to best tackle this sparsity problem (e.g. Pitsilis and Knapskog (2009), Claypool et al. (1999, p.3), Ziegler (2005, p.19)).

Recommender systems face many challenges other than this sparsity problem. A directly related problem is the need for large datasets. Since the data is often sparse, the systems will most often perform well if used on large numbers of items and users. In addition, as in many machine learning methods, concept drift (Widmer and Kubat, 1996, p.1), where the characteristics of a user or item changes over time, is another recurring problem.

Another potential problem is that the performance of RSs is often closely tied to their computational complexity (as mentioned in Adomavicius and Tuzhilin (2005, p.6)). Real world usage of the most precise methods can be hindered by the computational power needed to actually put them into production.

Finally, the scale of the data in question should be a concern. If the ratings are ordinal data (e.g. 1-5) input directly by users, the RS should take into account the domain specific meaning of these intervals. For example, in a system for rating movies, the jump between ratings 4-5 might not have the same significance as the jump from 2-3. However, this is a fact seldom mentioned in the literature. Most RSs employ metrics that assume a normal distribution, and even the common evaluation measures such as the RMSE treat ordinal data as a continuous scale. We will get back to these problems in Chapters 4 *&* 5.

2.3.2 Predicting Ratings

The crucial part of any RS is how it predicts unknown ratings. Because of this, each method may be categorized based on certain dimensions of its predictive capabilities (see Table 2.1). In this thesis we will use a taxonomy where these dimensions are (1) the available *data*, (2) the prediction *method*, (3) the model *granularity*, (4) the knowledge *temporality* and (5) the knowledge gathering *agents*.

(1) The data variable represents what data an RS uses to perform predictions. *Content-Based* (CB) methods use only items, intra-item relations, and an individual user's past history as predictive signals of future actions (Pazzani and Billsus, 2007, p.1). By only considering the individual user in adapting an application, highly personal models can be created. However, such methods may require a lot of interaction before reliable models can be created (Adomavicius and Tuzhilin, 2005, p.4). The problem of having to do complex inference from sparse data, as it often is in CB methods, is called the

Variable	*Possible values*
Data	Content-based | Collaborative | Hybrid
Method	Heuristic | Model-based
Granularity	Canonical | Typical | Individual
Temporality	Short-term | Long-term
Agents	Implicit | Explicit

Table 2.1: A taxonomy of recommender systems. From Bjørkøy (2010).

cold start or *new user* problem. This is closely related to the common AI problem of *overfitting*, where the algorithms creates models that match the training data, but not the actual underlying relationships. As with the sparsity problem, a lot of research looks at ways to overcome the new user problem, i.e. achieving "warmer" cold start (e.g. Umbrath and Hennig (2009), Lilegraven et al. (2011)).

Formally, when using content-based predictions, the utility function $p(u, i)$ of user u and item i is extrapolated from $p(u, i_x)$, where i_x is an item similar to i and $p(u, i_x)$ is known (Adomavicius and Tuzhilin, 2005, p.2).

Collaborative Filtering (CF) methods build predictive models for users based on the actions of similar users (Schafer et al., 2007). The key observation is that similar users share a common taste in items. By using data from more than one user, expansive models that rely on actual user preferences may be built. These methods are especially useful when considering new users arriving in a system. A central problem with CF methods is that the resulting model is not as individually tailored as one created through CB methods. CF models must be careful not to represent the *average* user, but a single individual.

Formally, when using a collaborative method, the utility $p(u, i)$ of item i for user u is extrapolated from the known $p(u_x, i)$ where u_x is a user similar to u (Adomavicius and Tuzhilin, 2005, p.4).

Because of *the new user problem* of content-based prediction and the *average user problem* of collaborative prediction, many systems use a hybrid approach (as introduced by Burke (2007)). By combining content-based and collaborative methods, systems that properly handle predictions for new users and avoid too much generalization in the models can be achieved. We will discuss hybrid aggregation systems later in this chapter.

(2) The method variable is another way to classify recommenders. Orthogonal to what data the method uses, this variable concerns *how* the data is used. First, we have the *model-based* approach, where the recommender system builds predictive models based on the known data. Unseen items can then be fed into this model to compute its estimated utility score (Adomavicius and Tuzhilin, 2005, p.5). For example, creating a Bayesian networks from past interaction is a model-based approach.

The other category is the *heuristic* or *memory-based* approach (Adomavicius and

Tuzhilin, 2005, p.5). These methods use the raw data of items, users and ratings to directly estimate unknown utility values. For example, recommending items similar to the ones already rated by computing the cosine similarity of their feature vectors is a heuristic approach.

(3) The granularity variable tells whether this approach creates models for the canonical user, stereotypical users or individual users. The canonical user is another term for the *average* user, indicative of systems that adapt by seeing all users as a single entity. Stereotypical systems look at groups of users. For example, Rich (1979) presented one of the first user modeling systems based on stereotypes, used to predict which books in a library the users would most enjoy. Here, a dialogue between the system and the user was performed to place the user into a set of stereotypes. Each stereotype has a set of *facets* which is then used to match books and users. This created user models of *typical* granularity, as opposed to common *individual* approaches.

(4) Temporality refers to how volatile the gathered knowledge will be. While most RSs produce long term, relatively stable knowledge based on lasting user preference and taste, some systems use fluctuating parameters such as the time of day, exact location and the current context to produce recommendations. For example, Horvitz et al. (2003) used clues from a user's calendar, camera and other sensors to determine the attentional state of the user before delivering personalized and contextual notifications.

(5) The agents variable signifies whether the knowledge gathering and presentation is implicit and opaque, or explicit and requires dedicated user interaction. Explicit feedback through ratings is common in movie, product or music rating services (e.g. Bell et al. (2007b); Basu et al. (1998); Hotho et al. (2006)). However, for other services such as personalized search, implicit mining of query logs and user interaction is often used to build predictive models (e.g. Shen et al. (2005); Agichtein et al. (2006); Speretta and Gauch (2000); Teevan et al. (2005)).

2.3.3 Examples of Recommender Systems

As our solution will combine different recommender systems, we need a short introduction to some of the approaches we will use. This section takes a closer look at (1) *baseline ratings*, (2) *neighborhood estimation*, (3) *dimensionality reduction*, and (4) *network traversal*. This is by no means an exhaustive list, but rather a quick rundown of common approaches in recommender systems that we will use in the Chapter 3. See Adomavicius and Tuzhilin (2005), Pazzani and Billsus (2007), Schafer et al. (2007) or Bjørkøy (2010) for a more comprehensive exploration of different types of recommenders. Segaran (2007) gives a good introduction to how RSs are used in practice.

(1) Baseline ratings are the simplest family of recommender systems. These methods compute predictions through varying types of averages of known data. The data is

content-based, and used to compute heuristic predictions. While simple in nature, these methods are often helpful starting points for more complex systems, or as benchmarks for exploring new approaches. For example, Koren (2008, p.2) computes the baselines for items and users, and use more involved methods to move this starting point in some direction. The baseline (predicted relevance) for a user/item pair is given by

$$b_{ui} = \mu + b_u + b_i.$$

Here, μ is the average rating across all items and users, b_u is the user baseline and b_i is the item baseline. The user and item baselines correspond to how the user's and item's ratings deviate from the norm. This makes sense as some items may be consistently rated higher than the average, some users may be highly critical in their assessments, and so on. Koren computes these baselines by solving the least squares problem

$$\min_{b*} = \sum_{(u,i)\in R} (r_{ui} - \mu - b_u - b_i)^2 + \lambda(\sum_u b_u^2 + \sum_i b_i^2).$$

This equation finds baselines that fit the given ratings while trying to reduce overfitting by punishing greater values, as weighted by the λ parameter. By using baselines instead of simple averages, more complex predictors gain a better starting point, or in other words, a better average rating.

Another approach based on simple averages is the *Slope One* family of recommender algorithms. As introduced by Lemire and Maclachlan (2005), these algorithms predict unknown ratings based on the average difference in ratings between two items. For example, if item i is on average rated δ points above item j, and user u has rated item j, that is, we predict $\hat{r}_{u,i}$ (the estimated rating) to be $r_{u,j} + \delta$, for all the user's ratings that match this pattern,

$$\hat{r}_{u,i} = \frac{\sum_{j\in R_u} \text{ratings}(j) \times (r_{u,j} + \delta(i,j))}{\sum_{j\in R_u} \text{ratings}(j)}.$$

Here, $\hat{r}_{ui}$ is the estimated rating, R_u is the items rated by user u, ratings(i) is the number of ratings for item i, and $\delta(i,j)$ is the average difference in ratings for items i and j. While simplistic, Slope One is computationally effective and produces results comparable to more complex methods (Lemire and Maclachlan, 2005, p.5).

(2) Neighborhood estimation is at the core of many recommendation systems. This is the basic principle behind most collaborative filtering algorithms. Unknown ratings are estimated by averaging the ratings of similar items or users, weighted by their similarity. Neighborhood-based approaches often work in two steps. First, a neighborhood of similar

elements is computed. Second, the similarities and connections within this neighborhood is used to produce a predicted relevance score.

A common method for computing user similarity is the *Pearson Correlation Coefficient* (PCC) (Segaran, 2007, p.11). While simple, the PCC compares favorably to more complex approaches, and is often used as a benchmark for testing new ideas (e.g. in Lemire and Maclachlan (2005); Ujjin and Bentley (2002); Konstas et al. (2009)).

The PCC is a statistical measure of the correlation between two variables. In our domain, the variables are two users, and their measurements are the ratings of co-rated items. The coefficient produces a value in the range $[-1, 1]$ where 1 signifies perfect correlation (equal ratings), 0 for no correlation and -1 for a negative correlation. The negative correlation can signify two users that have diametrically opposing tastes. We compute PCC by dividing the covariances of the user ratings with their standard deviations:

$$\mathrm{pcc}(u, v) = \frac{cov(R_u, R_v)}{\sigma_{R_u}\sigma_{R_v}}.$$

When expanding the terms for covariance and standard deviations, and using a limited neighborhood size n, we get

$$\mathrm{pcc}_n(u, v) = \frac{\sum_{i\in K}^{n}(R_{ui} - \bar{R_u})(R_{vi} - \bar{R_v})}{\sqrt{\sum_{i\in K}^{n}(R_{ui} - \bar{R_u})^2}\sqrt{\sum_{i\in K}^{n}(R_{vi} - \bar{R_v})^2}}.$$

The limited neighborhood size becomes the statistical sampling size, and is a useful way of placing an upper bound on the complexity of computing a neighborhood. n does not have to be a stochastic sampling — it can also be limited by the number of ratings the two compared users have in common, the number of ratings each user have, or something similar, as denoted by K in the formula.

After a neighborhood is determined, it is time to predict the unknown rating. For Collaborative filtering approaches, we are interested in the similarity of users, which means averaging the user neighborhood ratings weighted by similarity (Segaran, 2007, p.16):

$$\bar{r}_{ui} = \frac{\sum_{v\in K(u,i)} \mathrm{sim}(u, v) \times R_{vi}}{\sum_{v\in K(u,i)} \mathrm{sim}(u, v)}.$$

Here, $\mathrm{sim}(u, v)$ is the similarity between two users, $K(u, i)$ is the set of users in the neighborhood of u that have rated item i. This is one of the simplest ways of computing a neighborhood-based prediction. Most systems use more complex estimations. For instance, Koren (2008) uses the baseline ratings discussed above instead of plain user and item ratings, to remove what they call global effects where some users are generous or strict in their explicit preferences, and some items are consistently rated differently than the average.

Content-based recommenders compute neighborhoods of items instead of users. The simplest approach is to find items highly rated by the current user, compute the neighborhood by finding items similar to these, and produce ratings by weighting the initial rating with the similarity of the neighboring items.

The PCC is but one of many methods used to compute neighborhoods. Other simple measures include the *euclidean distance* (Segaran, 2007, p.10), Spearman's or Kendall Tau rank correlation coefficients (Herlocker et al., 2004, p.30) — variations on the PCC. Of course, user similarity does not have to rely on ratings. If the system has access to detailed user profiles, these can be used to estimate the similarity of users. Similarity metrics from the field of *information retrieval* (IR), such as the *cosine correlation* of rating vectors, or content-based similarity metrics are applicable, as we shall see in Section 2.4.

Bell and Koren (2007b) shows a more sophisticated neighborhood estimation which computes global interpolation weights, that can be computed simultaneously for all nearest neighbors. Combinations of different types of neighborhoods are also possible. Ujjin and Bentley (2002) use a simple euclidean metric to gather a larger neighborhood, which is then refined using a *genetic algorithm*. Another common way of computing neighborhoods is by reducing the dimensions of the ratings matrix, as we will now describe.

(3) Dimensionality reduction is an oft-used technique when creating recommender systems. The ratings matrix is factored into a set of lower dimension matrices, that can be used to approximate the original matrix. By reducing the noise in the ratings matrix, and keeping those ratings that contribute to global patterns, we can identify groups of users, items or combinations that have something in common. This can then be used to find neighborhoods, compute similarities and estimate unknown ratings.

Singular Value Decomposition (SVD) is a common method for such matrix factorization (e.g. Billsus and Pazzani (1998, p.5), Sun et al. (2005), Bell et al. (2007b)). This is the same underlying technique used by *latent semantic indexing* in information retrieval (Baeza-Yates and Ribeiro-Neto, 1999, p.44). Formally, SVD is the factorization $R = U\Sigma V^T$. R is an $m \times n$ matrix, in our case the ratings matrix, with m users and n items. U is an $m \times m$ factor, V^T (the transpose of V) is an $n \times n$ factor. Σ is a $m \times n$ diagonal matrix. Σ is a diagonal matrix, made up of what is called the *singular values* of the original matrix.

The dimensionality reduction can be performed by truncating the factor matrices each to a number of rows or columns, where the number is a parameter depending on the current domain and data, called the *rank* (r). The truncated matrix is $R_r = U_r \Sigma_r V_r^*$, where U_r are the first r columns of U, V_r^T are the first r rows of V^T and Σ_r is the top-left $r \times r$ sub-matrix of Σ. There are many more complex ways of compressing the matrix than pure truncation, but this is a common way of reducing the factors. By truncating

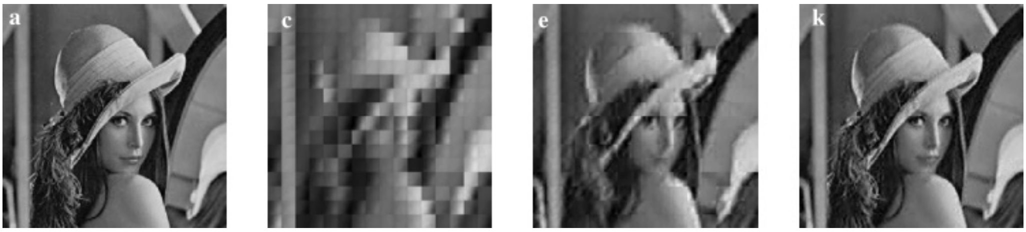


Figure 2.4: SVD-Based Image Compression: a variation of using SVD to compress an image, from Ranade et al. (2007). The original image is on the left, and successive images use an increasing number of factors (2, 8 and 30) when performing compression. Figure adapted from Ranade et al. (2007, p.4).

the factors, we in essence create a higher-level approximation of the original matrix that can identify latent features in the data. With the factors reduced to r dimensions, the resulting matrices are compressed:

$$\begin{bmatrix} \\ \quad R_{m,n} \quad \\ \\ \end{bmatrix} \Rightarrow \begin{bmatrix} \\ U_{m,r} \\ \\ \end{bmatrix} \begin{bmatrix} \Sigma_{r,r} \end{bmatrix} \begin{bmatrix} \quad V^*_{r,n} \quad \end{bmatrix}$$

Two important transformations happen in this dimensionality reduction. First, ratings that do not contribute to any greater pattern are removed as noise. Second, ratings that in some way correlate to each other are enhanced, giving more weight to the actual predictive parts of the data. This means that the reduced factors can for instance identify features that correspond to correlations between items or users. These features are comparable to the mapping of terms to concepts in LSI.

Because SVD can find the most descriptive parts of a matrix, this technique is often used for image compression. The image we wish to compress is treated as an $N \times M$ matrix, which is run through an SVD factorization. The factors are truncated, and the result expanded to a matrix that is much simpler to represent than our original image matrix. As seen in Figure 2.4, how close the compressed image resembles the original image depends on the chosen *rank*, i.e. how many rows and columns we keep during truncation. A higher rank means less dimensionality reduction and less compression of the image.

The key question for any SVD algorithm is how it performs the factorization, and which rank the original matrix is reduced to. Two common factorization methods are the EM and the ALSWR algorithms. An EM factorizer uses the Expectation-Maximization algorithm to find the factors. An ALSWR factorizer performs the same factorization with a least-squares approach (Zhou et al., 2008). The number of features refers to the truncation of the factors in order to reduce the concept-space. These features then correspond to the number of latent taste categories we wish to identify. Naturally, different numbers of features will yield different recommenders. The SVD factorization algorithms are iterative methods, where each iteration yields more accurate results.

In recommender systems, SVD is used to compress the ratings space into what is sometimes called a *taste space*, where users are mapped to higher-level *taste* categories (e.g. Ahn and Hong (2004, p.5), Brand (2003, p.4) or Liu et al. (2006, p.2)). In a taste space, the collections of individual ratings are reduced to groups of users, items and combinations that have patterns in common. This reduction makes it easy to find similar users that share some global characteristic. We can also find similarities between items, clusters of items and user and so on, all based on latent categories discovered by the automatic identification of patterns in the data. SVD is then an ingenious way of dealing with the commonly sparse ratings data, by identifying latent correlations and patterns in the data, which is exactly what we need to predict unknown ratings or connections.

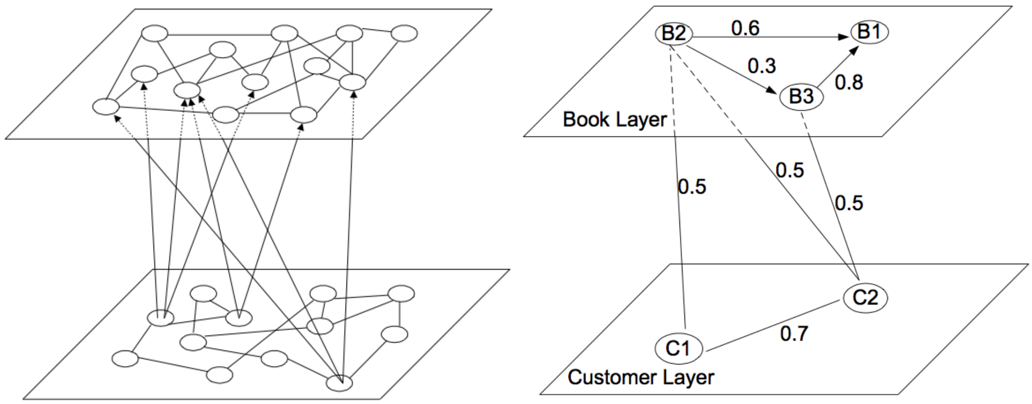
(4) Network traversal recommenders refers to estimating predictions by traversing a graph of users and items to provide recommendations. The connections between nodes can be any type of relation that makes sense to the RS. Examples include linking item and user nodes based on purchases or explicit ratings, linking user nodes from asymmetrical (directed edges) symmetrical (undirected edges) relations, or linking items based on some similarity metric. Recommender systems can use common graph-traversal algorithms to infer unknown connections in this graph.

Huang et al. (2002) used network traversal to create a simple system for recommending books to customers. Here, edges between items and users correspond to ratings, and edges connecting nodes of the same type are created by connecting elements that have similarity above a certain threshold. Predictions are generated by traversing the graph a preset number of steps starting at the current user, and multiplying the weights of paths leading to the target item (see Figure 2.5).

The complexity of recommender systems based on networks are only limited by the kinds of relations we can produce. For example, recommending other users in social networks can easily utilize friend or friend-of-a-friend relations to find others the current user might be interested in connecting to. Indeed, any relevant similarity metric can be used to connect nodes of the same type, or nodes of different types.

One variation comes from Walter et al. (2008), who create a network of *transitive trust* to produce recommendations. Here, the neighborhood of users is determined by traversing through users connected by a level of trust. The trust can for example be a function of how many agreeable results the connection to a user has produced. Users trust each other's recommendations based on previous experience.

Konstas et al. (2009) takes yet another approach that measures the similarity between two nodes through their *random walks with restarts* (RWR) technique. Starting



(a) *(b)*

Figure 2.5: Network Traversal Example: (a) A graph with two kinds of nodes, e.g. items and users. (b) A graph with books and customers, where recommendations can be made by traversing the weighted connections. Connections between nodes of the same type represent similarity, while connections between books and customers represent purchases. Figures from Huang et al. (2002).

from a node x, the RWR algorithm randomly follows a link to a neighboring node. In every step, there is a probability α that the algorithm will restart its random walk from the same node, x. A user-specific column vector $P^{(t)}$ stores the long term probability rates of each node, where $P_i^{(t)}$ represents the probability that the random walk at step t is at node i. S is the column-normalized adjacency-matrix of the graph, i.e. the transition probability table. Q is a column vector of zeroes with a value of 1 at the starting node (that is, Q_i is 1 when the RWR algorithm starts at node x). The stationary probabilities of each node, signifying their long term visiting rate, is then given by

$$P^{(t+1)} = (1-\alpha)SP^{(t)} + \alpha Q.$$

This algorithm is then run to convergence (within a small delta). Then, the *relatedness* of nodes x and y is given by P_y where p is the user model for the user represented by node x. Konstas et al. found that this approach outperformed the PCC, as long as the social networks were an explicit part of the system in question. The connections between users had to be one actively created by the users to be of such quality and precision that accurate predictions could be made.

After this whirlwind tour of recommender systems, it is time to look at some closely related topics: information retrieval and personalized search. This will form the basis for the case study performed in Chapter 4.

2.4 Personalized Search

Personalized search means adapting the results of a search engine to each individual user. As we shall see, this field has a lot in common with recommender systems. In both situations, we wish to predict how relevant each item will be to each user. Before delving into the techniques of personalizing search results, we will quickly present the basics of *information retrieval* (IR).

2.4.1 Information Retrieval

Manning et al. (2008, p.1) define IR as *"finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)"*.

How does this relate to recommender systems? There is an important distinction. The purpose of *recommendations* is twofold. (1) To show the user items similar to another item, and (2) to allow discovery of relevant items the user did not know exist. The purpose of *search* is a bit different. To allow the user to find the location of information he or she knows (or hopes) exists. The defining separator is often the knowledge of existence. However, as we shall see, the two fields employ similar methods and terminology. In the next chapter, we will show how these can work together, by allowing an IR method to constrain the item-space worked on by our recommender system.

Baeza-Yates and Ribeiro-Neto (1999, p.23) presents a formal definition of an IR system: IR $= (Documents, Queries, Framework, ranking(q_i, d_i))$.

As evident by the scope of IR literature, these elements can be just about anything that has to do with retrieving information. However, in what is often called *classic IR*, the documents contain free text with little internal structure, and the queries are short user-initiated descriptions of an *information need* (Baeza-Yates and Ribeiro-Neto, 1999, p.19). For example, this model describes Web search engines, where the documents are web pages and queries are short sentences or keywords input by users.

The *Framework* in our quadruple refers to how documents are stored and retrieved. Basic approaches to IR split each document into a set of terms (e.g. words), and create an *inverted index* (Manning et al., 2008, p.22), that lists documents by terms. There are numerous extensions to this framework, including:

- Positional information for phrase search (Manning et al., 2008, p.39)
- Stop word removal (removing the most common terms) (Manning et al., 2008, p.27)
- Stemming (reducing words to their root forms) (Manning et al., 2008, p.32)
- Lemmatisation (contextual inflection removal) (Manning et al., 2008, p.32)
- Query reformulation (Baeza-Yates and Ribeiro-Neto, 1999, p.117)

All these techniques help improve (among other things) the *recall* and *precision* of the retrieval engine. Recall, precision and relevance are well defined measures for evaluating the quality of a search engine (Manning et al., 2008, p.5):

- A document is *relevant* if it satisfies the user's information need.
- *Recall* is the fraction of relevant documents retrieved by the system.
- *Precision* if the fraction of retrieved documents that are relevant.

There are many more measures, but recall and precision succinctly define what a search engine must to to be successful — retrieve many relevant documents and few irrelevant documents. Failing this test is to neglect the main purpose of IR: to prevent information overload by allowing efficient access to relevant parts of an otherwise overwhelming collection of information.

In relation to this thesis, the most interesting part of any IR system is its *ranking function*. This function computes the score of each document relative to the current query. The relation to recommender systems is quite clear. Both the ranking function and RSs compute the relevance of items in the current context, either based on a query or the current user. Indeed, IR systems use many of the same metrics to measure the similarity of queries and documents, as RSs measure the similarity of items.

A common framework for storing and ranking documents is the vector space model (VSM). This model stores documents as term vectors. Each term represents a dimension, and documents are vectors in this term-space. When performing a query, the query terms are also represented as a vector in the same space. By computing the cosine similarity between the query and each document, we get a good estimate of how well a document matches a query (Baeza-Yates and Ribeiro-Neto, 1999, p.29).

The next question is what to store at each $(document, term)$ coordinate in the vector space (called the document/term weights). Storing simple 1 or 0 values representing whether or not terms are present gives a model where a document's relevance is proportional to how many query terms it includes. However, this is not very precise. For example, by this definition, a document containing every conceivable query term would be the most relevant to any query. A better idea is to use something like the TF-IDF weighting scheme (Baeza-Yates and Ribeiro-Neto, 1999, p.29):

$$w_{t,d} = tf_{t,d} \times idf_t = \frac{f(t,d)}{\sum_{k \in d} f(k,d)} \times \log \frac{N}{n_t}.$$

The final weight is computed by multiplying the term frequency score (TF) $tf_{t,d}$ with the inverse document frequency (IDF) idf_t. TF evaluates how well the term describes the document contents, while IDF punish terms that appear in many documents. $f(t,d)$ gives the frequency of a term in a document. N is the total number of documents, and n_t the number of documents in which t appears. The effect of the IDF factor is dampened by taking its log-value. Together, TF and IDF ranks documents higher by words that discriminate well within the document corpus, and ignores words that appear in so many documents that they have little to no predictive capacity.

While simple, TF-IDF has proven itself resilient when compared to more complex methods, and many more complex methods have been built on its foundations (e.g. BM25, one of the most successful probabilistic weighting algorithms (Robertson, 2010)).

There are as many IR models as there are domains that need search. Even the basic VSM can be constructed in a myriad of ways. Other models include the simple *boolean search model*, where queries are based on boolean algebra. Probabilistic models frame the similarity question as the probability that the document is relevant. Latent Semantic Indexing (LSI) is the application of SVD to IR by performing dimensionality reduction of the term-space into concept-space. See Manning et al. (2008), Robertson (2010) or Baeza-Yates and Ribeiro-Neto (1999) for a more comprehensive introduction to models in IR.

The important take-away is that, while serving different use cases, RSs and IR systems employ much of the same technology, only with different input and expected output.

2.4.2 Ranking Signals

Modern Web search engines have long ago moved on from simple ranking metrics such as TF-IDF. While traditional metrics may form the foundation of modern search engines, a lot more thought go into the final results. Different types of ranking functions are combined to produce the final *search engine results page* (SERP), with each ranking function often being referred to as a *signal*. Alternate names include *re-ranking* or *re-scoring* functions.

Google, the company behind the popular online search engine, writes that *"Today we use more than 200 signals, including PageRank, to order websites, and we update these algorithms on a weekly basis. For example, we offer personalized search results based on your web history and location."*[1] Bing, a Web search engine from Microsoft, uses the same terminology: *"We use over 1,000 different signals and features in our ranking algorithm."*[2]

Signals are often products of the document structure of the current domain. Sergey and Lawrence (1998, p.5) points to the use of the proximity of query terms in matching documents. Those where the terms appear close together are natural candidates for a higher ranking. Other signals, still reliant on the documents themselves, are more domain oriented. One signal they point out is how words in a larger or bold font can be weighted higher than normally typeset words. In this way, the design of a document is used to choose the most important terms.

(1) google.com/corporate/tech.html — accessed 11.04.2011
(2) bing.com/community/site_blogs/b/search/archive/2011/02/01/thoughts-on-search-quality.aspx — accessed 11.04.2011

Signals can also depend on the query. Manning et al. (2008, p.145) describes a system that takes multi-word queries, breaks them up into different permutations and runs the new queries against the same document index and ranking function. Each query corresponds to its own ranked set of results, which are sent to a *rank aggregation function* which turns the accumulated ranking evidence into one coherent result. We will have more to say on rank aggregation in Section 2.5.

Signals can also be external to the collection or relational within the collection. PageRank (Sergey and Lawrence, 1998, p.4) is perhaps the best known of the relational signals. The algorithm forms a probability distribution over web pages, ranking their perceived authority or importance according to a simple iterative estimation. Every web site is given its rank based on how many pages that link to it. For each page that provides links, the score it contributes to the linked-to page is its own page rank, inversely proportional to the number of outbound links the page has. Another intuitive justification for a site's PageRank is the *random surfer model* (Sergey and Lawrence, 1998, p.4). The probability that the random surfer visits a page is its PageRank. The *randomness* is introduced by a damping parameter d, which is the probability that a user will stop browsing and start at a new random page:

$$\text{PageRank}(x) = \frac{1-d}{N} + d \sum_{y \in B_x} \frac{\text{PageRank}(y)}{\text{Links}(y)}.$$

B_x is the set of pages linking to page x, and $\text{Links}(y)$ is the number of outbound links from page y. The first term distributes an equal PageRank score to all pages that have no outbound links, as N is the total number of pages. This iterative algorithm is run until convergence inside a small delta.

Let us now finally take a look *personalized search*, a field where such signals may play an important part.

2.4.3 Personalizing Search Results

Search engines, especially online search engines, face a huge challenge. In addition to the wide range of websites, the ambiguity of language and the restricted nature of queries comes the wildly differing users. Each user is unique. Even when considering one user, there might be many different use cases, for example when using the same search engine at work and at home. Another identified problem is that users use search engines for navigation as well as pure search. Teevan et al. (2007) found that as many as 40% of all queries to the Yahoo! search engine were "re-finding queries", i.e. attempts to find information the user had accessed before.

Personalized search attempts to solve these problems by introducing individually adaptive search results. These techniques are based on user modeling (as introduced in Section 2.2), and attempts to build predictive models based on mined user preferences.

Commonly, this is done through query log analysis (e.g. Liu et al. (2002); Sugiyama et al. (2004); Shen et al. (2005); Speretta and Gauch (2000)). These are often model-based techniques with implicit knowledge gathering agents, that create individual, long-term user models (these terms are described in Section 2.3).

There are two leading approaches to personalizing search results (Noll and Meinel, 2007, p.2). The first method is query reformulation, where the actual user query is enhanced in some way, before traditional IR retrieves and ranks documents. The second method is results re-ranking, where the IR results are sorted based on personalized metrics. This section describes the latter approach.

To demonstrate how these methods compare to traditional recommendation systems, we will explore a few different approaches to personalized search. (1) *Personalized topic-sensitive PageRank*, (2) *folksonomy-based personalization* and (3) *social network search ranking*.

(1) Personalized topic-sensitive PageRank Haveliwala (2003) introduced a topic-sensitive PageRank algorithm, that they found to *"generate more accurate rankings than with a single, generic PageRank vector"*. They show how to create topic-specific PageRank vectors for a number of pre-set topics, creating many rankings per page, one per topic. This new PageRank is computed based on an existing set of websites that belong to one or more topics. Qiu and Cho (2006) achieved "significant improvements" to this approach by adding a personally adaptive layer to the topic-sensitive PageRank algorithm, creating a *personalized PageRank algorithm*.

In addition to the topic vector, Qiu and Cho creates a topic-preference vector for each user. When the user has clicked on a few search results, a learning algorithm kicks in and estimates approximately how likely the user is to be interested in the pre-set topics, creating the topic-preference vector T. When it is time to rank a page p in response to the query q, they compute the personalized ranking:

$$PersonalizedRanking(T, p, q) = \sum_{t=1}^{m} T(i) \times Pr(q|T(i)) \times TSPR_i(p)$$

We will not deduce this equation here (see Qiu and Cho (2006, p.5)), but let us explain it. T is the user-specific topic preference vector. i is the index of a topic and m the total number of topics. $Pr(q|T(i))$ is the probability that the query belongs in topic i. This can be as simple as the total number of times the query terms appear in websites under topic i. $TSPR_i(p)$ is the topic-sensitive PageRank score for page p in topic i. Basically, this is the normal PageRank vector computed within a pre-set topic i.

The construction of $T(i)$, i.e. the training phase of the algorithm, is performed by mining the query logs from individual users. By identifying how many sites the user has visited in per topic, computing T can be done through linear regression or by using a

Maximum-likelihood estimator. Qiu and Cho (2006, p.10) reports improvements of 25% to 33% over the Topic-sensitive PageRank approach, which Haveliwala (2003) reports outperformed the original PageRank algorithm.

(2) Folksonomy-based personalization Web applications often have more information about users and items (documents, sites or articles) than simple ratings. One of these extra resources are tags, simple keywords assigned from users to items. The collection of users, items, tags and user-based assignment of tags to resources is called a *folksonomy*.

Hotho et al. (2006) defines a folksonomy as a tuple $F = (U, T, R, Y, \prec)$. Here, U, T and R are finite sets of users, tags and resources (items), respectively. Y is a ternary relation between users, tags and resources, called tag assignments. $\prec$ is a user-specific tag hierarchy, applicable if the tags are organized as super- and sub-tags. The *personomy* P_u is a user-specific part of F, i.e. the tags, items and assignments related to one user u. In our terms, this personomy would be the user model. Hotho et al. use folksonomies to do information retrieval based on their *FolkRank* search algorithm, a derivative of PageRank.

Bao et al. (2007) shows how folksonomies can be used to personalize search. They first create a topic-space, where every user and document are represented. Each tag in the system is a dimension in this topic-space, or tag-space. Whenever a new query is issued, two things happen. First, a regular IR method computed a standard, non-personalized ranking of documents. Second, a personalized ranking list is computed by performing a simple vector-space model matching in the topic-space, for example by using cosine similarity (as previously explained). The personalized list is then unrelated to the actual query, and is simply a ranking of the most relevant pages to the current user.

The two ranks are aggregated by a simple consensus-metric, the *Weighted Borda-Fuse* aggregation method (Xu et al., 2008, p.3), which is another name for weighted combination of the rankings:

$$\text{rank}(u, q, p) = \alpha \times \text{rank}_{IR}(q, p) + (1 - \alpha) \times \text{rank}_{RS}(u, p)$$

Xu et al. tried many combinations of weights, topic selection and datasets, with the general conclusion that folksonomy-based personalized search has great potential. If nothing else, this example shows how easily tags can be integrated to provide an individual searching experience.

(3) Social network search ranking Carmel et al. (2009) developed a personalized search algorithm based on a user's *social network*. By re-ranking documents according to their relation to with individuals in the current user's social network, they arrived at a document ranking that "significantly outperformed" non-personalized social search (Carmel et al., 2009, p.1). Note the qualifier "social search". Their project searches through social data within an enterprise, naturally conducive to algorithmic alterations based on social

concepts. However, as social data is data just as well, seeing how a personalized approach improves standard IR in this domain, is helpful.

Their approach: first, documents are retrieved by a standard IR method. Second, the user's socially connected peers are also retrieved. Third, the initial ranked list of documents is re-ranked based on how strongly they are connected to the user's peers, and how strongly those peers are connected to the user. The user-user similarity is computed based on a few signals (Carmel et al., 2009, p.2), e.g. co-authoring of documents, the use of similar tags (see Example 2 in this section), or leaving comments on the same content. The user model also includes a list of terms the current user has employed in a social context (profile, tags, et cetera). This is all done to infer implicit similarity based on social connections.

The algorithm is quite powerful, and combines three types of rankings. The initial IR score, the social connection score, and a term score, where the terms are tags and keywords used by a user. The user model is $U(u) = (N(u), T(u))$, where $N(u)$ are the social connections of u and $T(u)$ the user's profile terms. The function $sim(x, y)$ measures the similarity of two elements, either users or items.

The re-scoring is performed in two steps. First, the score based on connections and terms is computed, weighted by β which determines the weighting of both approaches:

$$S_P(q, d, U(u)) = \beta \sum_{v \in N(u)} sim(u, v) \times sim(v, d) + (1 - \beta) \sum_{t \in T(u)} sim(u, t) \times sim(t, d)$$

Finally, the results are combined with the ranking returned by the IR method (R_{IR}). A parameter α is used to control how much each method is weighted:

$$S(q, d, U(u)) = \alpha \times R_{IR}(q, d) + (1 - \alpha) \times S_P(q, d, U(u))$$

This approach, while simple, shows how social relations and social annotations can easily be used to personalize a search experience. However, Carmel et al. (2009, p.10) notes that the high quality data in their enterprise setting were important to achieve the improved results.

2.5 Recommender Aggregation

So far we have seen a lot of modeling methods, both for recommender systems (RS) and for personalized search (PS). *Aggregate modeling* is the act of merging two or more modeling methods in some way. A proper aggregation method creates a combined result that is better than either of the individual methods, where the sum is greater than the parts. We have already seen a few examples of aggregate modeling:

- Koren (2008) aggregates global, individual and per-item averages to a baseline.
- Huang et al. (2002) aggregates different types of graph relations into one prediction.
- Haveliwala (2003) combined their personalized PageRank with another approach.
- Carmel et al. (2009) combined classic IR with social relations and annotations.
- Sergey and Lawrence (1998, p.5) aggregates signals measured from website structure.

The reason for combining different approaches is that no one method can capture all the predictive nature of available data. For example, Bell et al. (2007a) created a recommender system where the neighborhood- and SVD-based approaches complement each other. While the neighborhoods correspond to *local effects* where similar users influence each other's predictions, the dimensionality reduction finds *regional effects*, i.e. major structural patterns in the data (Bell et al., 2007a):

> *"Both the local and the regional approaches, and in particular their combination through a unifying model, compare favorably with other approaches and deliver substantially better results than the commercial Netflix Cinematch recommender system [...]."*

An interesting question is whether or not all hybrid recommenders, that combine content-based and collaborative methods, are aggregators. This is mostly a question of semantics and implementation. Burke (2007, p.4) liberally defines a hybrid system as *"any recommender system that combines multiple recommendation techniques together to produce its output"*. Some hybrid methods combine stand-alone methods, and are definitely aggregations. Other methods merge the methods themselves into one implementation that uses the data in different ways. Burke describes a few types of hybrid recommenders:

- Weighted combinations of recommenders.
- Switching and choosing one recommender in different contexts.
- Mixing the outputs and presenting the result to each user.
- Cascading, or prioritized recommenders applied in succession.
- Augmentation, where one recommender produces input to the next.

However, without being to pedantic, these can all be seen as aggregations. In each case, there are two main use cases for recommender aggregation Liu et al. (2007):

(1) Rank (or *order-based*) aggregation (RA) lets a set of methods produce a sorted list of recommendations or search results. These lists are then combined into one final list, through some aggregation method (see Dwork et al. (2001) or Klementiev et al. (2008)). These methods only require the resulting list of items from each method Aslam and Montague (2001, p.2).

(2) Prediction (or *score-based*) aggregation (PA) works on the item- or user-level by combining predicted scores one-by-one, creating an aggregated result for each element that should be evaluated. These methods require the actual prediction scores for any item from the recommender methods (Aslam and Montague, 2001, p.2).

2.5.1 Rank Aggregation

RA combines multiple result lists into one list through aggregation. Dwork et al. (2001) shows a few metrics applicable to meta-search, the combination of results from multiple search engines. For example, Borda's method (Dwork et al., 2001, p.6) is based on positional voting, where each result gets a certain number of points from each result set, based on where it appears in the sorted list. Items at the top gets the most points, while lower items gets fewer points. This is in essence an approach where the predictors have a set number of votes (c, the number of results) that they give to the items.

As we saw in Section 2.4, Xu et al. (2008, p.3) used a weighted version of this approach to combine an IR and personal approach to result ranking. Aslam and Montague (2001, p.3) calls their version of this *Weighted Borda-Fuse*, where the points given from a method to an item is controlled by the weights estimated for the methods. Aslam and Montague (2001, p.4) also explain a Bayesian approach (*bayes-fuse*), that combined with the *naive Bayes* independence assumption produce the following formula:

$$\text{relevance}(d) = \sum_{i \in Methods} \log \frac{\Pr(r_i(d)|rel)}{\Pr(r_i(d)|irr)}.$$

Here, $\Pr(r_i(d)|rel)$ is the probability that document d is relevant given its ranking by method i. Conversely, $\Pr(r_i(d)|irr)$ is the probability that the document is irrelevant. The probability values are obtained through training, and evaluating the results against known relevance judgements. An interesting note is that the standard Borda method does not require training data, while the weighted version and the Bayesian approach do. Aslam and Montague (2001) achieved positive results with these methods:

> *"Our experimental results show that meta-search algorithms based on the Borda and Bayesian models usually outperform the best input system and are competitive with, and often outperform, existing meta-search strategies."*

Liu et al. (2007) presents a rank-aggregation framework, where the task of estimating a ranking function by using training data. They treat this task as a general optimization problem, with results showing that this framework can outperform existing methods (Liu et al., 2007, p.7).

Rank aggregation is a substantial topic, with many approaches. The main take-away is that this approach combines list of results into one single results, and experiments show that results superior to the best of the combined methods are attainable. See Aslam and Montague (2001), Liu et al. (2007) or Klementiev et al. (2008) for more information.

2.5.2 Prediction Aggregation

Unlike rank aggregation, prediction aggregation (PA) does not deal with lists of results. PA works on the item-level, collecting scalar predictions of an item's relevance from a number of methods, and combining these predictions into a final score. This thesis is most concerned with prediction aggregation, and it is where we shall evaluate the performance of our approach.

Aslam and Montague (2001) describe a number of simple approaches. For example, minimum, maximum and sum aggregations combine the individual predictions based on simple arithmetics, or select one or more of the results as the final prediction. Other models use the average, or log-average of the different methods. Another example is the linear combination model, that trains weights for each predictor and weighs predictions accordingly. At slightly more complex approach is to train a logistic regression model (Aslam and Montague, 2001, p.3) over a training set, in an effort to find the combination that gives the lowest possible error. This last method improved on the top-scoring predictor by almost 11%, showing that even fairly simple combinations have merit.

Early approaches to recommender systems experimented with aggregating content-based and collaborative approaches. Claypool et al. (1999) combined the two approaches in an effort to thwart problems with each method. CF methods have problems rating items for new users, radically different users or when dealing with very sparse data. CB methods do not have the same problems, but are less effective than CF in the long run, as CB does not tap into the knowledge of other users in the system — knowledge that out-performs simple content analysis. In Claypool et al. (1999), the two types of recommenders were used to create a simple weighted result.

Generally, methods for aggregating predictions in the field of machine learning is called *ensemble methods* Dietterich (2000). While most often used to combine classifiers that classify items with discrete labels, these methods are also used for aggregating numerical values (see the numerical parts of Breiman (1996)). Approaches include *bootstrap aggregation* (bagging) and *boosting* for selecting proper training and testing sets, and creating a *mixture of experts* for combining the predictors (Polikar, 2006, p.27).

Bell et al. took method aggregation to its logical conclusion when winning the Netflix Challenge, by combining 107 individual results from different recommenders. They found this to significantly outperform each standard recommender (Bell et al., 2007b, p.6):

> *"We strongly believe that the success of an ensemble approach depends on the ability of its various predictors to expose different, complementing aspects of the data. Experience shows that this is very different from optimizing the accuracy of each individual predictor. Quite frequently we have found that the more accurate predictors are less useful within the full blend."*

Both rank and prediction aggregation are extensive topics. In both cases, the take-away stays the same. By combining different modeling methods, more patterns in the data can be mined, and the resulting combination can outperform the best performing method. This is key to the model we shall build in the next chapter.

— 3 —

Methods & Implementation

In this chapter we will build our approach to relevance prediction, which we call *adaptive recommenders*. We will first explain why a new approach is needed, and develop three hypotheses based on our assumptions. The two final sections will show how adaptive recommenders can perform prediction aggregation in a recommendation scenario, and rank aggregation in a personalized search scenario.

3.1 Latent Subjectivity

As described in the previous chapter, there are many ways of predicting the relevance of an item to a user. In fact, judging by the number of different approaches, the only limiting factor seems to be the different patterns researchers discover in available data.

As described in Section 2.5, modern approaches to recommender systems try to combine many of these methods. By leveraging so called *disjoint patterns* in the data, several less than optimal predictors can be combined, so that the combination outperforms the individual algorithms. Moderns search engines work much in the same way, combining multiple ranking signals into a final results list (see Section 2.4.2).

Why then, when we have all these valid approaches, would we need yet another technique? As explained in Chapter 1, we believe the *latent subjectivity problem* is a fundamental issue with each approach described so far. Consider the following examples of relevance judgement:

- PageRank (Sergey and Lawrence, 1998) assumes that the relevance of a web page is represented by its authority, as computed from inbound links from other sites.
- Some systems considers a user's social connections to be important in ranking search results, when performing personalized search (e.g. Carmel et al. (2009)).
- When recommending movies, one predictor may be based on the ratings of users with similar profile details. Another predictor might be dependent on some feature, e.g. production year of well liked movies.
- SVD-based approaches assume global patterns and groups of users and items are best suited to produce predictions (see Section 2.3.3).
- Recommendations based on the Pearson Correlation Coefficient (Segaran, 2007, p.11) assumes that the statistical correlation between user ratings is a good measure for user similarity.

While these methods may perform well, their selection reflects how whoever created the system assumes how users can and *should* be modeled. This underlying subjectivity is not desirable. We see two different approaches one can take when creating a recommender system, represented by two questions:

1. What combination of which methods will accurately predict unknown scores?
2. Which methods could possibly help predict a score for a user or an item?

The first question is what has to be considered in traditional modeling aggregation. First, a set of applicable methods leveraging disjoint patterns must be selected. Then, an optimal and generalized combination of these must be found, most often through minimizing the average error across all users.

The second question is quite different. Instead of looking for an optimal set of methods and an optimal combination, we look for the set of *any applicable methods* that *some users* might find helpful, or that might work for *some items*. We believe this is a much simpler problem. Instead of trying to generalize individuality, it can be embraced, by allowing users to implicitly and automatically select which methods they prefer, from a large set of possible predictors.

As explained in Chapter 1, assuming that users will place similar importance on each modeling method is contrary to the goals of user modeling. Our goal should instead be a system which can leverage the differences between users and employ the available algorithms based on their individual preferences.

Just as users are different, items have their own characteristics. Needless to say, items are often quite different from another, along a myriad of dimensions. For example, if items correspond to websites, the number of metrics we can use to judge the relevance of an item is immense. If items are indeed as different as the users themselves, it stands to reason that the same modeling method will not perform as well for every item.

An approach where we only need to consider the second question is desirable. Regardless, both traditional, single-approach modeling methods, and modern aggregation approaches often treat every user and item the same. No matter its intrinsic qualities, an element will be judged by the same methods as every other element.

This chapter will develop a way to aggregate a host of modeling methods on a per-user and per-item basis. By adapting the aggregation to the current item and user, we should be able to mitigate the latent subjectivity problem. This adaptive aggregation is performed implicitly and automatically, without any extra interaction required from each user.

As mentioned in Chapter 1, this has an important consequence. If each algorithm is *only used* in situations it will probably perform well, any possible recommender algorithm becomes a worthy addition. In other words, instead of finding an optimal combination, we can use any algorithm that *might work for some elements* in our aggregation.

With adaptive recommenders, the users are in control of which methods best fit their needs, and a method's priority is influenced by how well it performs for the current item. However, in order to test whether or not these assumptions hold, we need a set of testable hypotheses.

3.2 Three Hypotheses

Our research goal is to develop the *adaptive recommenders* technique, and determine if this is a viable approach. To solve the subjectivity problem we need our modeling method to adaptively aggregate a set of predictions based on the current context. By automatically adapting how a set of disjoint recommenders are combined, based on individual users and items, we should be able to achieve a result that is better than any of the stand-alone recommenders. This adaptive method should also outperform other, generalized aggregation approaches.

In order to achieve our goal, this thesis will consider three hypotheses (H1-H3). H1 and H2 will consider the approach in regard to prediction aggregation in a recommendation scenario. H3 will consider using this approach for rank aggregation in an information retrieval scenario. Let us start with prediction aggregation:

> *H1: Adaptive recommender aggregation can achieve higher accuracy than each of the combined recommenders.*

It stands to reason that if a recommender system is indeed impaired by the subjective selection of modeling methods, an adaptive combination of these methods, based on individual users and items, should outperform the individual approaches. Second:

> *H2: Adaptive recommender aggregation can achieve higher accuracy than generalized aggregation methods.*

If our assumption that model aggregation inherits the subjective nature of its chosen parts, an adaptive aggregation without such misplaced subjectivity should outperform a generalized combination. Third:

> *H3: The ordering of results from an information retrieval query can be personalized by using adaptive recommenders.*

As described in Section 2.4.2, modern search engines combines multiple ranking functions called signals into a final results list. We shall use H3 to see whether or not adaptive recommenders can be used for this type of rank aggregation, where a set of recommender systems constitute a set of input signals. Unlike the experiments for the other hypotheses, the experiments for H3 will be a qualitative exploration of personalized search. A quantitative performance measure of personalized search is outside the scope of this thesis.

By answering these three hypotheses, it should become clear whether or not our approach is a viable technique for improved relevance predictions between users and items.

3.3 Adaptive Recommenders

Adaptive Recommenders (AR) is our technique for combining many recommender systems in a way that is optimal for each user and item. Given that we wish to predict the relevance of an item to a user, using many methods that consider disjoint data patterns, there are two important questions:

1. What rating does each method predict?
2. How accurate will each of these predictions be?

Recommender systems traditionally only care about the first question. A single method is used to predict an unknown rating. Modern aggregation techniques goes one step further, and combine many methods using a generic (often weighted) combination. We wish to make the aggregation *adaptive*, so that the aggregation itself depends on which user and which item we are considering.

Formally, we define adaptive recommenders as *adapting a set of recommender systems with another complementary set of recommender systems* (see Figure 3.2). This then is a form of meta-modeling, where one set of modeling methods is adapted by another set of modeling methods. The first set creates standard prediction scores, and answers the first question. The second set predicts how accurate each method will be for the current user and item, answering the second question. The interesting bit is that AR can use recommender systems for both these tasks, as we shall see.

A system for adaptive recommenders can be specified by the following 6-tuple:

$$\begin{aligned} \text{AR} &= (Items, Users, Ratings, Framework, Methods, Adapters) \\ &= (I, U, R, F, M, A). \end{aligned}$$

We have sets of $Users$ and $Items$, and a set of $Ratings$. Any user $u \in U$ can produce a rating $r \in R$ of an item $i \in I$. As mentioned, items can be just about anything, for example documents, websites, movies, events, or indeed, other users. The ratings can be explicitly provided by users, for example by rating movies, or they can be implicitly extracted from existing data, for example by mining query logs.

As before, we use the term *rating* loosely. Equivalent terms include *relevance*, *utility*, *score* or *connection strength*. This is still a measure of what a user thinks of an item in the current domain language. However, since *rating* will match the case study we present in the next chapter, we will be using this term.

The $Framework$ variable specifies how the data is represented. The two canonical ways of representing users, items and ratings are graphs and matrices (see Section 2.3). We shall use a matrix, where the first dimension corresponds to users, the second to items, and each populated cell is an explicit or implicit rating:

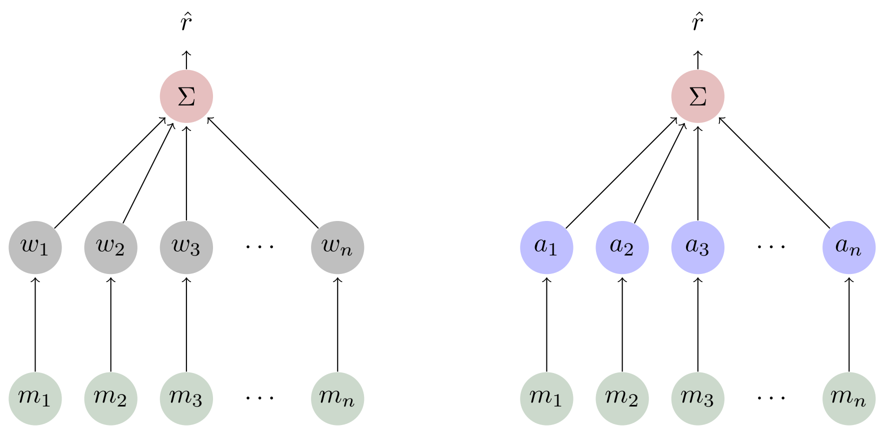


Figure 3.1: Comparison of generalized aggregation and adaptive recommenders. (left) Modern aggregation approaches uses a set of pre-trained weights to prioritize each modeling method. The weighted predictions are aggregated into a final prediction $\hat{r}$. (right) Adaptive user modeling employs secondary modeling methods instead of weights. These estimate the accuracy of the initial method for the current user and item.

$$R_{u,i} = \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,i} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ r_{u,1} & r_{u,2} & \cdots & r_{u,i} \end{pmatrix}.$$

As we wish to leverage disjoint data patterns, we have a set of modeling *Methods*, with their own ways of estimating unknown ratings. Each model $m \in M$ is used to compute independent and hopefully complimentary predictions. In our case, these methods are recommender systems.

As demonstrated in Chapter 2, there are many different recommendation algorithms. Examples include Slope One predictions, SVD factorization and Nearest Neighbor weighted predictions (see Section 2.3.3). These methods predict unknown connections between users and items based on some pattern in the data, for example user profile similarity, rating correlations or social connections. As previously explained, to achieve the best possible combined result, we wish to use methods that look at disjoint patterns, i.e. complementary predictive parts of the data (see Section 2.5).

The *Adapters* part of our 6-tuple refers to the second level of recommender systems. In traditional prediction aggregation this step is a simple linear function for combining the different predictions, for example by pre-computing a set of weights, one per method. As found by Bell et al. (2007b, p.6) the accuracy of the combined predictor is more

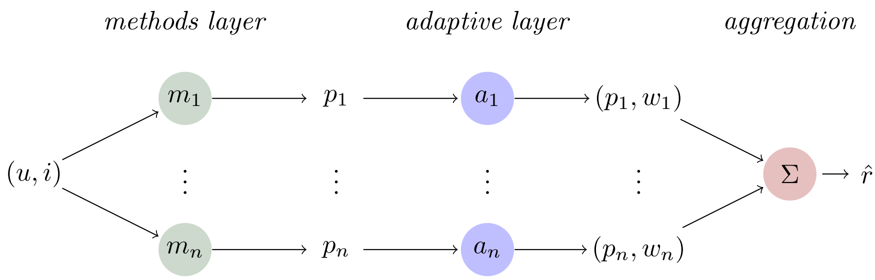


Figure 3.2: Layers of recommenders: The method layer consists of ordinary modeling methods, each predicting the rating between a user and an item. This produces a set of predicted ratings (p). The adaptive layer estimates how well each modeling method will perform for the current user and item, and weighs the predictions accordingly. This produces a set of predictions and weights [(p, w)]. The aggregation weighs the predictions into a final score $\hat{r}$.

dependent on the ability of the various predictors to expose different aspects of the data, than on the individual accuracy of the predictors. As described in Section 2.5, a set of recommender systems can be combined by estimating generalized weights that minimize the prediction error across all users.

With adaptive recommenders, the *Adapters* are themselves recommender systems (see Figure 3.1). However, instead of modeling users, we wish to model the behavior of the recommender systems. More specifically, we wish to model the *accuracy* of our recommender systems. Methods in this second layer are used to predict how accurate each of their corresponding basic recommenders will be. It is these methods that will allow us to do adaptive aggregation based on the current user and item. We then have two distinct layers of recommender systems (see Figure 3.2):

1. *The methods layer* consists of traditional recommender systems, that use a single aspect of the data to produce predictions. When presented with an item and a user, these methods produce a predicted rating $\hat{r}_{u,i}$ based on their algorithms.

2. *The adaptive layer* is another set of corresponding recommenders that work a bit differently. These methods take an item and a user and estimates how well its underlying method will perform this prediction. The accuracy estimations are then combined with the predictions by aggregation. However, each of these adaptive methods do not have to employ the same algorithm as their corresponding methods.

Another way of describing (and implementing) the two levels is with the map and reduce functions of functional programming. We can express prediction aggregation as:

$$\hat{r}_{u,i} = \text{reduce}(u, i, \text{map}(M, u, i)).$$

First, each modeling method is applied by the map function, with the current user, item and set of modeling methods as input. This operation returns a set of scalar prediction values. These values are then combined by the reduce function, which also takes the current user and item as input. In our terms, map is the methods layer, and reduce is the adaptive layer. If we wish to do rank aggregation, the equation is a bit different:

$$\tau_{u,n} = \text{reduce}(u, \text{map}(M, u, n)).$$

Here, τ_u is the list of recommended items for user u (following the notation in Dwork et al. (2001, p.3)). Note that there is no input item in this formula as we wish to produce a ranking of the top n recommended items. The result is an adaptively sorted list of the top n items for the current user. A common use case for rank aggregation is personalized search: an IR algorithm restricts the item space, which is then adapted by recommender systems, as we shall see.

Expressing ourselves in terms of map and reduce is helpful as this will guide our implementation. Note that our terminology is a bit different from the proper MapReduce framework for parallel computation (as explained in Manning et al. (2008, p.75)). However, as with the standard key/value approach to MapReduce, the fact that our tasks can be run in parallel will help us implement efficient algorithms.

3.3.1 Adaptive Aggregation

To perform adaptive recommender aggregation, we need the *Adapters* to be actual recommender systems. Until now we have talked about both prediction aggregation (scores) and rank aggregation (sorted lists). For now we shall stick to scalar predictions, but will return to rank aggregation in Section 3.5.

The simplest generalized way of prediction aggregation is to take the average of all predictions made by the different methods (e.g. Aslam and Montague (2001, p.3)):

$$\hat{r}_{u,i} = \frac{1}{N} \sum_{m \in M} p(m, u, i).$$

Here, $\hat{r}_{u,i}$ is the estimated rating from user u to item i, N is the number of methods in M, and $p(m, u, i)$ is the predicted rating from method m. To achieve an even more optimal result, many aggregators weigh each method differently (e.g. Claypool et al. (1999)):

$$\hat{r}_{u,i} = \sum_{m \in M} w_m \times p(m, u, i) \quad \text{where} \quad 0 \leq w_m \leq 1, \quad \sum_{m \in M} (w_m) = 1.$$

$$
\begin{tikzcd}
 & \text{ratings model} \arrow[r] & \text{predicted rating} \arrow[dr] & & \\
(\text{u,i}) \arrow[ur] \arrow[dr] & & & \text{adaption} \arrow[r] & \hat{pr} \\
 & \text{error model} \arrow[r] & \text{predicted error} \arrow[ur] & &
\end{tikzcd}
$$

Figure 3.3: Multiple models for adaptive weights: The data flow through the adaption of a single recommender method. The current user and item is fed into two distinct models. The ratings model, which predicts unknown ratings, and the error model, which predicts how accurate this rating will be for the current input. The two predictions are then aggregated into a final part of a rating ($\hat{pr}$). Each of the recommenders contribute parts to the final rating.

In this equation, w_m is the weight applied to modeling method m. These weights fall in the range $[0, 1]$ and sum up to 1. As described in 2.5, these weights can be estimated by different machine learning methods. However, as discussed in Section 3.1, this is still a generalized result, averaged across every user and item. The system assumes that the best average result is the best result for individual users and items. This means that, even with method-specific weights, we are still hindered by the latent subjectivity problem.

In order to leverage as many patterns as possible while sidestepping any latent subjectivity, we need *adaptive weights* that are computed specifically for combinations of users and items. This is more difficult than simply estimating generalized weights. If we wish the weights to be combination-specific, then pre-computing weights for each method becomes unfeasible. We would have to compute a weight for every method for every possible rating. The adaptive weights also have to be estimated just as the unknown ratings:

$$\hat{r}_{u,i} = \sum_{m \in M} p_w(m, u, i) \times p_r(m, u, i) \quad \text{where} \quad \sum_{m \in M} (p_w(m, u, i)) = 1.$$

Here, $p_w(m, u, i)$ is the predicted optimal weight for method m when applied to user u and item i. Adaptive recommenders is one way to estimating these weights, i.e. one way to implement p_w.

We wish to use standard recommender systems for predicting optimal adaptive weights. To do this, we need to create a matrix (or graph) that stores known values of how accurate some of the rating predictions will be.

The key insight is that *the predicted accuracy of a method is the opposite of its predicted error*. By modeling the errors of a method through standard recommender systems, we can in turn predict errors for untested combinations (see Figure 3.3). If we predict the error of a recommender system for a user and an item, we have also predicted its accuracy. To achieve this, we create an *error matrix*:

$$E_{u,i} = \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,i} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ e_{u,1} & e_{u,2} & \cdots & e_{u,i} \end{pmatrix}$$

Creating an error matrix for each modeling method is done by splitting the ratings data in two. The first set can be used for the actual RS modeling, and the second can be used to populate an error matrix for each RS.

With adaptive recommenders, the standard modeling methods produce error matrices, where some of the cells have values. A value in this matrix corresponds to the prediction error for a combination of a user and an item. To achieve this, each modeling method is only trained with a part of the ratings data. The error matrix is populated from the rest of the data, by computing the error of every known rating the method was not trained for:

$$\forall (u, i, r) \in (d_e - d_m) : E(m)_{u,i} = |r - p(m, u, i)| \quad \text{where} \quad d_e, d_m \subset D$$

Here, D is the current dataset, and d_m and d_e are subsets of D. m is a modeling method trained with the subset d_m. To populate the error matrix for this method, we take every rating which have not been used to train the method and calculate the error of the method on this combination. Since we are only interested in the magnitude of the error, we take the absolute value of the measurements. The result is a sparse error matrix we can use to predict unknown errors.

Notice the similarity of this matrix and the previously introduced ratings matrix. This similarity is what will allow standard recommender systems to perform adaptive aggregation. Whenever we wish to train a new modeling method, we apply the following algorithm:

1. Split the ratings data into two sets for training and error estimation.
2. Train the modeling method in its specific way with the first training set.
3. Use the error estimation data set to create the error matrix.
4. Train an error model based on the error matrix.

The error models are trained using standard recommender systems. After all, the expected input and output is the same. We have two dimensions, with a sparse set of known connections, and wish to predict unknown connections from this data.

The result is a set of modeling methods that can predict the error of a recommender system when its used on a particular user and item combination.

What will happen when we train a recommender system with the error matrix? First of all, the errors will be on the same scale as the initial ratings. Second, just as the ratings matrix will include noise (ratings that do not contribute to any underlying pattern), this will be true for the error matrix as well.

For example, one method might have a large error for a particular user and item combination, yet still work well for both these elements. However, this is just the kind of noise recommender systems are good at pruning away. What we are interested in are situations where a method has stable and significant errors for many ratings from a user, or many ratings of an item. In this case, there is a pattern where this method does not work well for the element in question. This is exactly the kind of pattern recommender systems are good at identifying.

The same capabilities that makes recommender systems work well on the ratings matrix, will also make them work well on the error matrix. The properties we need for predicting ratings are the same as those needed to predict accuracy.

Of course, some recommender systems will work better than others for the adaptive layer. Most often we are seeking global patterns in the data. We are looking for groups of users or items (or both) that suite some recommenders especially well, or that some recommenders will not work for. SVD-based recommenders is one type of RS that can be used for this purpose. By reducing the method-error space into an *error category space*, we can identify how well a set of groups suite the available methods. We will get back to this when performing experiments in the next chapter.

When we have an error model for each modeling methods we can estimate optimal weights. Whenever we wish to create an adaptive aggregate prediction, we apply the following algorithm:

1. Collect predictions from the modeling methods for (u, i).
2. Collect estimated errors for each method for (u, i).
3. Compute weights for the methods based on their relative predicted errors.
4. Sum the weighted predictions to get the adaptively predicted rating.

The next section will explain these steps in detail. We can now express the prediction phase of adaptive recommenders as an equation. Each rating/relevance prediction is weighted by its predicted accuracy, conditioned on the current user and item:

$$\hat{r}_{u,i} = \sum_{(m_e, m_r) \in M} \left(1 - \frac{p(m_e, u, i)}{error(u, i)}\right) \times p(m_r, u, i) \quad \text{where} \quad error(u, i) = \sum_{m_e \in M} p(m_e, u, i)$$

In this equation, each recommender method has two corresponding models: m_r is the ratings model, used to predict ratings, and m_e is the error model, used to predict errors. $p(m, u, i)$ is the prediction of the model m (a recommender system) for the relevance of item i to user u. This means that a method is weighted by its predicted accuracy. The weights are computed by taking the opposite of a methods predicted error. The errors are normalized across users and items by the $errors(u, i)$ function, which is the sum of the errors of the methods for the current combination. This gives us weights in the range $[0, 1]$ ensuring final rating predictions on the same scale as that returned by the basic recommenders.

Notice that the *only* difference between m_e and m_r is how they are created. m_r is trained with the standard ratings matrix, and m_e is trained using the error matrix. This means we can use *any* standard recommender system to perform adaptive aggregation. Hence, the name *adaptive recommenders*: a set of secondary recommenders is used to adapt a set of standard recommenders to each user and item.

It is also important to note that the types of recommenders used for the adaptive layer is independent of the basic recommenders. The adaptive recommenders need only predicted ratings from the basic recommenders, and does not care which algorithms they employ. When making predictions, the calculations in the methods layer and adaptive layer are independent, as both use pre-computed models: the method layer use the ratings matrix, or their own models created during training, while the adaptive layers use the error matrices for the basic methods.

The result of this is a system that does not only aggregate a number of predictions for unknown combination of users and items, but that also combines these methods based on how accurate each prediction is likely to be.

Let us now see how adaptive recommenders may be implemented. We shall first do prediction aggregation in a recommendation scenario, then rank aggregation in an information retrieval scenario.

3.4 Prediction Aggregation

Adaptive prediction aggregation means combining the results from multiple scalar predictors conditioned on the current context. As mentioned, we have two levels of predictors. The first level is a set of traditional recommender systems that produce estimations of unknown ratings between users and items. The second level is another set of recommender systems that predict how accurate each of the first level recommenders will be. There are two distinct phases when using adaptive recommenders:

1. The modeling phase creates the user models for both levels.
2. The prediction phase uses the created models to estimate ratings.

We shall first explain the modeling phase, then the prediction phase. The next section will explain a similar situation where we wish to do *adaptive rank aggregation* by combining ordered lists of results, depending on the current user and item.

3.4.1 Modeling Phase

Listing 1 gives the basic algorithm for training our models. The input to this method is the standard ratings matrix, and a set of untrained modeling methods (in this case, untrained recommender systems).

Algorithm 1 Adaptive Prediction Aggregation Modeling

Input: ratings: The ratings matrix
Input: methods: The set of modeling methods
Output: rating_models, error_models: trained rating and error models

1. $rating\_models \leftarrow \emptyset$
2. $error\_models \leftarrow \emptyset$
3. **for all** $m \in methods$ **do**
4. $\quad sample \leftarrow \text{BootstrapSample}(ratings)$
5. $\quad rating\_models_m \leftarrow \text{TrainModel}(m, sample)$
6. $\quad error\_models_m \leftarrow \text{TrainErrorModel}(rating\_models_m, ratings)$
7. **end for**
8. **return** $(rating\_models, error\_models)$

An important question is how we should split the ratings data. In this scenario, we need to split the data for a number of purposes. The following sets must be created during training:

1. Training sets for the standard recommenders.
2. Training sets for the error estimation models.
3. A testing set to measure the performance of our final system.

Constructing these subsets of the available data is a common task in ensemble learning (Polikar, 2006, p.7). As seen in Listing 1, we use an approach called *bootstrap aggregation*, also known as *bagging* (introduced by Breiman (1996)). Originally, bagging is used by ensemble learning classification methods, where multiple classifiers are trained by uniformly sampling a subset of the available training data. Each model is then trained on one of these subsets, and the models are aggregated by averaging their individual predictions.

Formally, given a training set D with n items, bagging creates m new training sets of size $n' \leq n$ by sampling items from D uniformly and with replacement. In statistics, these types of samples are called *bootstrap samples.* If n' is comparable in size to n, there will be some items that are repeated in the new training sets.

Bagging suits our needs for a few reasons. First, it helps create disjoint predictors, since the predictors are only trained (or specialized for) a subset of the available data. When using multiple similar recommenders, this means we can create specialized models for parts of the data with a higher performance than if they were trained on the entire dataset. Second, bagging allows us to easily train the underlying modeling methods without any complex partitioning of the data. To partition and use the available data, we use the following algorithm:

1. Split the entire dataset into a training and testing set.
2. Train modeling methods through bootstrap aggregation of the training set.
3. Train error models from the complete training set.
4. Test the resulting system with the initial testing set.

Each modeling method is trained in ways specific to their implementation. Model-based approaches create pre-built structures and provide offline training, while heuristic methods simply store the data for future computation. Either way, it is up to each modeling method what it does with the supplied training data. The result of this algorithm is a set of trained rating models and error models.

Algorithm 2 Prediction Error Modeling

Input: ratings: the ratings matrix
Input: rating_model: a recommender system user model
Output: error_model: a trained error model for this method

$errors \leftarrow [[]]$
for all $user, item, rating \in ratings$ **do**
    $errors_{user,item} \leftarrow |ratings_{user,item} - \text{Predict}(rating\_model, user, item)|$
end for
$error\_method \leftarrow \text{NewModelingMethod}(SVD)$
$error\_model \leftarrow \text{TrainModel}(error\_method, errors)$
return $error\_model$

Listing 2 shows an algorithm for training the error models. The input is the entire ratings matrix, and a trained recommender model that this error model should represent. We first create the aforementioned error matrix by estimating predictions for each known combination in the ratings data. The NewModelingMethod call simply creates a new, untrained recommender model of some pre-specified *type* (in this case, a new SVD-based model, but any applicable RS will do). A new model is then trained based on the created error matrix, and returned as our new $error\_model$.

When the computations of the algorithm in Listing 1 is complete, we have a set of trained recommender systems, and a set of trained error models. Each recommender model has a corresponding error model, forming two layers, that we shall use when performing predictions.

3.4.2 Prediction Phase

In the prediction phase of adaptive prediction aggregation, we wish to use our layers of trained models to produce adaptive combinations of multiple predictions and accuracy estimations. Listing 3 gives the basic algorithm.

Algorithm 3 Adaptive Prediction Aggregation

Input: user, item: a user and an item
Input: rating_models: the set of trained modeling methods
Input: error_models: the set of trained error models
Output: prediction: the predicted relevance of the item to the user

$ratings \leftarrow \emptyset$
$errors \leftarrow \emptyset$
for all $m \in rating\_models$ **do**
  $ratings_m \leftarrow \text{Predict}(rating\_models_m, user, item)$
  $errors_m \leftarrow \text{Predict}(error\_models_m, user, item)$
end for
$errors \leftarrow \text{Normalize}(errors)$
$prediction \leftarrow 0$
for all $m \in rating\_models$ **do**
  $weight_m \leftarrow 1 - error_m$
  $prediction \leftarrow prediction + weight_m \times ratings_m$
end for
return $prediction$

The first input is the user and item for which we wish to predict a rating. We assume that this rating is unknown — predicting ratings for known combinations would mean recommending items the user has already seen and considered (however, if we are dealing with a task such as personalized search, these known ratings are important, as we shall see in the next section).

The other inputs are the trained rating models, and the corresponding error models. The algorithm begins by creating empty sets for predicted ratings and errors. Next, the modeling methods are used to predict ratings, and their error models to predict errors. Note that the step in the first for-loop are independent, and both steps inside the for loop are also independent. This is then an algorithm well suited for parallelization. In a MapReduce framework, this for loop would be run as a map operation, where the input user and item is mapped over the sets of modeling methods (see Appendix A for implementation details).

After the predictions have been collected, the errors are normalized, i.e. converted to the range $[0, 1]$, so that they sum to 1. This is vital before last stage of the prediction algorithm, which weighs the predictions from the different rating models. The last step corresponds to the previously explained reduce operation, that combines multiple scores into one final result. The weights of the methods are computed as $1 - error$, where $error$ is the normalized error for this method, for the current user and item. The rating predictions are then multiplies with their weight, and combined to form the final adaptively aggregated prediction.

There is an important performance different between the modeling and prediction phases: While the modeling phase is the most computationally expensive, it can be performed independently of making predictions. As the prediction phase is when the user has to wait for the system, this is where performance is most important.

As users rate more items and new items arrive, the models have to be recreated based on this new reality. However, as the modeling phase is an offline operation, the training can be performed in the background, while new and computationally efficient predictions are always available.

3.5 Rank Aggregation

It is time to see how to do *adaptive rank aggregation.* Rank aggregation means combining sorted lists of items. In this scenario, the modeling methods take the current user as input, and produce lists of items ranked in order of rating (see also Section 2.5.1).

Aggregating lists is desirable in a number of situations. Often we wish to produce lists of recommended items, not just estimate the rating of a single user/item pair. Consider the task of personalizing a list of search results (see Section 2.4). The important part is not the score given to each result, but rather the order in which they appear. The underlying technology stays the same. A number of recommenders are used to predict the ratings of items to users. However, to do rank aggregation, another layer is added, that requests lists from each method, not only singular items.

Because it is such an important use case, we shall use personalized search to present our approach to adaptive rank aggregation. In addition to the standard recommenders, we have an information retrieval method, as introduced in Section 2.4.1. The IR method takes in a user-initiated query (a collection of words or a sentence), and returns a number of search results, in an ordered list. In traditional personalized search, a recommender system can then be used to estimate a rating for each of the returned items, and re-sort, or re-score, the results list (e.g. Xu et al. (2008, p.3)).

The key insight is that both the IR method and the recommender systems form *input signals* (see Section 2.4.2). An input signal is some measure of how an item should be ranked in the final results list. The relevance scores returned from our IR ranking functions are signals, and the predicted ratings from the recommender systems are signals. Adaptive aggregation then entails estimating *how accurate each of these signals are likely to be for the current user and item.* This is almost the same task as in adaptive prediction aggregation, only in a list-oriented fashion.

There is an important difference. The IR methods should be used to constrict the range of items worked on by the recommender systems. As the IR methods identify items that may be relevant to the users query, these are the items we wish the recommender systems to work on. This goes back to the previously mentioned difference between *search* and *recommendations*:

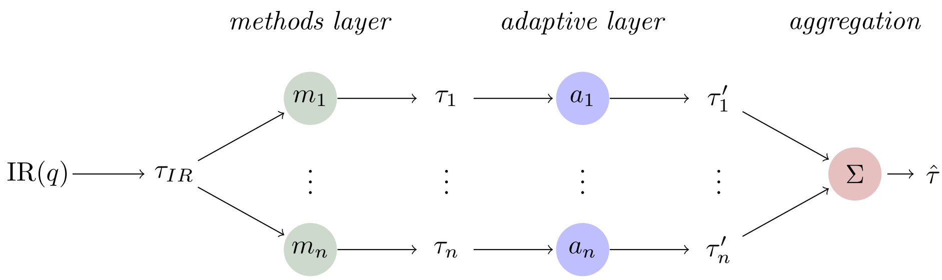


Figure 3.4: Example of Adaptive Rank Aggregation: An IR method returns a results list of possibly related items, together with a ranking score. The methods layer estimates ratings for items in the results list. The adaptive layer predicts how accurate these ratings are likely to be. Finally, the ranking scores, ratings and accuracy estimations are combined into one result list, $\hat{\tau}$.

- Recommenders find relevant items the user does not already know exists.
- Search engines find relevant items the user knows or hopes exists.

The difference lies in the knowledge of existence. As personalized search is still a search task, the IR methods should determine the set of items that might be relevant. Their relevance scores for these items becomes the first input signals. The recommender systems works on this set of items, re-scoring them as needed. We still have the adaptive layer that estimates how well the signals will perform for the current user and item. This is especially important considering that we may have multiple IR methods that define multiple sets of relevant items. The final result is an adaptive combination of the rating and accuracy predictions for each signal, as seen in Figure 3.4. Let us now see how the modeling and prediction phases are performed in adaptive rank aggregation.

3.5.1 Modeling Phase

We shall only deal with settings where we have a single IR method. While multiple IR methods and corresponding error models is an interesting setting, we are most interested in using the IR method for constraining the Item-space considered by the recommender systems. As we shall see, this does not introduce many changes to our algorithms.

The modeling phase for the recommender system stays almost the same, with one important change. As we are dealing with a search engine, we might not have an explicit ratings matrix to rely on. Most feedback we can gather from user initiated searches are from query logs. These logs show the current user, query, and the item that is finally selected after the query is performed. Query log mining is a common approach in personalized search (e.g. Liu et al. (2002); Sugiyama et al. (2004); Shen et al. (2005); Speretta and Gauch (2000)). By mining query logs we can create an implicit ratings matrix. Each populated cell represents a selected item.

For example, Venetis et al. (2011) shows an interesting approach where they use requests for directions from online map services to infer implicit ratings: when a user asks for the directions from A to B, this is taken as a vote from this user that location B is interesting. This is just one of many ways implicit ratings matrices can be mined.

The values in this implicit ratings matrix can take many forms. If we only care about selected items, binary ratings may suffice: selected items are then represented by a 1 in the ratings matrix. These ratings can be further improved by considering different metrics, including:

- Time spent before selecting the item.
- The items initial placement and the effort required to identify it.
- How far the user was willing to scroll before clicking the item.
- Whether or not the user resubmitted the same query shortly after.

Based on these and other similar metrics, one can achieve quite accurate implicit ratings. Naturally, ratings can also be gathered from other sources. If we have more data on each user, or know of secondary systems such as social networks or other systems where ratings are present, these can be used to augment the implicit ratings matrix (e.g. Carmel et al. (2009)). There are also search systems where we already have explicit ratings. Consider, for instance, the use case of searching for movies on a movie rating site, or searching for people in a social network. In these cases, we have explicit ratings that can be used to train the recommender models.

A thorough explanation of turning query logs into ratings matrices is beyond the scope of this thesis. Extensive research already looks at how implicit user model information may be gleaned from simple query logs. Examples include Joachims et al. (2007), Lee and Liu (2005), Agichtein et al. (2006), Mobasher et al. (2000) and Speretta and Gauch (2000).

As in prediction aggregation, the strength of our resulting system is in large part dependent on the accuracy of our ratings. This means that deciding and understanding how implicit ratings are created, or finding auxiliary sources to provide explicit ratings, is a critical step. The algorithms are only as strong as the data they use. Methods for personalized search may work best in settings where we have explicit ratings, or can gather explicit ratings from secondary sources, for example from external social networks or publishing platforms.

When we have the implicit or explicit ratings matrix, the modeling phase consists of two parts. Training the IR models and the recommender models. The recommender models are trained as before, given in Listing 1. The one or more IR methods are not trained with a ratings matrix, but with the items and their respective data. Of course, the actual IR modeling method depends on the IR system itself. However, as a through explanation of IR systems is outside our scope, we assume that the IR model is trained with the system's items, and that it returns relevant items when given a query.

Algorithm 4 Adaptive Rank Aggregation

Input: user: the current user
Input: items: the set of all items and their meta-data
Input: query: the user initiated query
Input: ir_model: a trained IR model
Input: rating_models: the set of trained modeling methods
Input: error_models: the set of trained error models
Output: results: the adaptively sorted list of results

$ratings \leftarrow \emptyset$
$errors \leftarrow \emptyset$
$results \leftarrow \text{Search}(ir\_model, items, query)$
for all $item \in results$ **do**
  for all $m \in rating\_models$ **do**
    $ratings_{m,item} \leftarrow \text{Predict}(rating\_models_m, user, item)$
    $errors_{m,item} \leftarrow \text{Predict}(error\_models_m, user, item)$
  end for
end for
$errors \leftarrow \text{Normalize}(errors)$
for all $item, ir\_score \in results$ **do**
  $prediction \leftarrow w_{IR} \times ir\_score$
  for all $m \in rating\_models$ **do**
    $weight_{item} \leftarrow 1 - error_{m,item}$
    $prediction \leftarrow prediction + weight_m \times ratings_{m,item}$
  end for
  $item_{prediction} \leftarrow prediction$
end for
$results \leftarrow \text{SortByPredictions}(results)$
return $results$

3.5.2 Prediction Phase

The prediction phase is where adaptive rank aggregation differs most from adaptive prediction aggregation. Listing 4 gives the basic algorithm. As input, instead of one item, we have the entire set of items, and a query. We run the query and items through the IR model to get the constrained set of items (*results*). Each of the recommender methods is then run for the items in the results list. As before, the first for-loop can be performed in parallel. The calls to *Predict* are independent of the other calls, allowing us to perform it as a *map* operation.

As before, the error estimations are normalized before converting them to weights. Since we are dealing with two dimensions of errors, for each item and method, the errors are normalized across items. For items, the errors from the recommenders fall in the range $[0, 1]$ and sum to 1.

After the items in the results list have an IR score, a set of predictions, and a corresponding set of error predictions, the adaptive aggregated prediction is computed.

Because we do not care of the final score we set the initial predictions to be the IR scores. The recommender systems simply add or subtract from this initial score. This means that the resulting predictions will not be in the same range as the known ratings, but since we are only interested in the order of the items, not the actual rating, this is not a problem.

An important coefficient is the w_{IR} (IR weight), which determines how much the IR method should decide of the final ranking. This is first and foremost adjusted to ensure that the IR scores are on a similar scale as the predicted ratings. At the same time, this weight determines how much the IR score influences the final placement of an item in the results list. In the next chapter, we shall see how the choice this parameter influences the ranked results lists.

After computing the predictions for each item in the results list, we sort the list by the item predictions, and return the list. The resulting list is adaptively sorted based on the current user and the specific items in the list, achieving adaptive rank aggregation.

Listing 4 considers rank aggregation in a scenario with a single IR model. In the case of more IR models, we would combine the scores for the items returned by the different models. In this case, estimating the accuracy of each IR model in the same way as the recommender systems would provide another level of personalization. Just as varying RSs work differently for elements, each IR model may have varying applicability to individual items. At the same time, users might place a different importance on the varying IR models.

This would be a simple extension of our prediction algorithm. The most important question becomes how to estimate the error matrix for an IR model. There are many ways to do this. For example, the error for a query could be based on how how often the element in question is selected for the current query. Another error might be based on the difference between the optimal placement (based on click-through rates) and the actual placement of an item in the result list.

However, as this is outside the scope of this thesis, we shall stick to situations where we have a single IR model. Experiment 3 in the next chapter will show how this IR model can be used in different ways by varying the IR weight.

The implications of adaptive rankings of search results are considerable. By estimating how accurate each algorithm will for current user, we get a list of results that is sorted based on how the user *should* be modeled. At the same time, by estimating accuracies for the current item, the items are sorted by the algorithms that can best predict their relevance.

This is the main strength of adaptive recommenders. Instead of a generic and averaged sorting, we achieve an adaptive sorting that creates an optimal algorithmic combination based on the current context. We shall discuss this further in the Chapter 5.

— 4 —

Experiments & Results

This chapter will perform experiments to find out whether or not *adaptive recommenders* is a viable technique. We will perform three experiments to test the three hypotheses outlined in Section 3.2. The first two experiments will take a quantitative look at prediction aggregation, while the last will be a case study of using adaptive recommenders for personalized search. The next chapter will discuss the implications and limitations of each experiment.

4.1 Three Experiments

Table 4.1 shows the experiments that will test our technique. Experiments 1 & 2 will test hypotheses H1 & H2. We will measure the performance of adaptive recommenders compared to standard and aggregate recommenders. Experiment 3 will test hypothesis H3. We will try using our method to personalize sets of search results in a number of ways. The first two experiments will be quantitative measurements of prediction aggregation performance. The last experiment will be a qualitative exploration of personalized search with adaptive recommenders. In particular, we will look at how different prioritizations of the IR model scores influence the final rankings.

	mission	*hypotheses*	*dataset*	*users*	*items*	*ratings*
Experiment 1	pred.agg.	H1, H2	MovieLens	943	1,682	100,000
Experiment 2	pred.agg.	H1, H2	Jester	24,983	100	1,832,275
Experiment 3	rank.agg.	H3	MovieLens	6,040	3,900	1,000,000

Table 4.1: List of Experiments performed in this chapter.

As seen in Table 4.1, we will use two distinct datasets in the experiments. Each dataset have different numbers of items, users and ratings. This is a desirable property. Testing adaptive recommenders in different scenarios will help us achieve more reliable results.

First is the MovieLens dataset[1]. This dataset is often used to test the performance of recommender systems (as described in Alshamri and Bharadwaj (2008, p.9), Lemire and Maclachlan (2005, p.4), Adomavicius and Tuzhilin (2005, p.1) and Herlocker et al. (2004, p.2)). It consists of a set of users, a set of movies, and a set of movie ratings from users, on the scale 1 through 5. We chose two subsets of the entire MovieLens collection. For Experiment 1 we use a subset of 100,000 ratings from 943 users on 1,682 movies. For

(1) See http://www.grouplens.org/node/73 — accessed 10.05.2011

Experiment 3 we use a much larger subset in order to have more items available for the IR model. This subset has 1,000,000 ratings from 6,040 users of 3,900 movies.

The MovieLens dataset also comes with meta-data on users, such as gender, age and occupation. There is also meta-data on movies, such as its title, release date and genre. In Experiment 1, we are only interested in the ratings matrix of this dataset. The titles of movies will be used in Experiment 3.

Our second set of ratings comes from the Jester dataset[2]. This is a set of 100 *jokes* rated by users on a continuous scale. As with MovieLens, this dataset is also commonly used to test recommender systems (as described in Goldberg et al. (2001), Herlocker et al. (2004, p.14), Adomavicius and Tuzhilin (2005, p.5) and Ahn and Hong (2004, p.30)). This dataset has many more users than those used in the other experiments. On the other hand, there are significantly fewer items than in the other dataset. The widely varying number of items, users and ratings in our selected datasets will give us more dimensions along which to verify our results.

Jester also has ratings on a different scale than the MovieLens dataset. While the movies are rated on a discrete scale from 1 through 5, the items in Jester are rated on a continuous scale from -10 to 10. However, in order to easily compare the measurements on both datasets, the ratings in Jester were converted to be on the scale $1-5$. Still, the difference between ordinal and continuous ratings remains, and will give us another differing quality to verify our results.

In another effort to achieve reliable results, both datasets were split into multiple disjoint subsets. We need disjoint subsets in order to perform cross-validation testing. This entails running the same experiments across all subsets and averaging the results. Each dataset is split into five sets which are again split into training and testing sets:

$$D_n = \{d_1 = \{base_1, test_1\}, d_2 = \{base_2, test_2\}, ..., d_5 = \{base_5, test_5\}\}$$

The $base_x$ and $test_x$ pairs are disjoint 80% / 20% splits of the data in the subsets. We shall perform five-fold cross-validation across all these sets in our experiments. This way we can be more certain that our results are reliable, and not because of local effect in parts of the data. As previously explained, the *base* sets are further split using bootstrap aggregation, into random subsets for training the standard recommender models. The entire base set is then used to train the adaptive error estimating recommenders. The corresponding *test* sets will be used to evaluate our performance on the subsets.

Before performing our experiments, let us take a closer look at the different types of recommender systems we will use.

(2) See eigentaste.berkeley.edu/dataset/ — accessed 22.05.2011

	method	*algorithm*	*description*
S	svd1	SVD	ALSWR factorizer, 10 features.
S	svd2	SVD	ALSWR factorizer, 20 features.
S	svd3	SVD	EM factorizer, 10 features.
S	svd4	SVD	EM factorizer, 20 features.
S	slope_one	Slope One	Rating delta computations.
S	item_avg	Baseline	Based on item averages.
S	baseline	Baseline	Based on user and item averages.
S	cosine	Cosine similarity	Weighted ratings from similar items.
S	pcc	Pearson Corr.	Weighted ratings from similar users.
A	median	Aggregation	Median rating from the above methods.
A	average	Aggregation	Average rating from the above methods.
A	adaptive	Adaptive agg.	Accuracy predictions from error models.

Table 4.2: Adaptive modeling methods: a short overview of the recommender methods used in our experiment. Each recommender is used in every experiment. See Section 2.3 for more information.

4.2 Recommenders

Standard recommender systems will be used for both the basic predictions, and the accuracy estimations, as described in Chapter 3. Naturally, we need a number of different RSs, preferably ones that consider disjoint patterns in the data. Table 4.2 gives a short overview of the recommender systems we shall employ. All three experiments will use every recommender in this table. This section gives a short introduction to these methods. See Section 2.3 for more information on RSs, and Appendix A for details on how these were implemented in this system.

4.2.1 Basic Recommenders

As seen in Table 4.2, we have two types of recommenders. First, we have the basic recommenders, denoted by S in the table. Each of these systems look at varying data patterns to predict ratings. We chose this wide range of recommenders for just this reason. As previously explained, the performance of aggregation methods are more dependent on the dissimilarity of the basic recommenders than their individual performance (Bell et al., 2007b, p.6).

Let us briefly explain how each basic RS works. The SVD methods look for global patterns in the data by reducing the ratings-space into a concept-space. By reducing this space, the algorithm is able to find latent relations, such as groups of movies that has the same rating pattern, or groups of users that often rate in a similar manner. For the SVD methods, the factorizers refers to algorithms used to factorize the ratings matrix (see Section 2.3.3).

The Slope One and baseline algorithms look at average ratings for items and from users, and use these to predict ratings. These are simple algorithms that often perform as well as more complex approaches.

The cosine similarity algorithm looks for items that are rated similarly by the same users, and infers item similarity from this measure. New ratings are then predicted by taking known ratings of other items, weighted by their item's similarity to the new item. This is based on the same method used in the previously introduced vector space model.

The PCC algorithm employs yet another approach. This algorithm, similar in strategy to the cosine similarity algorithm, looks for users with similar rating patterns. The similarity is measured with the Pearson Correlation Coefficient. Predictions are created by collecting ratings from similar users of the item in question, weighted by their respective similarity. See Section 2.3 for more detailed information on how these recommenders work.

The main difference between our recommenders are the scope of the patterns they leverage. The SVD and baseline methods look at global effects, such as latent categories and overall rating averages. The cosine similarity and PCC algorithms look at smaller clusters of similar users and items, and compute an average rating weighted by similarities of elements. This wide range of recommender should give us the desired effect of looking at disjoint rating patterns.

4.2.2 Adaptive Recommenders

The second type of recommenders are the aggregation methods, that combine the result of each of the basic system (the methods below the middle line in Table 4.2, denoted "A").

The first two of these methods are simple aggregation approaches. These will be used to test hypothesis H2. The median aggregation method choses the median value of the predictions produced by the standard recommenders. Similarly, the average aggregation method takes the mean of the standard predictions. While not complex in nature, these methods will help us see how our method compares to simple aggregation techniques.

The last entry in Table 4.2 refers to our technique. This is the recommender outlined by the algorithms in Chapter 3, that create secondary accuracy estimating recommender systems, in order to adaptively weigh the basic recommenders. All the aggregation approaches, including our technique, employ every basic recommender system described so far.

As explained in Section 3.3, any basic recommender system can be used for the adaptive method. The only difference is how this method is trained. While the basic methods are trained using the ratings matrix, the adaptive methods are trained using the error matrix, as seen in Listing 1. We have as many possibilities for choosing the adaptive recommenders as the basic recommenders.

For our experiment, we chose SVD recommenders for the adaptive models. That is, the basic recommender methods get secondary accuracy predicting recommenders, which in this case are standard SVD-based recommender systems. The SVD recommender is a natural choice in this case, since we wish to uncover latent patterns of accuracy for each model. SVD recommenders look for global patterns in the data, which in this case would mean situations where a standard RS works especially well for, or does not work for, a set of users, items, or a combination of these.

It is important to note that the same configuration of recommenders was used for all three experiments. Neither the basic nor the aggregate or adaptive recommenders were heavily tailored to the datasets. To be sure, higher performance could probably have been achieved by tailoring the recommenders to the available data. However, as our goal is to compare our finite set of methods, we are currently only interested in how they perform compared to each other.

As with the basic recommenders, the same SVD recommender configuration was used for the adaptive layer in every experiment. We chose to use en EM-factorizer to perform the actual decomposition, consisting of 20 features. The decomposition was performed by 20 iterations. See 2.3.3 for more information on how SVD-based recommenders work. The choices of recommenders will be further discussed in Chapter 5.

4.3 Evaluation Strategies

To evaluate how our model performs during prediction aggregation, we need a measure for computing the total error across a large number of estimations. The canonical measure for estimating the error of a recommender system is the *Root Mean Squared Error* (RMSE) measure (for example in Herlocker et al. (2004, p.17), Adomavicius and Tuzhilin (2005, p.13) and Bell et al. (2007b, p.6)). We shall use this measure to estimate the performance of our adaptive prediction aggregation algorithm. The RMSE of a set of estimations $\hat{R}$, compared the correct rating values R, is defined as:

$$\text{RMSE}(\hat{R}, R) = \sqrt{\text{E}((\hat{R} - R)^2)} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{R}_i - R_i)^2}{n}}.$$

Here, n is the total number of predictions. The RMSE combines a set of errors into one single combined error. A beneficial feature of the RMSE is that the resulting error will be on the same scale as the estimations. For example, if we are predicting values on the scale $1 - 5$, the computed error will be on this scale as well. In this case, an error of 1 would then say that we are on average 1 point away from the true ratings on our $1 - 5$ scale.

RMSE is a non-linear error estimator. This means that larger errors are harshly punished. Because the differences are squared by the formula, many small errors are much less significant than a few big errors. The RMSE will judge methods that provide stable predictions more favorably than methods that, while precise, have a few items or users for which the method breaks down. For example, when RMSE was used in the Netflix movie recommender challenge (Bennett and Lanning, 2007), the participating teams found that a few hard to predict movies often single-handedly severely impacted their total error.

While the RMSE works well for evaluating scalar predictions, we need another measure for evaluating rank aggregation methods. Here, we are not interested in the predicted scores, but rather in which position each item appears in a sorted list of results. This is for instance needed when measuring the performance of a personalized search engine. However, H3 does not state anything regarding explicit performance, only that our method should be applicable in an information retrieval scenario.

The performance of personalized search is hard to determine, as there are many types of rankings that make sense in a number of different use cases, as we shall see. In light of this, H3 will be experimented with as a case study, by looking at how our algorithm performs personalized search in a number of use cases.

4.4 Prediction Aggregation

Our first hypothesis, H1, states that *adaptive recommender aggregation can achieve higher accuracy than each of the combined recommenders.* The second hypothesis, H2, states that *adaptive recommender aggregation can achieve higher accuracy than generalized aggregation methods.*

In order to test these hypotheses, we performed adaptive prediction aggregation on the two datasets previously described. Five-fold cross validation was performed to further verify the result.

Table 4.3 gives the results from Experiment 1 (MovieLens). Table 4.4 gives the results from Experiment 2 (Jester). A cell corresponds to the RMSE values for a dataset, for each recommender and aggregation approach. The bottom entry in this table refers to our adaptive recommenders method. As seen in this table, the adaptive recommender achieved lower RMSE values than any of the other applied methods.

Statistics for the experiments are given in the last parts of Tables 4.3 & 4.4. The statistical values are the minimum, maximum and mean values for the methods. We also include the standard deviation (σ) across the collections of subsets. This table confirms the results from the full results table. Our adaptive recommenders approach improves the mean performance of our system. The mean RMSE values of Experiment 1, along with their standard deviation are shown in Figure 4.1.

Let us take a look at the standard deviation measures from the different methods. As seen in Figure 4.1, most of the methods, including the adaptive models, exhibit quite a lot of variation in their results. If these variations occurred as a result of unstable predictions of the same dataset, this would be a substantial problem, resulting in unreliable predictions. However, as seen in Figure 4.2 (based on Experiment 1), the standard deviation is mostly caused by the differing performance across the varying datasets. As we

Results from Experiment 1

	method	d_1	d_2	d_3	d_4	d_5
S	svd1	1.2389	1.1260	1.1327	1.1045	1.1184
S	svd2	1.2630	1.1416	1.1260	1.1458	1.1260
S	svd3	1.0061	0.9825	0.9830	0.9815	0.9797
S	svd4	1.0040	0.9830	0.9849	0.9850	0.9798
S	slope_one	1.1919	1.0540	1.0476	1.0454	1.0393
S	item_avg	1.0713	0.9692	0.9662	0.9683	0.9725
S	baseline	1.0698	0.9557	0.9527	0.9415	0.9492
S	cosine	1.1101	0.9463	0.9412	0.9413	0.9382
S	pcc	1.4850	1.1435	1.1872	1.2156	1.2022
A	median	0.9869	0.8886	0.8857	0.8857	0.8855
A	average	0.9900	0.8536	0.8525	0.8525	0.8519
A	adaptive	**0.9324**	**0.8015**	**0.7993**	**0.8238**	**0.8192**

	method	*min*	*max*	*mean*	σ
S	svd1	1.1045	1.2389	1.1441	0.2197
S	svd2	1.1260	1.2630	1.1605	0.2277
S	svd3	0.9797	1.0061	0.9865	0.0991
S	svd4	0.9798	1.0040	0.9873	**0.0924**
S	slope_one	1.0393	1.1919	1.0756	0.2415
S	item_avg	0.9662	1.0713	0.9895	0.2023
S	baseline	0.9415	1.0698	0.9738	0.2196
S	cosine	0.9382	1.1101	0.9754	0.2595
S	pcc	1.1435	1.4850	1.2467	0.3487
A	median	0.8855	0.9865	0.9065	0.2005
A	average	0.8519	0.9900	0.8801	0.2344
A	adaptive	**0.7993**	**0.9324**	**0.8352**	0.2225

Table 4.3: Results from experiment 1 (MovieLens): The cells give RMSE values for a method. The first table gives errors for subsets of the data (d_x). Lower values indicate better results. Bold values indicate the best result in a column. S *refers to singular methods, and* A *to aggregation methods.* σ *refers to the standard deviation of each method across the subsets.*

Results from Experiment 2

	method	d_1	d_2	d_3	d_4	d_5
S	svd1	1.0866	1.0815	1.0741	1.0823	1.08179
S	svd2	1.0831	1.0833	1.0739	1.0897	1.09024
S	svd3	0.9499	0.9440	0.9420	0.9480	0.94595
S	svd4	0.9508	0.9469	0.9430	0.9479	0.94688
S	slope_one	1.1015	1.1017	1.0978	1.1140	1.12536
S	baseline	1.1006	1.0866	1.0859	1.0933	1.10122
S	item_avg	1.0023	0.9990	0.9965	0.9997	0.99446
S	cosine	1.0746	1.0734	1.0702	1.0797	1.08452
S	pcc	1.1836	1.1780	1.1708	1.1672	1.17731
A	median	0.9199	0.9170	0.9136	0.9213	0.92005
A	average	0.9168	0.9131	0.9108	0.9175	0.91678
A	adaptive	**0.9016**	**0.9092**	**0.8929**	**0.8994**	**0.87823**

	method	*min*	*max*	*mean*	σ
S	svd1	1.0741	1.0866	1.0812	0.063377
S	svd2	1.0739	1.0902	1.0840	0.076837
S	svd3	0.9420	0.9499	0.9460	0.052801
S	svd4	0.9430	0.9508	0.9471	0.050071
S	slope_one	1.0978	1.1253	1.1081	0.101039
S	baseline	1.0859	1.1012	1.0935	0.080985
S	item_avg	0.9944	1.0023	0.9984	0.052021
S	cosine	1.0702	1.0845	1.0765	**0.0709**
S	pcc	1.1672	1.1836	1.1754	0.0758
A	median	0.9136	0.9213	0.9184	0.052478
A	average	0.9108	0.9175	0.9150	0.051037
A	adaptive	**0.8782**	**0.9092**	**0.8962**	0.102056

Table 4.4: Results from experiment 2 (Jester): The cells give RMSE values for a method. The first table gives errors for subsets of the data (d_x). Lower values indicate better results. Bold values indicate the best result in a column. S *refers to singular methods, and* A *to aggregation methods.* σ *refers to the standard deviation of each method across the subsets.*

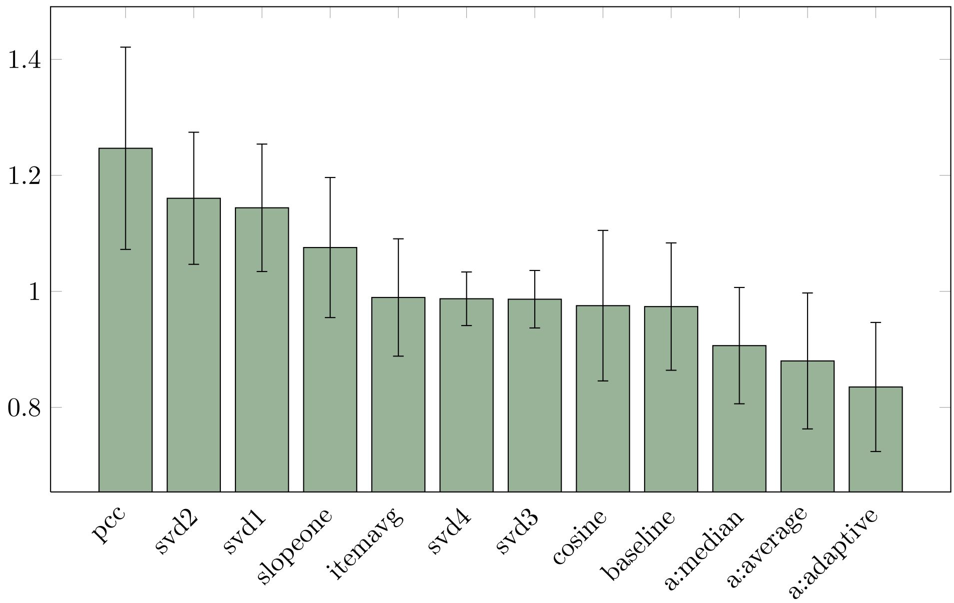


Figure 4.1: Average RMSE plot: This plot shows the average RMSE for the recommenders and the aggregators (denoted "a:"). The actual numbers are given in Table 4.3. The error bars indicate the standard deviation for the methods. Note the scale on the y-axis — the errors are not as pronounced as they might seem. See also Figure 4.2.

see, the performance of the aggregation methods, as well as the best performing standard recommender, follow each other closely. At the same time, performance varies across the different datasets, which results in high values for σ.

What does this mean for hypotheses H1 and H2? Expressed in terms of this experiment, H1 posits that adaptive recommenders should outperform the standard modeling methods in Table 4.2. The adaptive methods blend the results of multiple predictors by estimating the accuracy on a per-item and per-user basis, satisfying the formulation of H1.

By outperform we mean that our model should have a lower mean RMSE score than the other singular methods. As we can see in Tables 4.3 & 4.4, *H1 is confirmed for these methods and these datasets.* While we can not generalize too much on this basis, the fact that this dataset is a common testing ground for recommender systems, that RMSE is the de facto measure for determining performance, and because of our five-fold cross-validation, the results allow us to confirm hypothesis H1 in these conditions, and likely for other, similar scenarios. We shall discuss this result in Chapter 5.

Similarly, expressed in the same terms, H2 posits that our adaptive recommenders should outperform the aggregation approaches given in Table 4.2. The *median* and *average* aggregation methods serve as global and generalized aggregation methods. Our

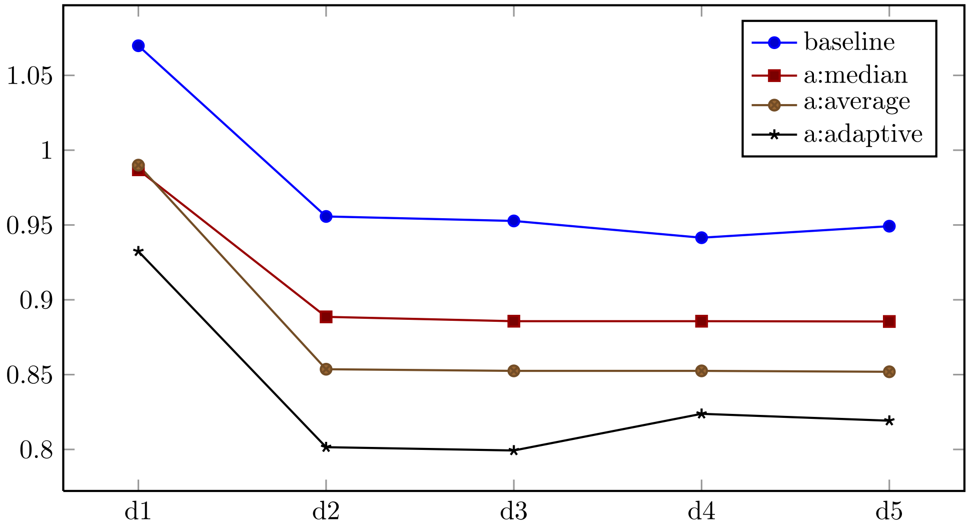


Figure 4.2: RMSE Variations: This plot shows that, while the standard deviation of each method may be high, this has more to do with the selected dataset than with their performance in comparison with each other. The performance of each of the aggregate methods, as well as the baseline standard method, follow similar performance paths across the disjoint datasets.

adaptive recommenders are adaptive in that each prediction is aggregated based on the current user and item, satisfying the language of H2.

As we can see in Tables 4.3 & 4.4, *H2 is confirmed for these methods and these datasets*. However, as our collection of aggregation methods is a lot simpler than our collection of recommender systems, the strength of this result is notably weaker than that of H1. Still, the fact that a adaptive recommender outperforms these simple aggregation approaches is a positive result warranting further experiments. This will also be discussed in Chapter 5.

Our system performs better in Experiment 1 than Experiment 2. While better performance in Experiment 2 would have been desirable, the results fit our original assumptions. The Jester dataset, used in Experiment 2, have very few items (100). This would intuitively mean that there are fewer disjoint patterns for the adaptive layer to leverage. As described in the previous chapter, adaptive recommenders are mostly mean for scenarios where we have a wide range of different users and items. However, as in Experiment 1, our method outperforms the standard and aggregate recommenders, if only by a small margin.

It would seem then that, based on our experiments, available data and assumptions of evaluation measures, both H1 and H2 are confirmed. Our adaptive aggregation approach outperforms both standard recommender methods and simple generalized aggregation methods.

Notably, our approach is more complex than the methods it outperforms, so the question whether the extra performance is worth its extra complexity becomes important. We shall discuss this, and other implications and limitations of these results in the next chapter. For now, let us proceed to the third experiment and hypothesis H3.

4.5 Rank Aggregation

Let us see how adaptive recommenders can be used for personalized search. While the previous experiment was a quantitative exploration of RMSE values, this experiment will focus on qualitative traits of rank aggregation.

The MovieLens dataset fit the needs of this experiment. Searching through movies is a scenario where the actual predicted rating of each movie could be a welcome signal for ranking results. We have a database of movies the user wishes to search through, where the results are ranked both by how well they match the free text query, and according to the predicted rating of each movie for the current user. The value of letting predicted ratings re-rank search results of movies is entirely dependent on the actual system and use case, but it presents a probable situation where personalized search may be employed.

Hypothesis H3 states that *the ordering of results from an information retrieval query can be personalized by using adaptive recommenders.* We wish to check if the prediction algorithm in Listing 4 performs personalized search in a meaningful way. There are a few important limitations to this experiment:

- We are not interested in measuring the actual performance of the IR system. It us assumed that the IR model returns items relevant to the current query, ranked by their individual relevance.
- We are not interested in measuring the performance of the resulting personalized search. This experiment will only show whether or not personalized search is achievable by using adaptive recommenders, as per our hypothesis.

This means that Experiment 3 is not statistically significant in any way, but rather a case study where we investigate the implications of using adaptive recommenders for personalized search.

To be sure, there are many ways of determining the accuracy of a personalized search algorithm. Examples include the mean average precision of the results list (the mean of the precision averaged over a set of queries). As always when dealing with personalization, these are subjective measures based on relevance judgements from each user.

Our hypothesis only states that our algorithm should be usable for such a system, which is what we shall explore in this section. To quantitatively measure the performance of personalized search, one would need detailed query logs, user profiles and click-through information.

Results and their IR ranking score for the query [*"new york" or washington*]:

#	*score*	*title*
1	2.8419	New York Cop (1996)
2	2.8419	King of New York (1990)
3	2.8419	Autumn in New York (2000)
4	2.8419	Couch in New York
5	2.4866	Escape from New York (1981)
6	2.4866	All the Vermeers in New York (1990)
7	2.1314	Home Alone 2: Lost in New York (1992)
8	1.0076	Saint of Fort Washington
9	1.0076	Washington Square (1997)
10	0.8816	Mr. Smith Goes to Washington (1939)

Predicted ratings from the adaptive recommenders method for each item:

#	*score*	*title*	Δ_{IR}
1	3.7255	Mr. Smith Goes to Washington (1939)	↑ 9
2	3.1430	Escape from New York (1981)	↑ 3
3	3.0003	King of New York (1990)	↓ 1
4	2.9498	Washington Square (1997)	↑ 5
5	2.7258	Saint of Fort Washington	↑ 3
6	2.6862	Couch in New York	↓ 2
7	2.6380	All the Vermeers in New York (1990)	↓ 1
8	2.1601	Home Alone 2: Lost in New York (1992)	↓ 1
9	1.7241	Autumn in New York (2000)	↓ 6
10	0.0	New York Cop (1996)	↓ 6

Final results list with IR and adaptive predictions combined:

#	*score*	*title*	Δ_{IR}
1	5.8422	King of New York (1990)	↑ 1
2	5.6297	Escape from New York (1981)	↑ 3
3	5.5281	Couch in New York	↑ 1
4	5.1247	All the Vermeers in New York (1990)	↑ 2
5	4.6072	Mr. Smith Goes to Washington (1939)	↑ 5
6	4.5661	Autumn in New York (2000)	↓ 3
7	4.2915	Home Alone 2: Lost in New York (1992)	=
8	3.9575	Washington Square (1997)	↑ 1
9	3.7334	Saint of Fort Washington	↓ 1
10	2.8419	New York Cop (1996)	↓ 9

Table 4.5: Complex IR query: The first table shows the results returned by our IR model, defining the item-space for the following tables. The middle table shows the predicted ratings for each of the items in the results set. Δ_{IR} shows how much each item has moved compared to the initial IR results. Notably, the recommenders were not able to predict the rating for the movie "New York Cop", which results in a low final placement for this item.

It is important to note that any recommender system can be used for personalized search. The interesting bit in regards to adaptive recommenders is what happens under the hood. First, the information retrieval score is itself treated as an input signal, just as the user modeling methods. Second, by using adaptive recommenders, the user is in control of which methods that actually determine how the final results are ranked.

We have considered four use cases to see how our algorithm performs in a number of scenarios. Each case presents a query and shows how a certain IR weight influences the final ranking.

To reiterate, the IR weight is the scalar value multiplied with the IR score before the adaptive recommender scores are incorporated in the result (see Listing 4). The actual choice of weight depends on scale of scores returned by the IR method, and how much the IR model should influence the final ranking. If the scores are on the same scales as the ratings themselves, an IR weight of 1 signifies that the IR score should have equal importance as each recommender score. Any higher, and the IR model should be prioritized above the recommenders. Any lower, and the recommender scores will dominate the initial IR rankings.

The actual IR weight must be calculated based on the scale of the IR scores. In this chapter, the scores returned by our IR model is normalized to the scale of our ratings. We then adapt the IR weight to achieve differing prioritisations of the IR scores.

We consider the following use cases: (1) searching for multiple topics, (2) searching for a series of movies, (3) searching for one particular topic, and (4) searching for a particular attribute. The first two use cases merge the IR and recommender scores into a combined ranking. The last two will let the recommender systems do the ranking in situations where the IR model is not able to properly rank the results.

(1) Searching Through Multiple topics Let us start with a simple use case. A user wishes to find movies about two separate topics, ranked by query match and predicted ratings. This is a realistic use case, for example if a user is interested in a few topics and wants to see the movie within these categories he or she will probably like the most. The IR algorithm takes care of finding the items within the categories, while the adaptive recommenders finds the most enjoyable movies, according to the metrics most preferred by this user in the past.

Table 4.5 shows this use case and how our algorithm performs. The results are for the query [*"new york" or washington*]. The first table shows the IR scores for the first 10 results, and their rank (position in the list) according to these scores. The second table shows the predicted rankings for each of these items. Finally, the third table shows the ranking after the IR scores and predicted ratings have been combined. The final column shows how each item have moved in relation to the IR results list.

In this run of the algorithm, the IR weight (w_{IR}) was set to 1.0, instructing the algorithm place about the same importance on the IR score and the predicted ratings. As we can see in the last section of Table 4.5 the final result list is a blend of the IR

Results and their IR ranking score for the query [*star trek*]:

#	*score*	*title*
1	4.2288	Star Trek: Generations
2	3.7002	Star Trek: First Contact
3	3.7002	Star Trek: The Wrath of Khan
4	3.7002	Star Trek: The Motion Picture
5	3.1716	Star Trek VI: The Undiscovered Country
6	3.1716	Star Trek III: The Search for Spock
7	3.1716	Star Trek IV: The Voyage Home
8	3.1716	Star Trek V: The Final Frontier
9	0.9670	Star Wars
10	0.9670	Lone Star

Predicted ratings from the adaptive recommender method for each item:

#	*score*	*title*	Δ_{IR}
1	4.8232	Star Wars	↑ 9
2	4.6016	Lone Star	↑ 8
3	4.2192	Star Trek: The Wrath of Khan	=
4	4.0324	Star Trek: First Contact	↓ 2
5	3.8667	Star Trek: Generations	↓ 4
6	3.7100	Star Trek IV: The Voyage Home	↑ 1
7	3.5604	Star Trek VI: The Undiscovered Country	↓ 2
8	3.4420	Star Trek: The Motion Picture	↓ 4
9	3.4242	Star Trek III: The Search for Spock	↓ 3
10	2.5249	Star Trek V: The Final Frontier	↓ 2

Final results list with IR and adaptive predictions combined:

#	*score*	*title*	Δ_{IR}
1	5.5507	Star Trek: The Wrath of Khan	↑ 2
2	5.5205	Star Trek: First Contact	=
3	5.3157	Star Trek: Generations	↓ 2
4	5.1187	Star Wars	↑ 5
5	4.9744	Star Trek IV: The Voyage Home	↑ 2
6	4.7596	Star Trek III: The Search for Spock	=
7	4.7595	Star Trek: The Motion Picture	↓ 3
8	4.7553	Star Trek VI: The Undiscovered Country	↓ 3
9	4.6376	Lone Star	↑ 1
10	4.0934	Star Trek V: The Final Frontier	↓ 2

*Table 4.6: These three table show adaptive rank re-scoring for the query [*star trek*]. In this example, an IR weight of* 0.3 *was used, instructing the algorithm to put about the same confidence in the IR score and recommender scores. Each score is considered an input signal, and each signal is weighted the same.*

rankings and prediction rankings. In other words, we have achieved personalized search. The results from the IR method are re-ranked according to personal preferences.

(2) Searching for Series of Items Let us consider another use case. A user wishes to see a movie in a certain series of movies, but does not know which one. In this case, the IR method can find all movies within this series, while the recommender systems ranks the result list according to the user's preferences.

Table 4.6 shows the intermediary and final rankings for the query [*star trek*], which refers to a collection of movies within the same series. The IR method returns all items that match this query, and the recommenders predict the rating for each of these items. However, since the IR method only ranks results based on how well they match the query, and the recommenders only care about the predicted rating, the combined result list can get the best of both worlds: the top ranked items are the ones that both match the query *and* are probable good fits for the current user.

(3) Searching in a Singular Topic What happens when the IR weight is set to 0? In this use case, the predicted ratings alone sort the final list. Consider the following use case. A user wishes to see a movie related to a certain topic, e.g. a city. Table 4.7 shows two results lists for the query [*paris*]. On the left are the standard ranking as returned by the IR model for this query, along with their respective scores. On the right we see the same results, re-ranked by user preferences.

For simple one-word queries, ignoring the IR score seems to give us the desired effect. If we can be sure that all items returned by a search have the same textual relevance (IR score), the IR method does not have any more information on which to rank the results. The ranking then becomes the task of the recommender systems. By employing adaptive recommenders, the results are not only ranked by one or more recommenders as chosen by the system, but by those of the recommenders best suited to the current user. At the same time, each of these recommenders are used differently for items in the list, based on how well they have previously performed in this context.

(4) Attribute-Based Search As we can see, ignoring the IR score gives us quite a different algorithm. Now, the search part is only performed to constrain the item-space worked on by the recommender systems. Another example of this is shown in Table 4.8. In this scenario, the user wishes to see a movie from a certain year, and issues the query [*1998*]. Naturally, the IR algorithm returns a whole lot of items, and each movie can be said to be perfect answers to the query, as they were all made in 1998.

In this case, setting the IR weight to 0 allows us to rank the results purely by predicted preference, which makes sense when the IR algorithm can not rank the results in any meaningful way. Note that the items in the left and right table are non-overlapping. This is because only the first 10 results are shown. The IR model returns a large number

#	score	title	#	rating	title	Δ
1	3.0149	An American in Paris	1	3.5277	An American in Paris	=
2	3.0149	Paris Is Burning	2	3.3416	Forget Paris	↑ 3
3	3.0149	Paris - Texas	3	3.2037	Paris - Texas	=
4	3.0149	Paris Was a Woman	4	3.1870	Window to Paris	↑ 2
5	3.0149	Forget Paris	5	3.1409	Paris Is Burning	↓ 3
6	3.0149	Window to Paris	6	3.1059	Last Time I Saw Paris	↑ 4
7	3.0149	Jefferson in Paris	7	2.7940	Rendezvous in Paris	↑ 2
8	3.0149	Paris - France	8	2.2964	Paris - France	=
9	2.6648	Rendezvous in Paris	9	1.7984	Jefferson in Paris	↓ 2
10	2.2611	Last Time I Saw Paris	10	0.9420	Paris Was a Woman	↓ 6

Table 4.7: Completely adaptive ranking: With the IR weight set to 0*, the adaptive recommender is alone responsible for sorting the results. In this example, the IR model returns a list of items for the query [*paris*], and the adaptive user models sorts the results according to the user's preferences. The top 10 results are shown.*

#	score	title	#	rating	title
1	2.7742	Fallen (1998)	1	3.8694	Apt Pupil (1998)
2	2.7742	Sphere (1998)	2	3.4805	The Wedding Singer (1998)
3	2.7742	Phantoms (1998)	3	3.1314	Fallen (1998)
4	2.7742	Vermin (1998)	4	3.1225	Tainted (1998)
5	2.7742	Twilight (1998)	5	2.9442	Blues Brothers 2000 (1998)
6	2.7742	Firestorm (1998)	6	2.9046	Sphere (1998)
7	2.7742	Palmetto (1998)	7	2.8842	Desperate Measures (1998)
8	2.7742	The Mighty (1998)	8	2.8798	Firestorm (1998)
9	2.7742	Senseless (1998)	9	2.8633	Vermin (1998)
10	2.7742	Everest (1998)	10	2.8511	The Prophecy II (1998)

*Table 4.8: Ranking many results: In this example, the user search for the query [*1998*], to get movies from that year. The top 10 of these are shown in the left table. As this query matches a lot of movies, the IR method returns a large number of results. By setting the IR weight to* 0*, and letting the adaptive recommenders do the ranking, the top 10 results change completely, while still being good matches for the current query.*

of items, all with the same ranking score. The recommender systems do the final ranking, and actually push every item in the top 10 IR ranking below the top 10 final results.

4.5.1 Adaptive IR Weights

As we have seen in this section, adaptive recommenders can provide personalized search in multiple ways. By varying the IR weight we can create quite a range of systems. On the one hand, an IR weight of 0 will let the recommenders do all the ranking. On the other hand, by increasing the IR weight, the recommenders will carefully adapt parts of the IR results list by moving some of the items.

We have not considered which IR weight or other parameters would result in the best performing personalized search system. However, this is completely dependent on the type of system and types of queries. By varying the IR weight, a number of different systems that work for different use cases can be constructed. For systems with simple one-word queries, setting the weight to 0 leaves ranking to the recommenders. For systems with more complex queries, an IR weight of 1.0 orders items both by IR score and predicted rating. This weight is then the defining characteristic of any personalized search based on adaptive recommenders.

An adaptive adjustment of the IR weight based on the current query and use case would seem to be the best choice for a system that should handle every scenario. When we have a short, specific query, the IR ranking function have little to no basis for ranking the items differently. In this case, a weight of 0 allows the recommenders to perform the ranking of a constrained item space.

In systems with complex queries, where the IR model *can* rank items based on their similarity to the information need, an IR weight of 1.0 gives us the desired result. Items are ranked both by their match to the query, and based on how users and items should be modeled. The most interesting bit happens under the hood By using adaptive recommenders, the re-scoring functions are conditioned on how well they have previously worked for the current item, and how well they suit individual users.

The performance of personalized search is hard to judge without extensive query logs with click-through information. While we had no access to such data, we have been able to show that *adaptive recommenders can be used to provide personalized search.* This positive result for Experiment 3 confirms hypothesis H3, at least for this dataset, this IR system and our chosen recommender algorithms. By confirming H3, we have shown that adaptive recommenders can be used for personalized search.

This results in a search engine results page that inherits the strengths of adaptive recommenders. The items on the result list is ranked not just based on query matching, but based on a number of signals, represented by recommender systems. The signals are adaptively used based on how well it suits the current user, and how well it has worked in the past for individual items.

By creating an adaptive results page, we help mitigate the latent subjectivity problem, by ranking each element based on the current context. We will discuss this further in the next chapter.

— 5 —

Discussion & Conclusion

This chapter will discuss the implications and limitations of our results. While our hypotheses may be confirmed, it is important to clarify what we have actually found out, and what limits there is to this knowledge. We will also summarise the contributions of this thesis, and suggest possible future work.

5.1 Implications & Limitations

Let us first discuss some general implications and limitations of our approach.

Our central assumption is that modern recommender systems are constrained by their misplaced subjectivity. Each system selects some measures to model its users, based on how they think users and items *should* be modeled. We believe this selection should be left to individual users. Different users and items will require adaptive recommender algorithms that consider the context before making predictions.

Adaptive recommenders can help solve this problem. In a collection of possible recommender algorithms, each is adaptively used based on how well it performs for the item and user in question. The experiments of the previous chapter shows the promise of this technique. At the same time, there are lots of use cases not yet considered.

It should be clear that adaptive recommenders would work best in situations where we have a wide range of diverse algorithms that can infer the relevance of an item to a user. For users, social connections is a good example. Whether or not social connections should influence recommendations or personalized search results is a contentious topic. Naturally, a system where every user's personal opinion determines if these connections are used is desirable.

This implication extends to the items that should be recommended. As evident by the field of information retrieval, there exists many ways of considering the relevance of an item. These algorithms can be based on a number of attributes, for example temporal information, geography, sentiment analysis, topic or keywords. It is not a huge leap to assume that these algorithms may have varying levels of accuracy for individual items. Adaptive recommenders can help solve this problem by adaptively combining the recommenders based on individual item performance.

The use of error models has an important implication. With adaptive recommenders, both the methods layer and the adaptive layer consists of standard recommender algorithms. Because we use ratings matrices for the taste models and error matrices for the weight estimations, we can use the same algorithms for both tasks. Using known algorithms for this new task is beneficial. They are known to work, enjoy multiple implementations and are already understood and battle-tested in many different systems.

There are some important general limitations to our research related to the complexity of our method, our choice of data and evaluation metrics, and common issues with recommender systems.

5.1.1 The Complexity of Adaptive Recommenders

As our approach is more complicated than standard recommenders, it is worth questioning if its gains are worth the extra complexity. This depends on the basic recommenders that are to be combined. If the system is made up by many different recommenders, that users might place varying importance on, and that may have varying success with individual items, adaptive recommenders may provide gains in accuracy.

On the other hand, if the recommenders are simple in nature, and look at similar patterns in the data, generalized aggregation methods might be more applicable. Clearly, the performance gains in our experiments are not substantial enough to declare anything without reservation. While we believe this technique has potential, without real-world success stories, it is hard to suggest that our method is particularly better than a simple standard recommender.

When considering the additional complexity of our approach, a natural response is whether or not current approaches to recommender systems are good enough. We do not think so. Information overload is such a nuanced problem that the only solution lines in intelligent, adaptive systems. However, as most of today's recommender systems perform quite simple tasks, they may be more than good enough for their purpose.

This will always be a trade-off, between complexity and required accuracy. As in many other scenarios, the systems described in this thesis have their use cases. In the end, the requirements of the current system must decide which method best suit their needs.

5.1.2 Evaluation Measures

To evaluate our approach, we chose datasets and evaluation measures that are commonly used to test recommender systems. By choosing traditional measures, our experiments can be compared to other research in the same field.

While our initial results are promising, it is important to stress the fact that only a few datasets were used in our testing. Because of the vast scope of users and items such a system might use, any real world application would need to test each approach with their actual data. Our results must be seen for what they are: initial and preliminary explorations of a new technique that has only been proven useful in a few use cases.

As mentioned in Chapter 2, the scale of known data points is an important concern. When we have a set of explicit ratings given by users, these are often given in discrete steps, and not on a continuous scale. As known from the field of statistics, when using ordinal scales, the significance of every step are not necessarily equal.

For instance, on a scale from 1 through 5, the difference between 2 and 3 might not be as significant as the difference between 4 and 5. This is a limitation of many recommender system, apparent by the algorithms they use. Most do not consider the implications of ordinal data. Naturally, in a real-world system, this limitation has to be considered.

5.1.3 Common Limitations of Recommender Systems

The topic of recommenders and adaptive systems in general raise a number of questions which is outside the scope of this thesis. For example, user privacy is a big issue. Whenever we have a system that tries to learn the tastes, habits and traits of its users, how they will react to this must be considered. This is often a trade-off between adaptability and transparency. The most adaptive systems will not always be able to explain to the users what is going on and what it knows about each person, especially when dealing with emergent behavior based on numerical user models.

Another important issue is the usability of autonomous interfaces. Whenever recommenders are used for more than simple lists of items, there is a question of how easy the resulting system will be to use. As mentioned in Chapter 2, unpredictability is the enemy of usability. Creating an autonomous system that is also predictable is a serious challenge, and a common trade-off.

The question of whether or not recommenders systems really will be able to curb information overload is another important discussion. The problem is especially apparent when considering information such as news articles. An RS will attempt to rank news articles based on previously read news, while a user might be as interested in other articles of an unknown nature. If the personalization of information is too comprehensive, information on seldom viewed topics may be wrongly deemed irrelevant. This is antithetical to the main purpose of an RS: discovery of unknown relevant content.

This problem is also apparent when the information at hand contains opinions and debates. An RS might strive to filter out information that the user does not "like", i.e. information that is contrary to the user's own views or opinions. Clearly, this would not be desirable. It should not be the goal of an RS to shield users from differing opinions, or constrain the range of information the user is exposed to. When considering employing an RS, these questions must be taken into account, or one might end up with a system that works against the wishes of each user.

While a thorough discussion of privacy, usability and information scope is outside the purview of this thesis, they are all important limitations to considered when using a recommender system.

Let us now take a look at some important implications and limitations of the experiments from Chapter 4.

5.2 Prediction Aggregation

Hypothesis H1 was confirmed by showing that adaptive recommenders can outperform standard single-approach recommenders. We achieved lower total RMSE scores across our datasets, which would imply that adaptive recommenders reliably performs better than our tested standard recommenders, at least with these datasets and this error measure. This is the most basic test that could be done to evaluate the performance of adaptive recommenders. The real test would be to use our approach in a situation with even more differing recommender systems.

Crucially, H1 was only tested against a limited number of standard recommenders. The key word here is *standard*. These recommenders were not heavily customized to fit the available data. As in much of machine learning, achieving relatively good performance is quite simple. Any improvements above this standard requires deep domain knowledge, and methods customized to the problem at hand. In an actual system, the adaptive recommenders should be tested against carefully selected standard recommenders, optimized for the current domain.

Hypothesis H2 was confirmed by showing that adaptive recommenders can outperform simple, generalized aggregation approaches. While our aggregators were simple, this result is promising. It remains to test our method against more complex generalized aggregation functions. However, the main point of adaptive recommenders should still hold against complex approaches. For example, a complex weight estimation to achieve an optimal combination is still a generalized result, averaged across all users. Whenever we have a situation where users and items will prioritise the available recommenders differently, adaptive recommenders should be able to provide this extra level of personalization.

It is important to note that H2 was only tested against simple aggregators. Many more complex aggregations are possible. While our tests show the basic viability of our approach, more testing against complex aggregation functions is still required. For example, adaptive recommenders should be tested against more complex weight estimation functions, that solve the problem by computing optimal and generalized weights.

However, the main strength of adaptive recommenders should still remain. Each of the modern approaches to recommender aggregation are generalized and averaged combinations. Adding another layer of personalization should result in better matches between how elements are modeled and how they wish to be modeled, no matter how complex the comparative averaged aggregation may be.

Both H1 and H2 was only tested with two datasets, and with a single error measure. While its true that these datasets and this error measure are the canonical ways of estimating the performance of recommender systems, more research is required to further verify these results. In particular, both datasets exhibit quite homogeneous types of items, while our approach may have different characteristics in scenarios with widely differing items and users.

Experiment 1 gave better performance results than Experiment 2. The differing variable was the dataset used in each experiment. The main difference between the datasets was that in experiment 1, we had many more items than in Experiment 2. However, given that the datasets contain different kinds of items, we can not generalize on this basis alone. Clearly, our method will perform differently depending on the data in question, as one would expect.

While our experiments with prediction aggregation were simple, they show the potential of adaptive recommenders. As explained in Chapter 1, the mismatch between how users and items *should* be modeled and how each system actually does represent elements hinders the full adoption of recommender systems for creating truly adaptive content, i.e. systems that adapt all their content based on the current user and items. By adaptively combining multiple prediction algorithms, this latent subjectivity can be overcome. However, much research still remains, as we shall describe in Section 5.4.

5.3 Rank Aggregation

Hypothesis H3 was confirmed by showing that our approach can be used to provide personalized search. While we did not evaluate the quantitative performance of this approach, our results show that adaptive recommenders can provide many types of personalized search. The different kinds of systems were created by varying the importance of the scores returned by our information retrieval system. The key insight is that the IR score can be seen as a signal on the same level as the adaptive recommenders, gaining the power of query matching and relevance matching in the same results set.

There are many ways in which a list of search results can be personalized, and users will have individual preferences as to how they should be modeled. At the same time, different recommenders will have varying performance across the many types of items such a system might encounter. Adaptive recommenders may allow personalized search systems to get even more adaptive, by customizing their internal workings to each user and item.

Crucially, H3 was only tested in a qualitative way. Ideally, if one has access to detailed query logs, user profiles and click-through information, a quantitative experiment should be performed. Such an experiment would have to be done to compare our approach to other ways of performing personalized search. However, we believe these initial results help demonstrate the probable value of our approach in this domain.

This is the core limitation of Experiment 3. Before employing personalized search with adaptive recommenders, this technique should be evaluated towards the result of other personalized search algorithms. This will in large part entail setting the IR weight of our algorithm, which decides how the resulting ranking functions should sort search results. Despite its limitations, our experiment shows that adaptive recommenders can be used to provide personalized search.

We have only touched the surface of whats possible when using recommender systems together with information retrieval systems. While it is true that search and recommendations are widely different use cases, a combination can get the best of both worlds. The IR system can constrain the universe of items based on the actual current information need, and the RS can estimate how well each element fit the current user.

Although not attempted in this thesis, by using multiple IR models, adaptive recommenders can be used to automatically estimate the accuracy of each IR model for each user and item. This would allow for truly adaptive information retrieval. We have not performed experiments with such a system in this thesis, but it is a logical extension of the personalized search system presented in the previous chapter. We will discuss this further in the next section.

5.4 Future Work

We have only shown the basic viability of adaptive recommenders, and how they can outperform traditional approaches on traditional datasets. This section outlines five interesting research topics which would shed more light on the subject.

(1) Quantitative Performance of Personalized Search We did not test how well adaptive recommenders would work for personalized search. Our third experiment was a case study, detailing how this might be done. With more data or test subjects, it would be possible to measure actual performance gains (or setbacks) by using adaptive recommenders to achieve personalized search.

To measure the performance of personalized search one would need detailed query logs with click-through information. By this we mean logs that show the queries from individual users, and which search result they selected for every query. These logs can be mined to create implicit ratings matrices, which can be split into training- and testing sets. However, as this is outside our scope, and we lack the necessary data, we leave this experiment to future research.

(2) Choosing Different Adaptive Recommenders We chose to use SVD-based recommenders for the adaptive part of our adaptive approach. The main reason for this is that we are looking for global traits of the data when performing accuracy estimations. We wish to identify clusters of users and items for which the algorithms may or may not be suited.

As the adaptive recommenders can utilize any standard recommender system to model the errors of another recommender, it would be interesting to perform a more in-depth study of how different choices for the adaptive layer influence the final system. There are many more recommenders that also look at global patterns that might be well suited for this task. Another interesting question is whether other machine learning methods can be used for the adaptive layer. For example, using neural networks to estimate non linear aggregation functions for individual users would be an interesting approach. This was attempted earlier in our research, but abandoned when recommenders were found to produce better results in a more elegant way.

(3) Using Adaptive Recommenders in Other Domains We chose to use the MovieLens dataset and the RMSE evaluation measure for testing our approach. The primary reason was to be able to directly evaluate our results towards those of other research thesis. As this dataset and this error measure is widely used to evaluate recommender systems, it is natural for a first look at a new approach to use the same notions of accuracy.

As mentioned above, the main strength of adaptive recommenders may be in situations with much more diverse data sources. Social networks or systems with widely varying sets of items would provide an interesting use case for adaptive recommenders. The main premise of our approach is that users and items have differing preferences for each algorithm. Naturally, the more diverse the data and algorithms get, the more dire the need for adaptive aggregation becomes. Because of this, using adaptive recommenders in other domains with more variation in the data and combined algorithms would be an interesting topic.

(4) Multiple IR Models as Signals As mentioned in Chapter 4, we only tried rank aggregation in a scenario with one IR model. Other systems may use multiple IR models that return a set of ranked items in response to a query. In the case of personalized search with multiple IR models and RSs, we would have a large set of differing input signals: one from each IR model and one from each RS.

In this case, adaptive recommenders could be used to combine both the RSs and IR models. In the same way different RSs have varying performance for individual users and items, the same should hold for different IR models. By using adaptive recommenders we would be able to adaptively restrict the item space based on the current user. While outside the scope of this thesis, using multiple IR models would add another adaptive aspect to the final results list in personalized search.

(5) Using Adaptive Recommenders in Other AI Fields We have only considered the notion of latent subjectivity within the field of recommender systems. However, as briefly mentioned above, the technique should be applicable to many more situations. Whenever there is a set of prediction algorithms that use different data to produce results, an adaptive aggregation should be able to combine these in a more nuanced way.

Ensemble learning is a big topic, used in many situations. By layering recommenders on top of the methods in an ensemble, we get a system capable of predicting the accuracy of the basic methods. Naturally, it would be interesting to see how this approach would fare in other fields such as document classification, document clustering, curve fitting (Polikar, 2006, p.7), and other fields of ensemble learning.

5.5 Conclusion

We have made two main contributions with this thesis: (1) described the latent subjectivity problem and (2) developed the technique of adaptive recommenders.

(1) The latent subjectivity problem is an issue we think hinders standard recommender systems reaching their full potential. As far as we know, this problem has not been described in the context of recommender systems. The main choice for any such system is how to predict unknown ratings. To do this, a pattern in the available ratings data must be leveraged. These patterns are plentiful, and their individual performance depends on the users and items of the system. Modern aggregation recommenders utilize many patterns, but on a generalized level, where each user and item is treated the same. This underlying subjectivity leads to a mismatch between the notions of whoever developed the systems, and the users and items of the service.

The latent subjectivity problem extends to any ensemble learning system (as those described in Polikar (2006)) that blends multiple algorithms to leverage patterns. Whenever we have multiple algorithms that work on a set of items (and possibly users), there is a question of how accurate each approach will be for any individual item. Averaged or generalized weighted approaches will always chose the combination that performs best *on average*, with little concern to the uniqueness of items (and users). This is a comprehensive problem that may be discovered amongst many machine learning techniques.

(2) Adaptive recommenders is our attempt to solve the latent subjectivity problem. As far as we know, this type of adaptive prediction aggregation has not been done before. Chapter 4 showed that an aggregation that combines predictions based on estimated accuracy can outperform both standard recommenders and simple aggregation approaches. Our technique is strengthened by the fact that standard recommender algorithms are used for the accuracy estimations. This is the core insight of this thesis. *We can use standard recommender systems to create error models for other recommender systems, that can be used to estimate the accuracy of each system for all possible relevance predictions.*

As far as the latent subjectivity problem extends to any ensemble learning system, the adaptive aggregation part of adaptive recommenders can be used to create better combinations of many types of predictors. Whenever we have a set of algorithms producing a set of predicted values based on items, a set of aggregating recommenders can model the probable errors of these approaches, based on individual items. This leads to adaptive ensembles that should outperform generalized approaches. Because of this, the technique build in this thesis should be applicable in situations other than recommender systems.

While the experiments of Chapter 4 show the general viability of adaptive recommenders, we believe there are greater opportunities in systems where there are even more diverging patterns to be leveraged. The prime examples of this are systems that may or may not use social connections between users, and systems which predict the relevance of widely varying items.

We have only tested our method in a limited number of use cases, with a few specific datasets. This is an important limitation. Until a method is successfully applied in a real world situation, claiming progress is premature. However, we believe more research into internally adaptive recommender systems would be a worthwhile effort.

On a more general note, we think our notion of adaptive model aggregation is key to stopping information overload, regardless of how it is done. Generalized methods is not enough. To curb the problem, systems must be able to adapt their internal algorithms based on a wide variety of users and items.

The information overload problem will always be present. No matter how elegant solutions one may find, the fact is that the overwhelming amount of available data quickly outgrows our ability to use it. We believe artificial intelligence is crucial to finding a solution. Only by creating intelligent systems that help us filter, sort and consume information can we hope to mitigate the overload.

Adaptive recommender show how applications can adapt their internal algorithms based on each user and item. As we have shown in this thesis, this extra layer of personalization leads to a better match between how users should be modeled and how the system actually performs this modeling. Applications should not only predict relevance of information items to users, but also allow flexible and adaptive usage of their algorithms. *After all, a system that insists on being adaptive in one particular way is not really adaptive at all.*

— A —
Implementation

This appendix describes how we implemented adaptive recommenders. This is a short description of the most important features and considerations made when implementing the system. While quite specific and not important to the viability of the technique in itself, this should give a short introduction to how it can be put into practice.

A.1 Libraries

The most important part of the implementations are the recommender systems. These are used for the basic ratings predictions, and to create the adaptive aggregation by predicting the accuracy of other recommenders. At the same time, these different recommenders need to have the same interface for training and testing, regardless of which context the experiments places them into. Our implementation makes use of a number of external libraries, as seen in Figure A.1.

To quickly get a large number of recommenders up and running, the system was linked with the *Apache Mahout machine learning library* (See Appendix B). Mahout provides a number of machine learning algorithms, amongst which a set of recommender systems. Examples include SVD- and neighbor-based recommenders, baseline recommenders, a Slope One recommender, cluster-based recommenders, and various generic recommenders for mixing different similarity and neighborhood measures. Mahout is a young project, launched in 2008, but was found to be quite mature and feature-rich in our experience.

Mahout is build on top of *Apache Hadoop*, a system for creating scalable and distributed data processing systems (See Appendix B). This is important to the performance of our system. As mentioned, a lot of the operations performed in layering recommenders are independent and lend themselves well to parallelization. By building on Hadoop, the recommenders are already implemented in a proper MapReduce framework for parallel computation (as explained in Manning et al. (2008, p75)). Each of the basic recommenders and adaptive aggregators can then be modeled at the same time, making the most out of whatever hardware is present.

For our IR tasks, we chose to build on another library. *Apache Lucene* (See Appendix B) is an open-source search engine, also built on top of Hadoop, gaining the same performance wins as Mahout. Lucene provides powerful methods for creating indexes of items, and for querying these indexes.

Mahout, Lucene and Hadoop are all written in the Java Programming Language, and runs on the Java Virtual Machine (JVM). To facilitate rapid prototyping, the Ruby scripting language was chosen as a "glue" language, for interfacing with the libraries. By using the JRuby implementation of Ruby, Java libraries can be imported directly into

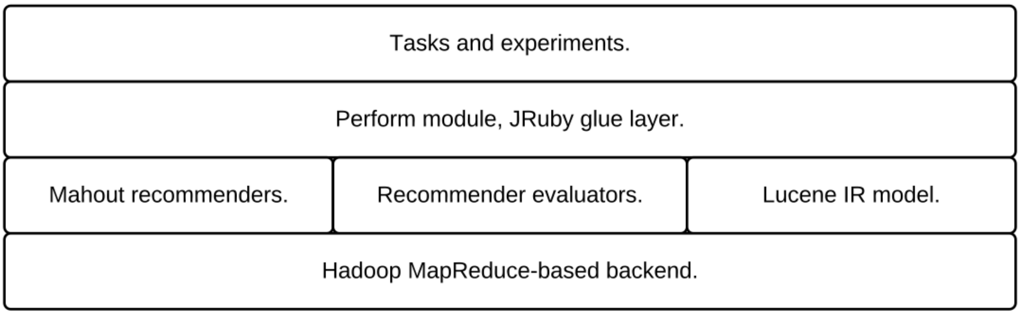


Figure A.1: Library layers: Tasks and experiments are performed by the custom JRuby glue layer. Recommenders are created by the Mahout machine learning library. IR models are based on the Lucene search engine library. Recommender evaluators are created in Ruby. Each of the intermediary layers are built on the Hadoop MapReduce framework for efficient parallel computation.

the language, allowing us to use Mahout and Hadoop almost as if they were written in the same language. The use of Ruby allowed us to quickly develop different combinations of recommenders and perform varying experiments in a short amount of time.

A.2 Task Structure

Our system is built around a few core concepts that can be used together in different ways. Everything the system does is considered a *task*. A task is a collection of settings and directives that serves as an instantiated configuration of the system. Tasks are created beforehand, and fed into the system, which carries them out. Tasks specify what the system should do, which dataset should be used, and other options. See Figure A.2 for the overall structure.

The most important task is creating a recommender. As recommenders are used both for the standard rating predictions, and for the adaptive error estimations, creating recommenders are the most common and important task of this system. Another important task is creating evaluators. An evaluator takes a set of recommenders as input, tests them against the dataset specified in the task, and returns the results of the evaluation.

A.3 Modeling & Prediction

The modeling phase consists of running our modeling algorithms and storing the resulting models. A task is created for each of the basic recommenders, and for each of the adaptive recommenders. If this is a rank aggregation scenario, an IR model is also created, based on the data specified by the current task. As mentioned, this is an offline approach, so

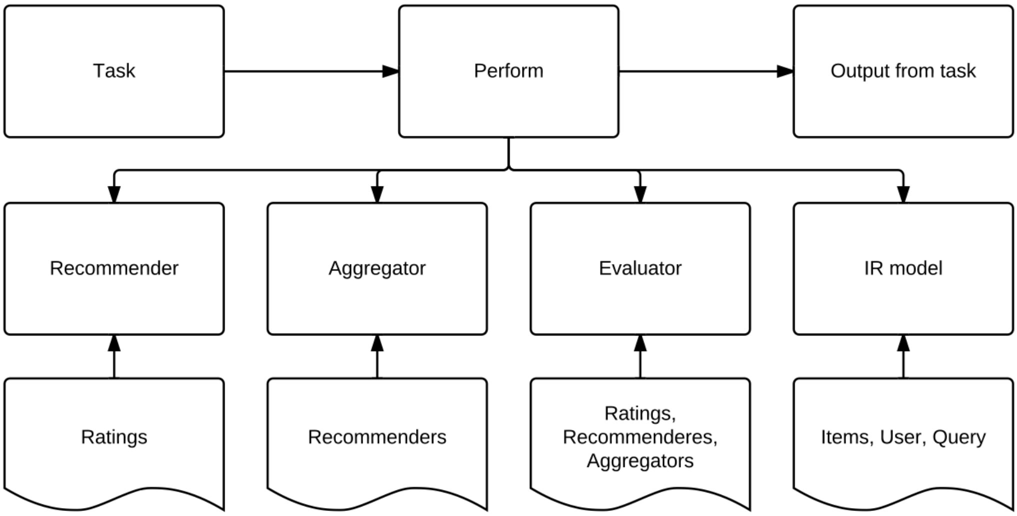


Figure A.2: Task structure diagram: A task (instantiated configuration) is passed to the perform module. This module creates a number of modules: recommenders, aggregators, evaluators and information retrieval models. Each module takes a set of inputs (bottom row), which are specified by the current task. These modules are then used as needed by the experiments.

that the models can be computed and recomputed, independent of making any actual predictions.

Our experiments required us to measure the performance of the basic recommenders, and the adaptive recommender, for every combination of a user and an item. In order to perform these experiments, an *evaluator* module was built. As both the standard recommenders and the adaptive recommender system presents the same interface, the evaluators simply takes a set of recommenders as input, and measures their accuracy across the dataset specified by the current task.

This prediction phase, where each user is compared to every unrated item, is not comparable to the prediction phase of a real-world system based on adaptive recommenders. In a real world application of this technique, a prediction is made whenever a user's actions requires it, e.g. when we need to know what a user will think of an item.

This is where the MapReduce operations previously mentioned come into play. Each of the basic recommenders and adaptive recommenders can be applied in parallel. The basic recommenders are applied through a map operation, where the current user and item (the input) is given to the modeling methods. These methods return a number of scalar predictions. The next step is the reduce operation, which is the adaptive layer. Here, the scalar predictions are reduced to one prediction by computing weights based on probable accuracy. These computations can for example be cached, if certain combinations of users and items often need predictions.

As mentioned, none of these aspects have any bearing on the viability of adaptive recommenders. However, as this does provide an example of how to implement such a system. See Appendix B for links to other resources.

A.4 Example Task

This section gives an example of an experiment run through our system. In this experiment, we wish to create our adaptive recommender and test it on the MovieLens dataset. The task specifications are written in JRuby, just as in the implementation.

First, we create our tasks and run these tasks to create the recommenders. We then create an evaluator to test our resulting adaptive recommender. The resulting RMSE values are output to the screen (see Listing A.1).

```
d_m = "movielens/base/1"
d_t = "movielens/test/1"

# Standard recommenders
recommender_tasks = {
  pcc:        Task.new(recommender: :generic_user, dataset: d_m),
  item_avg:   Task.new(recommender: :item_average, dataset: d_m),
  slope_one:  Task.new(recommender: :slope_one, dataset: d_m),
  baseline:   Task.new(recommender: :item_user_average, dataset: d_m),
  cosine:     Task.new(recommender: :generic_item, dataset: d_m)
}
rs = Perform.perform(recommender_tasks)

# Aggregate recommenders
aggregate_tasks = {
  average: Task.new(recommender: :aggregate, method: :average, recommenders: rs),
  median:  Task.new(recommender: :aggregate, method: :median,  recommenders: rs)
}
aggregate_recommenders = Perform.perform(aggregate_tasks)

# Adaptive recommender
adaptive_task = Task.new(recommender: :adaptive, recommenders: rs)
adaptive_recommender = Perform.perform(adaptive_task)

# Merge all recommenders
all = rs.merge(aggregate_recommenders).merge(adaptive_recommender)

# Evaluation
evaluator_task = Task.new(mission: :rmse_evaluator, recommenders: all, dataset: d_t)
evaluator = Perform.perform(evaluator_task)

# Run experiment
result = evaluator.evaluate
Log.evaluation(result)
```

Listing A.1: Example code showing a test of adaptive recommenders.

Each of our experiments were run with a similar task structure. The specifics of our experiments can be seen in the implementation code, links to which are given in Appendix B.

A.5 Running the Experiments

To run the experiments, a few libraries must be installed. First, JRuby must be available (see links in Appendix B). This has to be a version capable of running code conforming to Ruby version 1.9 (e.g. JRuby 1.6). The *rake* build library should also be installed, as it is used to start and run the experiments.

To get the system up and running in a POSIX-based environment, the following steps should be run. The $ refers to the prompt in a terminal window. All commands should be run from the top-level folder of the implementation source code.

1. Install JRuby (version >= 1.6) from their website.
2. Install rake: `$ jruby -S gem install rake`
3. Run the first experiment: `jruby -S rake e1`
4. Substitute `e1` with `e2` or `e3` to run each experiment.

On some systems, the JRuby VM might run out of memory due to its low default setting. To manually allow more memory to be used, the following command can be substituted in when running each experiment:

```
jruby --1.9 -J-Xmn512m -J-Xms2048m -J-Xmx2048m -S rake e1
```

The first parameter specifies that this application should be run with version 1.9 of the Ruby language specification. The second specifies the minimum garbage collection memory size. The two following gives the minimum and maximum heap memory size. The last part of this command is the file that should be run. Each experiment has its own file.

Due to the different dataset sizes, Experiment 1 takes little time to run compared to Experiments 2 or 3. Neither experiment is optimized for speed, but the comparably limited dataset of the first experiment makes it a lot faster than the others.

A.6 Customizing the Experiments

Each experiment can be customized and adapted in a number of ways. Experiments 1 through 3 can be found in three separate files in the top-level code directory.

The set of recommenders and aggregators, and their individual settings are the same in every experiment. These settings are customized in the `code/experiments.rb` file. The settings not given in this file take the default values for each setting, as given in the `code/lib/ar/task.rb` file.

The datasets used in this thesis is bundled together with the implementation code, and can be found in the `code/data` folder. Experiment 1 use the standard recommenders, aggregate recommenders, and the adaptive recommender on the MovieLens dataset. Experiment 2 runs the same recommenders on the Jester dataset.

Experiment 3 is a bit different, as it uses the Lucene IR model on the MovieLens dataset to create an IR system for searching through movie titles. As this experiment is a case study, it does not test every combination of users and items, but rather some queries for some users. The actual query and user that are considered when running the experiment are set in the file code/experiment3.rb. See the comments in this file for more information.

The file structure for the most important files of the implementation is given in Listing A.2. This listing also gives a short description of each file.

```
code/
  experiments.rb     # common settings for all experiments
  experiment1.rb     # prediction aggregation with movielens
  experiment2.rb     # prediction aggregation with jester
  experiment3.rb     # rank aggregation with movielens
  Rakefile           # used to run each experiment with jruby

  data/
    ir/              # contains the IR models for Experiment 3
    movielens/       # raw data for the movielens dataset
    jester/          # raw data for the jester dataset
    tmp/             # tmp files created by the adaptive recommender

  lib/
    ar.rb            # main library file
    ar/
      ext/           # core extensions for the ruby language
      lucene/        # the lucene IR library
      mahout/        # the mahout machine learning library

      adaptive.rb    # the adaptive recommender
      aggregate.rb   # the aggregate recommenders
      exceptions.rb  # custom exceptions for our library
      log.rb         # custom logging to stdout
      lucene.rb      # integration with the lucene library
      mahout.rb      # integration with the mahout library
      perform.rb     # executes task objects
      rank_eval.rb   # evaluation of rank aggregation
      ranker.rb      # adaptive rank aggregation algorithm
      recommender.rb # creates all types of recommenders
      rmse_eval.rb   # evaluation of prediction aggregation
      scale.rb       # utilities for scaling rating values
      task.rb        # default settings for tasks
```

Listing A.2: Files and folders of the implementation.

See Appendix B for links to the implementation and other resources.

— B —

Resources

This appendix gives pointers to additional resources mentioned throughout this thesis.

Implementation Code The code for the implementation outlined in Appendix A is available online. It resides in version control at github.com/olav/thesis/tree/master/code.

The implementation is built on three open-source libraries from the Apache Project[1]. The Hadoop distributed computing library[2], the Mahout machine learning library[3], and the Lucene information retrieval library[4].

Specific versions of these libraries are bundled together with the source code as JAR-files, that run on the JVM. Note that these libraries are released under their own terms, namely the Apache License[5]. The repository also includes the glue-code, written in ruby and run on the JRuby[6] interpreter.

Previous Work Parts of this thesis is based on a previous work in the same field: Bjørkøy (2010). This report is available from github.com/olav/papers/raw/master/user.modeling.on.the.web.pdf. A short version of this thesis is also available: github.com/olav/thesis/raw/master/paper/dist/paper.pdf

Document Details This thesis is written in the LaTeX document preparation system. It is based on a LaTeX-template called Memoir[7]. Most of the figures and graphs are made with the TikZ and PGF graphics libraries[8].

The entire source code for this document can be found at github.com/olav/thesis/tree/master/thesis. The most current PDF-version is also available from this site. For citation purposes, use the following BibTex entry:

```
@mastersthesis{Bjørkøy2011,
  address = {Trondheim, Norway},
  author  = {Bjørkøy, Olav},
  school  = {NTNU},
  year    = {2011},
  title   = {{Adaptive Aggregation of Recommender Systems}}
}
```

(1) See www.apache.org — accessed 19.05.2011
(2) See hadoop.apache.org — accessed 19.05.2011
(3) See mahout.apache.org — accessed 19.05.2011
(4) See lucene.apache.org — accessed 19.05.2011
(5) See www.apache.org/licenses — accessed 19.05.2011
(6) See www.jruby.org — accessed 09.05.2011
(7) See www.ctan.org/tex-archive/macros/latex/contrib/memoir — accessed 19.05.2011
(8) See www.texample.net/tikz — accessed 23.05.2011

References

Adomavicius, G. and Tuzhilin, A. (2005). Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749.

Agichtein, E., Brill, E., Dumais, S., and Ragno, R. (2006). Learning user interaction models for predicting web search result preferences. *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '06*, page 3.

Ahn, J. and Hong, T. (2004). Collaborative filtering for recommender systems: a scalability perspective. *International Journal of Electronic Business*, 2(1):77–92.

Albert, R., Jeong, H., and Barabási, A. (1999). The diameter of the world wide web. *Arxiv preprint cond-mat/9907038*, pages 1–5.

Alshamri, M. and Bharadwaj, K. (2008). Fuzzy-genetic approach to recommender systems based on a novel hybrid user model. *Expert Systems with Applications*, 35(3):1386–1399.

Aslam, J. a. and Montague, M. (2001). Models for metasearch. *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '01*, pages 276–284.

Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern information retrieval*, volume 463. ACM press New York.

Banko, M. and Brill, E. (2001). Mitigating the paucity-of-data problem: Exploring the effect of training corpus size on classifier performance for natural language processing. In *Proceedings of the first international conference on Human language technology research*, pages 1–5. Association for Computational Linguistics.

Bao, S., Xue, G., Wu, X., Yu, Y., Fei, B., and Su, Z. (2007). Optimizing web search using social annotations. In *Proceedings of the 16th international conference on World Wide Web*, pages 501–510. ACM.

Barabási, A. (2003). Linked: The new science of networks. *American journal of Physics*.

Basu, C., Hirsh, H., and Cohen, W. (1998). Recommendation as Classification: Using Social and Content-Based Information in Recommendation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 714–720. JOHN WILEY & SONS LTD.

Bawden, D. and Robinson, L. (2009). The dark side of information: overload, anxiety and other paradoxes and pathologies. *Journal of Information Science*, 35(2):180–191.

Bell, R., Koren, Y., and Volinsky, C. (2007a). Modeling relationships at multiple scales to improve accuracy of large recommender systems. *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '07*, page 95.

Bell, R., Koren, Y., and Volinsky, C. (2007b). The BellKor solution to the Netflix prize. *KorBell Team's Report to Netflix.*

Bell, R. M. and Koren, Y. (2007a). Lessons from the Netflix prize challenge. *ACM SIGKDD Explorations Newsletter*, 9(2):75.

Bell, R. M. and Koren, Y. (2007b). Scalable Collaborative Filtering with Jointly Derived Neighborhood Interpolation Weights. *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 43–52.

Bennett, J. and Lanning, S. (2007). The netflix prize. In *Proceedings of KDD Cup and Workshop*, volume 2007, page 8. Citeseer.

Billsus, D. and Pazzani, M. (1998). Learning collaborative information filters. In *Proceedings of the Fifteenth International Conference on Machine Learning*, volume 54, page 48.

Bjørkøy, O. (2010). User Modeling on The Web: An Exploratory Review.

Brand, M. (2003). Fast online SVD revisions for lightweight recommender systems. *SIAM International Conference on Data Mining.*

Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

Burke, R. (2007). Hybrid web recommender systems. In *The adaptive web*, pages 377–408. Springer-Verlag.

Carmel, D., Zwerdling, N., Guy, I., Ofek-Koifman, S., Har'el, N., Ronen, I., Uziel, E., Yogev, S., and Chernov, S. (2009). Personalized social search based on the user's social network. *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*, page 1227.

Cato, J. (2001). *User-centered web design.* Addison Wesley Longman, 1st edition.

Claypool, M., Gokhale, A., Miranda, T., Murnikov, P., Netes, D., and Sartin, M. (1999). Combining Content-based and collaborative filters in an online newspaper. In *Proceedings of ACM SIGIR Workshop on Recommender Systems*, number June, pages 60–64. Citeseer.

Davenport, T. and Beck, J. (2001). *The attention economy: Understanding the new currency of business.* Harvard Business Press.

Dietterich, T. (2000). Ensemble methods in machine learning. *Multiple classifier systems*, pages 1–15.

Dwork, C., Kumar, R., Naor, M., and Sivakumar, D. (2001). Rank aggregation methods for the Web. *Proceedings of the tenth international conference on World Wide Web - WWW '01*, pages 613–622.

Edmunds, A. and Morris, A. (2000). The problem of information overload in business organisations: a review of the literature. *International Journal of Information Management*, 20(1):17–28.

Eppler, M. and Mengis, J. (2004). The concept of information overload: A review of literature from organization science, accounting, marketing, MIS, and related disciplines. *The Information Society*, 20(5):325–344.

Fischer, G. (2001). User modeling in human–computer interaction. *User modeling and user-adapted interaction*, 11(1):65–86.

Goldberg, K., Roeder, T., Gupta, D., and Perkins, C. (2001). Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151.

Halevy, A. and Norvig, P. (2009). The unreasonable effectiveness of data. *Intelligent Systems, IEEE*, 24(2):8–12.

Haveliwala, T. (2003). Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796.

Herlocker, J., Konstan, J., Terveen, L., and Riedl, J. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53.

Horvitz, E., Kadie, C., Paek, T., and Hovel, D. (2003). Models of attention in computing and communication: from principles to applications. *Communications of the ACM*, 46(3):52–59.

Hotho, A., J, R., Schmitz, C., and Stumme, G. (2006). Information Retrieval in Folksonomies: Search and Ranking.

Huang, C., Sun, C., and Lin, H. (2005). Influence of local information on social simulations in small-world network models. *Journal of Artificial Societies and Social Simulation*, 8(4):8.

Huang, Z., Chung, W., Ong, T.-H., and Chen, H. (2002). A graph-based recommender system for digital library. *Proceedings of the second ACM/IEEE-CS joint conference on Digital libraries - JCDL '02*, page 65.

Jameson, A. (2009). Adaptive interfaces and agents. *Human-Computer Interaction: Design Issues, Solutions, and Applications*, page 105.

Joachims, T., Granka, L., Pan, B., Hembrooke, H., Radlinski, F., and Gay, G. (2007). Evaluating the accuracy of implicit feedback from clicks and query reformulations in Web search. *ACM Transactions on Information Systems*, 25(2):7–es.

Kirsh, D. (2000). A few thoughts on cognitive overload. *Intellectica*, 1(30):19–51.

Klementiev, A., Roth, D., and Small, K. (2008). A Framework for Unsupervised Rank Aggregation. *Learning to Rank for Information Retrieval*, 51:32.

Konstas, I., Stathopoulos, V., and Jose, J. (2009). On social networks and collaborative recommendation. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 195–202. ACM.

Koren, Y. (2008). Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434. ACM.

Lee, U. and Liu, Z. (2005). Automatic identification of user goals in web search. *international conference on World Wide Web*, (1):391–400.

Lemire, D. and Maclachlan, A. (2005). Slope one predictors for online rating-based collaborative filtering. *Society for Industrial Mathematics*.

Lieberman, H. (1997). Autonomous interface agents. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 67–74. ACM.

Lieberman, H. (2009). User interface goals, AI opportunities. *AI Magazine*, 30(2).

Lilegraven, T. N., Wolden, A. C., Kofod-Petersen, A., and Langseth, H. (2011). A design for a tourist CF system. In *The Eleventh Scandinavian Conference on Artificial Intelligence*, pages 193–194, Trondheim.

Liu, F., Yu, C., and Meng, W. (2002). Personalized web search by mapping user queries to categories. *Proceedings of the eleventh international conference on Information and knowledge management - CIKM '02*, page 558.

Liu, H., Maes, P., and Davenport, G. (2006). Unraveling the taste fabric of social networks. *International Journal on Semantic Web and Information Systems*, 2(1):42–71.

Liu, Y., Liu, T., Qin, T., Ma, Z., and Li, H. (2007). Supervised rank aggregation. In *Proceedings of the 16th international conference on World Wide Web*, pages 481–490, New York, New York, USA. ACM.

Manning, C., Raghavan, P., Schutze, H., and Corporation, E. (2008). *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, UK.

Mirza, B. and Keller, B. (2003). Studying recommendation algorithms by graph analysis. *Journal of Intelligent Information*.

Mobasher, B., Cooley, R., and Srivastava, J. (2000). Automatic personalization based on Web usage mining. *Communications of the ACM*, 43(8):142–151.

Newman, M., Moore, C., and Watts, D. (2000). Mean-field solution of the small-world network model. *Physical Review Letters*, 84(14):3201–3204.

Noll, M. and Meinel, C. (2007). Web search personalization via social bookmarking and tagging. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, pages 367–380. Springer-Verlag.

Norman, D. (2002). *The design of everyday things*, volume 16. Basic Books New York.

Pazzani, M. and Billsus, D. (2007). Content-based recommendation systems. In *The adaptive web*, pages 325–341. Springer-Verlag.

Pitsilis, G. and Knapskog, S. (2009). Social Trust as a solution to address sparsity-inherent problems of Recommender systems. *Recommender Systems & the Social Web.*

Polikar, R. (2006). Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45.

Qiu, F. and Cho, J. (2006). Automatic identification of user interest for personalized search. *Proceedings of the 15th international conference on World Wide Web - WWW '06*, page 727.

Ranade, a., Mahabalarao, S., and Kale, S. (2007). A variation on SVD based image compression. *Image and Vision Computing*, 25(6):771–777.

Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994). GroupLens : An Open Architecture for Collaborative Filtering of Netnews.

Rhodes, B. J. and Maes, P. (2000). Just-in-time information retrieval agents. *IBM Systems Journal*, 39(3):685–704.

Rich, E. (1979). User modeling via stereotypes. *Cognitive science*, 3(4):329–354.

Robertson, S. (2010). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389.

Schafer, J., Frankowski, D., Herlocker, J., and Sen, S. (2007). Collaborative filtering recommender systems. *The adaptive web*, pages 291–324.

Segaran, T. (2007). *Programming collective intelligence.* O'Reilly Books, 1st edition.

Sergey, B. and Lawrence, P. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117.

Shen, X., Tan, B., and Zhai, C. (2005). Implicit user modeling for personalized search. *Proceedings of the 14th ACM international conference on Information and knowledge management - CIKM '05*, page 824.

Smyth, B. (2007). Case-based recommendation. *The adaptive web*, pages 342–376.

Speretta, M. and Gauch, S. (2000). Personalized Search Based on User Search Histories. *The 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI'05)*, pages 622–628.

Sugiyama, K., Hatano, K., and Yoshikawa, M. (2004). Adaptive web search based on user profile constructed without any effort from users. *Proceedings of the 13th conference on World Wide Web - WWW '04*, page 675.

Sun, J., Zeng, H., Liu, H., and Lu, Y. (2005). CubeSVD: a novel approach to personalized Web search. *on World Wide Web*, pages 382–390.

Teevan, J., Adar, E., Jones, R., and Potts, M. (2007). Information re-retrieval: repeat queries in Yahoo's logs. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 151–158. ACM.

Teevan, J., Dumais, S. T., and Horvitz, E. (2005). Personalizing search via automated analysis of interests and activities. *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '05*, page 449.

Totterdell, P. and Rautenbach, P. (1990). Adaptation as a problem of design. *Adaptive user interfaces*, pages 61–84.

Ujjin, S. and Bentley, P. (2002). Learning user preferences using evolution. In *Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution And Learning (SEAL'02)*, pages 6–10. Citeseer.

Umbrath, A. and Hennig, L. (2009). A hybrid PLSA approach for warmer cold start in folksonomy recommendation. *Recommender Systems & the Social Web*, pages 10–13.

Venetis, P., Gonzalez, H., and Jensen, C. (2011). Hyper-local, directions-based ranking of places. *Proceedings of the VLDB*, pages 290–301.

Walter, F., Battiston, S., and Schweitzer, F. (2008). A model of a trust-based recommendation system on a social network. *Autonomous Agents and Multi-Agent Systems*, 16(1):57–74.

Widmer, G. and Kubat, M. (1996). Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101.

Xu, S., Bao, S., Fei, B., Su, Z., and Yu, Y. (2008). Exploring folksonomy for personalized search. *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '08*, page 155.

Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R. (2008). Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic aspects in information and management: 4th international conference, AAIM 2008, Shanghai, China, June 23-25, 2008. proceedings*, volume 5034, page 337. Springer-Verlag New York Inc.

Ziegler, C. (2005). *Towards decentralized recommender systems*. PhD thesis, Universitatsbibliothek Freiburg.