Problem Description

Study how to aggregate user models on a per-user and per-item basis when combining results from complementing modeling algorithms. 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: 17. January 2011
Supervisor: Asbjørn Thomassen

Adaptive Recommenders

Personalized Prediction Aggregation
Through Accuracy Estimation

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

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

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

Abstract

In the field of artificial intelligence, *recommender systems* are user modeling methods that can predict the relevance of an item to a user. Items can be just about anything: documents, articles, movies, music, events or other users. Recommender systems examine data such as ratings, query logs, user behaviour and social connections to predict what each user will think of each available item.

Modern recommender systems use ensemble learning techniques, combining multiple standard recommenders, in order to leverage disjoint patterns in the available data. By combining different methods, complex predictions that rely on much evidence can be made. These aggregations are done on a generalized level, often by weighting each recommender in a way that achieves an optimal result.

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 each user and item *should* be modeled. We believe the selection of recommender methods should be adaptively and automatically chosen based on how accurate each prediction is likely to be 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, called *adaptive recommenders*. Multiple recommender systems are combined on a per-user and per-item basis by estimating how accurate each recommender will be for the current user and item. This is done by creating a set of secondary error estimating recommenders. The core insight is that standard recommenders can be used to estimate the accuracy of other recommenders, for each user/item pair. As far as we know, this type of adaptive prediction aggregation has not been done before.

We test prediction aggregation (combining scores) in a recommendation scenario, and rank aggregation (sorting results lists) in a personalized search scenario. Our initial results are promising, showing that adaptive recommenders can outperform both standard recommenders and simple aggregation methods. We 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).

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, Kjetil Valle and Magne Matre Gåsland, who helped me formulate my thoughts and provided feedback on the work represented by this document.

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

Information is a curious thing: having too much can be as harmful as having no information at all. While lacking information is an obvious problem, too much leads to information overload, where relevant content drowns in irrelevant noise. This is a common problem: whenever we have enough information, any extra data only leads to confusion. Our ability to make informed decisions is often the first thing to go (Davenport and Beck, 2001, p1).

However, while people struggle with excessive information, algorithms in *artificial intelligence* (AI) can not get enough. Halevy and Norvig (2009, p1) calls this the "unreasonable effectiveness of data": perhaps surprisingly, more data often trumps more efficient algorithms. For example, Banko and Brill (2001, p3) show how common algorithms in AI can be substantially improved 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). An item can be just about anything: 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 soon see. Recommender systems are especially prolific on the Web: wherever there is personalized recommendations of news, books, movies, articles, social connections, search results, et cetera, recommender systems are working behind the scenes.

Modern recommendation approaches often embrace the unreasonable effectiveness of data, by combining multiple recommender systems, that each predict relevance in various ways. By considering different aspects of users and items when making predictions, the methods provide quite complex predictions that rely on much evidence. For example, Bell et al. (2007b, p1) took this 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 might seem constraining, recommender systems are incredibly powerful tools. If we can accurately predict how each user will react to each item, we will have come a long way towards solving information overload. Likewise, chronological sorting of items would be a figment of a simplistic past, as sorting based on actual relevance would be far superior.

However, despite their apparent power, recommender systems are often confined to simple tasks like creating small lists of recommended items or suggesting similar items to the ones being considered by a user. Common examples are lists of recommended items

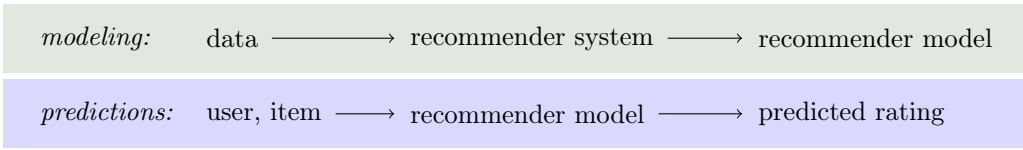


Figure 1.1: A simplified view of recommender systems: Ratings of items by users are used to create a model. *This model is then used to predict unknown ratings between users and items. Note that many recommender systems work differently, as we shall soon see.*

based on the one being viewed, recommending new social connections, or suggesting news articles based on previous reading. Seldom are their full potential reached by creating completely adaptive content systems, that work hard to mitigate any signs of information overload.

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 the full adoption of these systems. There is a mismatch between how recommender systems perform predictions, and how each user and item wants these predictions to be made.

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 might feel the similarity of movie titles are a good predictor, while another might be more influenced by 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 items: 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 — what is important is that they exist.

Another way of explaining the latent subjectivity problem is that *user modeling methods are dependent on the subjective assumptions of their creators.* In other words, a modeling method use some aspect of available data to make predictions, and this aspect is chosen by whoever creates the system.

Aggregate modeling methods face the same problem of misplaced subjectivity:

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 of course selected to minimize some error over a testing set, the subjective nature remains: the compiled aggregation is a generalization, treating all users the same — hardly a goal of user modeling.

Should it not be up to each user to implicitly decide which method best describes their preferences? And, considering the vast scope of items we can come by, will the selected methods really 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 each user and item. Without this adaptability, it may be hard for recommender systems to gain traction 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.

We propose a novel method called *adaptive recommenders*, where these decisions are left to each user and item, providing an extra level of abstraction and personalization. The 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: in the hands of each individual user, and dependent on each specific item, without any extraneous effort. If each method of relevance prediction is *only used* based on how well it performs for each element, any possibly applicable recommender system suddenly becomes a worthy addition to the system.

As far as we know, such adaptive prediction aggregation has not been done before. Can a system where each user and item implicitly decides how they should be modeled outperform traditional approaches? *That is the main research question of this thesis.*

This thesis is structured as follows. Chapter 2 will present background theory and previous work for the information overload problem, recommender systems, prediction and rank aggregation, and personalized search. Chapter 3 will build the *adaptive recommenders* approach from the ground up, and show how this can be used for both prediction aggregation and rank aggregation.

Chapter 4 will test three hypotheses and experiment with our newly built model. We will experiment with prediction aggregating for singular items, and 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 take on user modeling. 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 our user modeling method 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 (Bjorkoy, 2010, p13). This is a widespread phenomenon, with as many names and definitions as there are fields experiencing the problem. Examples include *sensory overload*, *cognitive overload* and *information anxiety* (Eppler and Mengis, 2004, p2). Two common tasks quickly become difficult in this situation:

1. Consuming relevant content is hindered by too much irrelevant noise.
2. Discovering new content is difficult because of too much 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.
- Discovering sub-par entertainment because the most relevant is never discovered.
- Reformulating search queries because the results include irrelevant items.

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, p6): "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, p5) 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 to increase productivity and reduce the level of stress through careful consideration of the user's needs. In other words, to solve the problems of information overload and content discovery, applications must be able to individually adapt to each user.

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. Attention can then be defined 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" (Davenport and Beck, 2001, p1). To evade information overload is then to maximize available attention, allowing more focus on the most important items.

Conceptual models used in interaction design can help us see when and where information overload interferes with the user experience. Norman (1988) 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). 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.

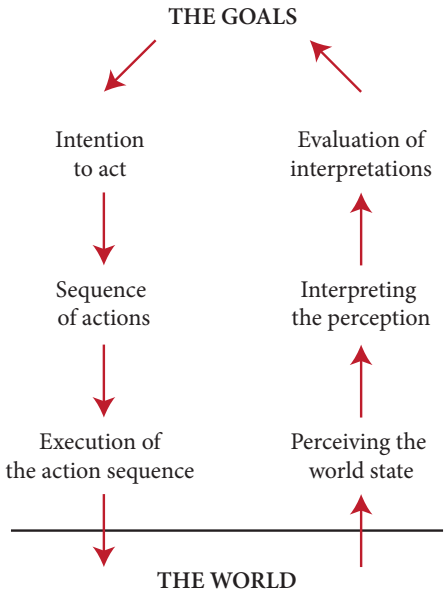


Figure 2.1: Stages of Action

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 happens throughout the interaction process, which is important to know when considering possible solutions.

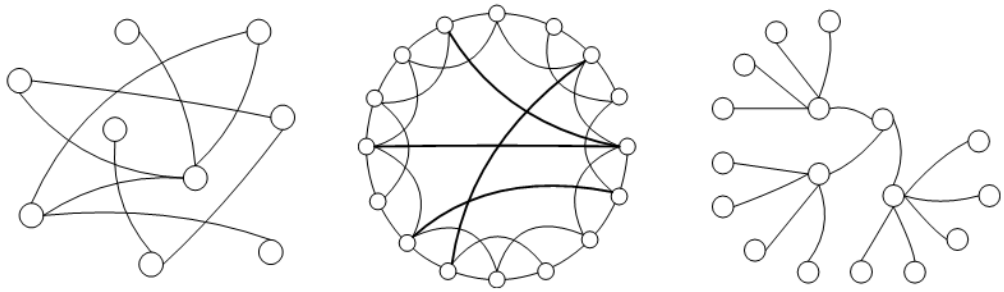


Figure 2.2: Complex Networks, from the left: A random *network, a* small-world *network and a* scale-free *network (which is a type of small-world network). Figure adapted from (Huang et al., 2005, p2).*

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*: sites that collate information from multiple other sites and sources. Online information retrieval (search engines), fall into this category, as does online newspapers, feed readers and portal websites.

The wealth and scope of data on the Web are natural culprits of online 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. 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.

The Internet has the properties of a *small-world network* ((Newman and Moore, 2000), (Huang et al., 2005, p2)), 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, p37), 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, p71).

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). In other words, a few important nodes have a huge number of inbound and outbound links — the hubs. Barabási (2003, p86) 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. In other words, older, more established and central nodes are preferred neighbors. This is called the Barabási-Albert model (Albert et al., 1999), and 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, that can predict how relevant each item will be for each 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 application 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. The source of information overload problem often comes down to a mismatch between presented content and desired content. Examples of this kind of user modeling 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, p6), 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 precise 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 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). As our goal is to combine different RSs into one coherent user model, we shall now describe what makes a recommender system, and introduce some of the many algorithms they employ.

2.3 Recommender Systems

While the name might seem constraining, recommender systems are incredibly powerful methods in 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 predictive functionality, including:

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

Common to these examples are a set of users, a set of items, and a sparse set of explicit ratings or preferences. Items can be just about anything: documents, movies, music, places, people, or indeed other users. 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. Items and users are nodes, while ratings, social connections et cetera are edges between the nodes. An RS performs predictive reasoning on this graph by estimating the strengths of hypothetical connections between nodes that are not explicitly connected.

Note that although we use "ratings", "utility", "preference", "relevance" and "connection strength" depending on the context, they all basically mean the same in this context.

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.

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.

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. In other words, 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.

2.3.1 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, p2). 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. We are only interested in fully autonomous, intelligent interfaces. We wish to create a system that implicitly, and without any effort from each user, can adapt the content of an application based on previous behaviour Examples of implicit user modeling include Qiu and Cho (2006), Shen et al. (2005) and Carmel et al. (2009).

2.3.2 Aspects of Recommender Systems

Formally, a recommender system can be seen as a quintuple, RS $= (I, U, R, F, M)$, where I is the set of items (e.g. products, articles or movies) and U is the set of users. R is the set of known ratings or utility, 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 behaviour (e.g. query log mining). F is a framework for representing the items, users and ratings, for example 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, p2), M is seen as a utility (the rating, in AI terms) estimation function $p : U \times I \to S$. 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 each user has either an explicit, implicit or predicted preference for each item in I. To predict the best unrated item for each 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 active user and the item in question. The *reason* for using a recommender system is that the utility (each r) is not defined for the entire $U \times I$ space, i.e. the system does not explicitly know the utility of each item for each user. The point of a recommender system is then to extrapolate R to cover the entire user-item space. In other words, to be able to rank items according to user preferences, the system must be able to predict each user's reaction to items they have not yet explicitly or implicitly rated themselves.

Another popular 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 an known rating. This matrix then becomes the framework 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 or connects to a few number of items. This is true for any scenario where users 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, p1). In other words, this recommender system had to be able to produce results from a matrix where only 1% of the cells have meaningful values.

This is the defining characteristic of many recommender systems: their ability to extract meaningful patterns from sparse data, through dimensionality reduction, neighborhood estimation and many other methods, as we shall soon see. Naturally, much research looks at ways to best tackle this sparsity (e.g. Pitsilis and Knapskog (2009), Claypool et al. (1999, p3), Ziegler (2008, p19)).

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. As in many machine learning methods, concept drift (Widmer and Kubat, 1996, p1), where the characteristics of a user or item changes over time, is also always present.

The performance of RSs is often closely tied to their computational complexity (as mentioned in Adomavicius and Tuzhilin (2005, p6)). Real world usage of the most precise methods is often 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 techniques such as RMSE or MAE treat ordinal data as a continuous scale. We will get back to this in Chapter 5.

2.3.3 Predicting Ratings

The crucial part of any RS is how it predicts unknown ratings. Because of this, each method is best categorized based on dimensions of its predictive capabilities (see Table 2.1). We use a taxonomy where these dimensions are: available *data*, prediction *method*, model *granularity*, knowledge *temporality* and knowledge gathering *agents*.

The *data* variable represents what data an RS uses to perform predictions. Content-based methods use only items, intra-item relations, and an individual user's past history as predictive of future actions (Pazzani and Billsus, 2007, p1). By only considering the individual user in adapting an application, highly personal models can be created. However, such methods often require a lot of interaction before reliable models can be created (Adomavicius and Tuzhilin, 2005, p4). The problem of having to do complex inference from little data, as is often is in content-based predictions, is similar to the

sparsity problem, is often called the *cold start* or *new user* problem. This is closely related to the common AI problem of *overfitting* data, 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 sparse data, i.e. achieving "warmer" cold start (e.g. Umbrath and Hennig (2009), Lilegraven et al. (2011)). 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, p2).

Collaborative or social recommendations build predictive models for users based on the actions of similar users (Schafer et al., 2007). The observation is that similar users should have similar usage and action patterns. By using data from more than one user, expansive models may be built. These methods are especially useful when considering new users of a service. A central problem with collaborative methods is that the resulting model is not as individually tailored as one created through content-based prediction. Collaborative models must be careful not to represent the *average* user, but a single individual. When using a collaborative method, the utility $p(u, i)$ of item i for user u is extrapolated from $p(u_x, i)$ where u_x is a user similar to u (Adomavicius and Tuzhilin, 2005, p4).

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.

The *method* variable, is another way to classify recommenders. Orthogonal to what data the method uses, this variable concerns *how* the data is used to produce recommendations. 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, p5). 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, p5). These methods use the raw data of items, users and ratings to

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 Bjorkoy (2010).

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.

The *granularity* variable tells whether this approach creates models for the canonical user, stereotypical users or individual users. For example, Rich (1979) presented one of the first user modeling systems based on stereotypes, used to predict which books in a library each user 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.

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.

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 user models (e.g. Shen et al. (2005); Agichtein et al. (2006); Speretta and Gauch (2000); Teevan et al. (2005))

2.3.4 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. Let us take 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 Bjorkoy (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, based on rating averages. The data is content-based, and used to compute heuristic predictions. While simple in nature, they are often helpful as starting points for more complex systems, or as benchmarks for exploring new approaches. (Koren, 2008, p2) computes the baselines for items and users, and use more involved methods to move this starting point in some direction. The baseline for a user/item pair is given by

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

where 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

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

which 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 the user u has rated item j, that is, we know r_{uj}, the predicted rating of i is simply $r_{uj} + \delta$, for all ratings that match this pattern. In other words,

$$\hat{r}_{ui} = \frac{\sum_{j\in K_u} \text{ratings}(j) \times (r_{uj} + \hat{\delta}(i,j))}{\sum_{j\in K_u} \text{ratings}(j)},$$

where $\hat{r}_{ui}$ is the estimated rating, K_u is the items rated by user u, ratings(i) is the number of ratings for item i, and $\hat{\delta}(i,j)$ 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, p5).

(2) *Neighborhood estimation* is part of many recommendation systems. It is the core principle behind most collaborative filtering algorithms, that estimate an unknown rating by averaging the ratings of similar items or users, weighted by their similarity. These approaches often work in two steps: First, a neighborhood if similar elements is computed. Second, the similarities and connections within this neighborhood is used to produce a prediction.

The principal method for computing user similarity is the *Pearson Correlation Coefficient* (PCC) (Segaran, 2007, p11). 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 for example signify two users that have diametrically opposing tastes in movies. We compute PCC by dividing the covariances of the user ratings with their standard deviations:

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

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

$$\text{pcc}_n(u, v) = \frac{\sum_{i \in K}^{n}(R_{ui} - \bar{R}_u)(R_{vi} - \bar{R}_v)}{\sqrt{\sum_{i \in K}^{n}(R_{ui} - \bar{R}_u)^2}\sqrt{\sum_{i \in K}^{n}(R_{vi} - \bar{R}_v)^2}}.$$

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 our the formula.

After a neighborhood is determined, it is time to predict our desired rating. When using *collaborative* recommenders, this means averaging the neighborhood ratings weighted by similarity (Segaran, 2007, p16):

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

where $\text{sim}(u, v)$ 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 possible the simplest way 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 based on neighborhoods work on similar principles. The difference is that we compute similarities between items, not 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, p10), Spearman's or Kendall Tau rank correlation coefficients (Herlocker et al., 2004, p30) — 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 information retrieval, such as *cosine correlation* from the *vector space model* (VSM) are also popular, used to compute item similarities (see 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 metrics are also possible: Ujjin and Bentley (2002) first use a simple euclidean metric to gather a larger neighborhood, which is then refined by a *genetic algorithm*. Another 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. Since this has the added effect of trying to approximate unknown cells, the lower dimension matrices can be used to predict unknown ratings (a type of least squares data fitting).

Singular Value Decomposition (SVD) is a common method for such matrix factorization (e.g. Billsus and Pazzani (1998, p5), 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, p44). Formally, SVD is the factorization $M = U\Sigma V^T$. M 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 (sometimes called the "hanger"), V^T (the transpose of V) is an $n \times n$ factor (the "stretcher"). Σ is a $m \times n$ diagonal matrix (the "aligner"). 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 $M_r = U_r\Sigma_r V_r^*$, 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 the most common way of reducing the factors. By truncating 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 result looks like this:

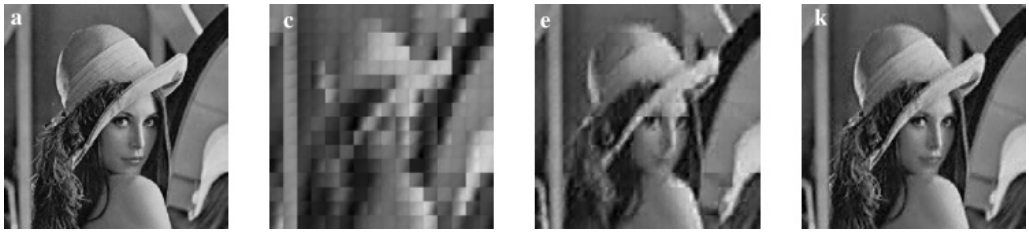


Figure 2.4: SVD Image Compression: A variation of using SVD to compress an image, from Ranade et al. (2007). The original image is on the left, and successive images use an increasing number of factors (2, 8 and 30) when performing compression. Figure adapted from Ranade et al. (2007, p4).

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

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 taste-space. In other words, the number of features corresponds 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, p5), Brand (2003, p4) or Liu and Maes (2006, p2)). 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 of course 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.

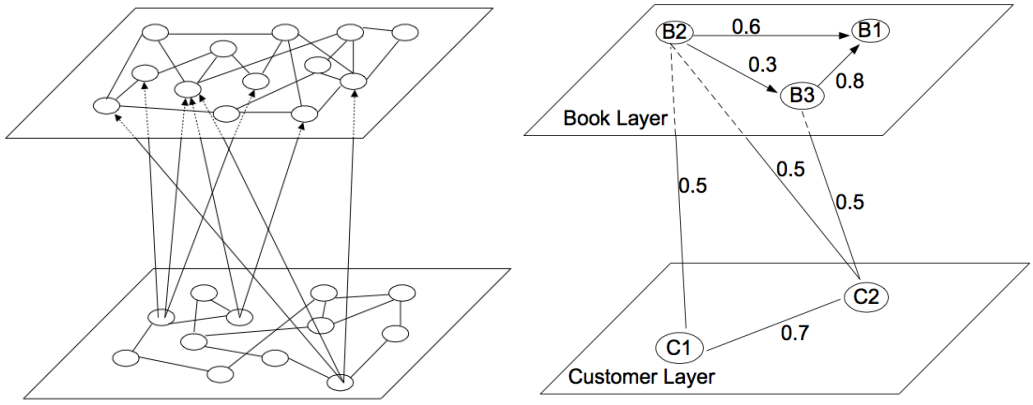
(4) *Network traversal* refers to estimating predictions by traversing a graph of users and items to provide recommendations. The connections between nodes can be any type of relation that makes sense to the RS. Examples include linking item and user nodes based on purchases or explicit ratings, linking 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. In other words, users trust each others 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 from a node x, the RWR algorithm randomly follows a link to a neighboring node. In



(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).

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

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

when it is 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. In other words, 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 the next chapter.

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 system. 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 present the basics of *information retrieval* (IR).

2.4.1 Information Retrieval

Manning et al. (2008, p1) 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 *recommending* 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. In other words, 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.

Baeza-Yates and Ribeiro-Neto (1999, p23) 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, p19). 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, p22), that lists documents by terms. There are numerous extensions to this framework, including:

- Positional information for phrase search (Manning et al., 2008, p39)
- Stop word removal (removing the most common terms) (Manning et al., 2008, p27)
- Stemming (reducing words to their root forms) (Manning et al., 2008, p32)
- Lemmatisation (contextual inflection removal) (Manning et al., 2008, p32)
- Query reformulation (Baeza-Yates and Ribeiro-Neto, 1999, p117)

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, p5):

- 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.

To us, the most interesting part of any IR system is the *ranking function*. This function compute the score of each document to the current query. The relation to recommender systems should be self-evident. 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, p29).

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, p29):

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

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), the application of SVD to IR by performing dimensionality reduction of the term-space into concept-space is another approach. 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: "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, p5) 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. Another 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 web page is used to choose the most important terms.

Signals can also depend on the query. Manning et al. (2008, p145) describes a system that takes multi-word queries, breaks them up into different permutations and

(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

runs each new query 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, p4) is perhaps the most 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. Each 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, p4). 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:

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

where 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 at one of the main uses of signals: *personalized search.*

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, 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)) In other words, these are often

model-based techniques with implicit knowledge gathering agents, that create individual, long-term user models (see Section 2.3).

There are two leading approaches to personalizing search results (Noll and Meinel, 2007, p2). 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) Haveliwala (2003) introduced a topic-sensitive PageRank algorithm, that they found to "generate more accurate rankings than with a single, generic PageRank vector". In essence, they create topic-specific PageRank vectors for a number of pre-set topics, creating many rankings per page, one for each topic. This new PageRank is computed based on an existing set of websites that belong to each topic. 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 each of 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, p5)), 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 for each user. By identifying how many sites the user has visited in each topic, computing T can be done through linear regression or by using a Maximum-likelihood estimator (basically, any method that can fit a curve). Qiu and Cho (2006, p10) reports improvements of 25% to 33% over the Topic-sensitive PageRank approach, which Haveliwala (2003) reports outperformed the original PageRank algorithm.

(2) 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* (WBS) aggregation method (Xu et al., 2008, p3), which is another name for weighted combination of the rankings:

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

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) 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, p1). Note, however 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, p2), e.g. co-authoring of documents, the use of similar tags (see (2)) or commenting 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. First, the score based on connections and terms is computed, weighted by β which determines the weighting of both approaches:

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

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:

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

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, p10) notes that the high quality data in their enterprise setting were important to achieve the improved results.

2.5 Aggregate Modeling

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. In other words, 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, p5) aggregates signals measured from website structure.

Clearly, aggregation is an important part of both RS and PS. The reasoning behind combining different approaches is that no one methods can capture all the predictive nature of available data. For example, Bell et al. (2007a) created a recommender systems 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", major structural patterns in the data: "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 [...]" (Bell et al., 2007a, p1)

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, p4) 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 aggregate approaches: Multiple prediction techniques are used to create a result better than any single methods would provide.

There are two main use cases for model aggregation (Liu et al., 2007, p1):

1. Rank (or *order-based*) aggregation (RA) lets each method 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, p1).

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 each method (Aslam and Montague, 2001, p2).

2.5.1 Rank Aggregation

RA combines multiple result lists into one list by some metric. Dwork et al. (2001) shows a few metrics applicable to meta-search, the combination of results from multiple search engines. Borda's method (Dwork et al., 2001, p6) 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 a method where each method has a set number of votes (c, the number of results) that they give to each item.

As we saw in Section 2.4, Xu et al. (2008, p3) used a weighted version of this approach to combine an IR and personal approach to result ranking. Aslam and Montague (2001, p3) 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 each method. Aslam and Montague (2001, p4) 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, p1) results with these rank aggregations were positive: "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, p7).

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.

Aslam and Montague (2001) describe a number of simple approaches: Min, max and sum models combine the individual predictions in some way, or select one or more of the results as the final prediction. Other models use the average, or log-average of the different methods. The linear combination model 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, p3) 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% (Aslam and Montague, 2001, p3), showing that even fairly simple combinations have merit.

Early approaches in recommender systems dabbled in aggregating content-based and collaborative approaches. Claypool et al. (1999) combined the two approaches in an effort to thwart problems with each method. Collaborative filtering (CF) methods have problems rating items for new users, radically different users or when dealing with very sparse data. Content-based (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* (EM) (Dietterich, 2000, p1). 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, p27).

Bell et al. (2007b) took method aggregation to its logical conclusion when winning the Netflix Challenge, by combining 107 individual results from different recommenders: "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." (Bell et al., 2007b, p6) In other words, the final result is improved because of the disjoint reasoning performed by the different predictors.

Like RA, PA is an extensive topic. 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 user modeling, called *adaptive recommenders*. We will first explain why a new approach is needed, and develop three hypotheses based on the assumptions in Chapter 1. The two final sections will show how *adaptive recommenders* can perform prediction aggregation (combining scores) in a recommendation scenario, and rank aggregation (combining sorted lists) in a personalized search scenario.

3.1 Latent Subjectivity

As we saw in Chapter 2, there are lots of 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.

Aggregate modeling is a common way of combining different, complimenting methods into one prediction system. By leveraging so called *disjoint patterns* in the data, several less than optimal predictors can be combined, so that the combination outperforms each part. 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, *latent subjectivity* is a fundamental problem of every 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.
- Recommendations based on the Pearson Coefficient (Segaran, 2007, p11) assumes that the statistical correlation between user ratings is a good measure for user similarity.

Are these metrics subjective? 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. Imagine creating a recommendation system, and consider the following 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 should be embraced, by allowing users to implicitly and automatically select which methods they prefer, from a large set of possible predictors.

Just as each user is different, each item may have their own characteristics. Needless to say, items are often quite different from another, along a myriad of dimensions. Consider the World Wide Web: If each web page is an item, 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.

As before, 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 item the same. No matter its intrinsic qualities, an item will be judged by the same methods as every other item.

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 sidestep the latent subjectivity problem. If each method is *only used* based on how well it performs for each element, any possibly applicable recommender system suddenly becomes a worthy addition to the system. The user is in control of which methods best fit their needs, and each method's priority is influenced by how well it performs for the current item.

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 user and item. In other words, by automatically adapting how a set of disjoint recommenders are combined, based on each user and item, we should be able to achieve a result that is better than each 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: The accuracy of relevance predictions can be improved through adaptive recommender aggregation.*

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 each individual user and item, should outperform each of the individual approaches. Second:

> *H2: Adaptive aggregation can achieve higher accuracy than global and 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 result set from an information retrieval query can be adaptively personalized by layering 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.

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

3.3 Adaptive Recommenders

Adaptive Recommenders (SR) is a technique for combining recommender systems in an effort to answer two important questions. Given that we wish to predict the relevance of an item to a user, using many methods that consider disjoint data patterns,

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

User modeling methods and 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 combines many methods using a generic (often weighted) combination. However, 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). 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 SR can use recommender systems for both these tasks, as we shall soon see. A system for adaptive recommenders is specified by a 6-tuple:

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

As always, we have sets of $Users$ and $Items$, and a set of $Ratings$: each user $u \in U$ can produce a rating $r \in R$ of an item $i \in I$. Items can be just about anything: 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 mined 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*. In other words, this is 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, that is what we shall use.

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:

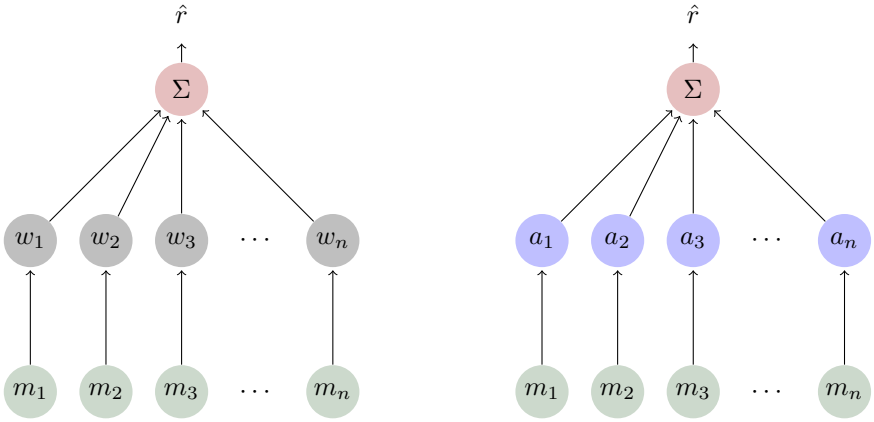


Figure 3.1: Comparison of aggregation and adaption: (left) modern aggregation approaches uses a set of pre-trained weights to prioritize each modeling method. The weighted predictions are aggregated into a final prediction $\hat{r}$. (right) Adaptive user modeling employs secondary modeling methods instead 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*, each with their own way 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, that consider different aspects in the data: users, items and ratings, as well as sources such as intra-user connections in social networks or intra-item connections in information retrieval systems. Examples of such recommender systems include Slope One predictions, SVD factorization and Nearest Neighbor weighted predictions (see Section 2.3.4). 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).

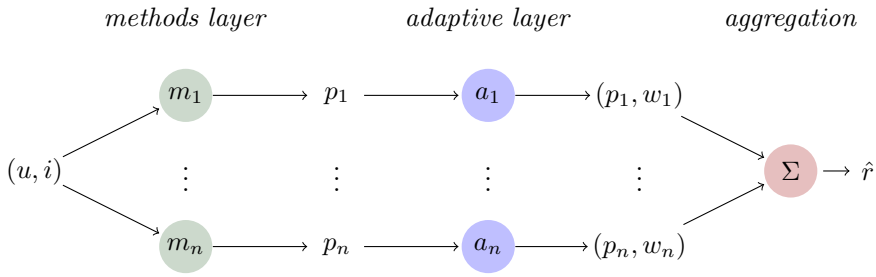


Figure 3.2: Layers of recommenders: The method layer consists of ordinary modeling methods, each predicting the rating between a user and an item. This produces a set of predicted ratings (p). The adaptive layer estimates how well each modeling method will perform for the current user and item, and weighs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

The *Adapters* part of our 6-tuple refers to the second level of user modeling methods. In traditional prediction aggregation this is a simple linear function for combining the different predictions, for example by pre-computing a set of weights, one for each method. As found by Bell et al. (2007b, p6) the accuracy of the combined predictor is more dependent on the ability of the various predictors to expose different aspects of the data, than on the individual accuracy of each predictor. As described in Section 2.5, multiple prediction results are normally combined into a final singular result, based on a generalized combination found by minimizing some error across all users.

With adaptive recommenders, the *Adapters* are themselves user modeling methods (see Figure 3.1). However, instead of modeling users, we wish to model each recommender system. More specifically, we wish to model the *accuracy* of each recommender system. 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. In other words, we have two distinct layers of user modeling (see Figure 3.2):

1. *The methods layer* consists of traditional user modeling methods, 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 modeling methods, 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. Each of these adaptive methods do not have to employ the same algorithm as their corresponding methods, the layers are only similar in that both consist of recommender systems.

Another way of describing (and implementing) these two levels is through the map and reduce functions of functional programming. For example, 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 other words, 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, p3)). 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 soon see.

Expressing ourselves in terms of map and reduce is helpful as this will guide our implementation into a proper MapReduce framework for parallel computation (as explained in Manning et al. (2008, p75)). For optimal performance, both layers can be performed in parallel, as we shall soon see.

3.3.1 Adaptive Aggregation

To perform adaptive 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, p3)):

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

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. However, many aggregators attempt to weigh each method differently (e.g. Claypool et al. (1999)):

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

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, the weights can be estimated through different machine learning methods. However, as discussed in Section 3.1, this is still a generalized result, averaged across every prediction. The system assumes that the best average result is the best result for each individual user and item. 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 the latent subjectivity, we need *adaptive weights* that are computed specifically for each combination of a user and an item. However, if we wish each weight to be combination-specific, pre-computing each weight for each method becomes unfeasible — we would have to compute a weight for each method for each possible rating. In other words, these adaptive weights also have to be estimated, just as the ratings themselves:

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

Here, $p_w(m,u,i)$ is the predicted optimal weight for method m when applied to user u and item i. In other words, we have reduced our mission of adaptive prediction aggregation to creating this function.

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 can 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 quite simple: by splitting the ratings data in two, the first set can be used for the actual training, and the second can be used to populate each error matrix.

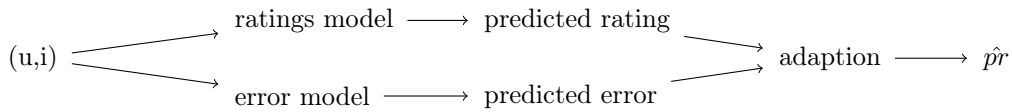


Figure 3.3: Multiple models for adaptive weights: The data flow through the adaption of a single recommender method. The current user and item is fed into two distinct models: the ratings model, which predicts unknown ratings, and the error model, which predicts how accurate this rating will be for the 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.

Each standard modeling method gets an error matrix where some cells have values: each value corresponds to the prediction error for a combination of a user and an item. Each modeling method is trained with a part of the ratings data. The error matrix is populated from the rest of the data, by computing the error of each 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 each rating which have not been used to train the method and calculate the error of the method on this combination. The result is a sparse error matrix we can use to predict unknown errors.

Notice how similar this matrix is to the previously introduced ratings matrix. This similarity is what will allow us using standard recommender systems to perform adaptive aggregation. Whenever we wish to train a new modeling method, *the modeling phase*, 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 mission 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.

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 each available method. We will get back to this when performing experiments in the next chapter.

When we have an error model for each modeling method, we can use these errors to estimate each weight. Whenever we wish to create an adaptive aggregate prediction, *the prediction phase*, we apply the following algorithm:

1. Collect predictions from each modeling method for (u, i).
2. Collect estimated errors for each method for (u, i).
3. Compute weights for each method 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:

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

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. Each method is weighted by its predicted accuracy. The weights are computed by taking the opposite of each methods predicted error. The errors are normalized across each user and item by $errors(u,i)$, which is the sum of the errors of each method 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. Each adaptive recommender need only predicted ratings from each basic recommender, and does not care which algorithm it employs. When making predictions, the calculations in the methods layer and adaptive

layer are independent of each other, 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 each basic method.

The result of this is a system that does not only aggregate a number of predictions for each unknown combination of users and items, but that also combines these methods based on how accurate each prediction is likely to be. Now that we have our model, it is time to see how it can be implemented. First, we shall do prediction aggregation in a recommendation scenario, and then rank aggregation in an information retrieval scenario.

3.4 Prediction Aggregation

Adaptive prediction aggregation means combining the results from multiple scalar predictors based on the current user and item (see Section 2.5.2). 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.

In this section, we shall explain how such a system may be created. Most importantly, there are two distinct phases to 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, when dealing with prediction aggregation. The next section will explain a similar situation where we wish to do *adaptive rank aggregation*: 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).

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 to create the standard recommenders.
2. Training sets to create the error estimation models.
3. A testing set to test our final system.

Algorithm 1 Adaptive Prediction Aggregation Modeling

Input: ratings: The ratings matrix
Input: methods: The set of modeling methods
Output:

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** $(rating\_models, error\_models)$

Constructing these subsets of the available data is a common task in ensemble learning (Polikar, 2006, p7). 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 perfectly, for a few reasons: First, the method helps create disjoint predictors, since each predictor is only trained (or specialized for) a subset of the available data. Second, it allows us to easily train the underlying modeling methods without any complex partitioning of the data. Our partitioning strategy is now clear:

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.

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

Algorithm 2 Prediction Error Modeling

Input: ratings: The ratings matrix
Input: rating_model: A standard user model
Output:
1. $errors \leftarrow [[]]$
2. **for all** $user, item, rating \in ratings$ **do**
3. $\quad errors_{user,item} \leftarrow |ratings_{user,item} - \text{Predict}(rating\_model, user, item)|$
4. **end for**
5. $error\_method \leftarrow \text{NewModelingMethod}(SVD)$
6. $error\_model \leftarrow \text{TrainModel}(error\_method, errors)$
7. **return** $error\_model$

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 recommender method 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.

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 of course 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, each modeling method is used to predict ratings, and their error models to predict errors. Note that each step in the first for-loop is independent of each other, and both steps inside the for loop is 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 $[0, 1]$ and changed to sum to 1. This is vital before last stage of the prediction

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:

1. $ratings \leftarrow \emptyset$
2. $errors \leftarrow \emptyset$
3. **for all** $m \in rating\_models$ **do**
4. $\quad ratings_m \leftarrow \text{Predict}(rating\_models_m, user, item)$
5. $\quad errors_m \leftarrow \text{Predict}(error\_models_m, user, item)$
6. **end for**
7. $errors \leftarrow \text{Normalize}(errors)$
8. $prediction \leftarrow 0$
9. **for all** $m \in rating\_models$ **do**
10. $\quad weight_m \leftarrow 1 - error_m$
11. $\quad prediction \leftarrow prediction + weight_m \times ratings_m$
12. **end for**
13. **return** *prediction*

algorithm, which weighs each prediction from the different rating models. This step corresponds to the previously explained reduce operation, that combines multiple scores into one final result. The weight of each method is computed as $1 - error$, where *error* is the normalized error for this method, for the current user and item. Each rating prediction is then weighted, 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 of course where performance is most important. Naturally, 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 speedy predictions are always at the users hands.

3.5 Rank Aggregation

Now that we have seen how to adaptively aggregate scalar scores, it is time to see how to do *adaptive rank aggregation*. Rank aggregation means combining sorted lists of items. In this scenario, each modeling method takes the current user as input, and produces a list of items ranked in order of rating (see 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 combination. 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 is still 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, p3)).

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 each 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 each 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.

The important difference is this: The IR methods 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*:

- 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 each as needed. We still have the adaptive layer that estimates how well each signal 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.

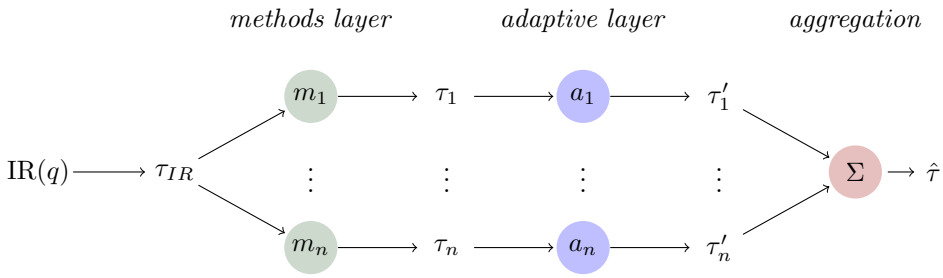


Figure 3.4: Example of Adaptive Rank Aggregation: An IR method returns a results list of possibly related items, each with a ranking score. The methods layer estimates ratings for each item in the results list. The adaptive layer predicts how accurate each of these ratings are likely to be. Finally,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

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 would be 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 previously developed algorithms.

The modeling phase for the recommender system stays 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 this log, we can create an implicit ratings matrix. Each populated cell represents a selected item. 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.
- 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 similar, metrics, we 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).

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, which is not our concern in this chapter.

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 each of the items in the results list. As before, the first for-loop can easily be performed in parallel. Each call to Predict is independent of the other operations, 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 each method, the errors are normalized across items. In other words, for each item, the errors from the recommenders fall in the range $[0, 1]$ and sum to 1.

After each item in the results list has 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 poses no 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

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:

1. $ratings \leftarrow \emptyset$
2. $errors \leftarrow \emptyset$
3. $results \leftarrow \text{Search}(ir\_model, items, query)$
4. **for all** $item \in results$ **do**
5. **for all** $m \in rating\_models$ **do**
6. $ratings_{m,item} \leftarrow \text{Predict}(rating\_models_m, user, item)$
7. $errors_{m,item} \leftarrow \text{Predict}(error\_models_m, user, item)$
8. **end for**
9. **end for**
10. $errors \leftarrow \text{Normalize}(errors)$
11. **for all** $item, ir\_score \in results$ **do**
12. $prediction \leftarrow w_{IR} \times ir\_score$
13. **for all** $m \in rating\_models$ **do**
14. $weight_{item} \leftarrow 1 - error_{m,item}$
15. $prediction \leftarrow prediction + weight_m \times ratings_{m,item}$
16. **end for**
17. $item_{prediction} \leftarrow prediction$
18. **end for**
19. $results \leftarrow \text{SortByPredictions}(results)$
20. **return** $results$

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 in adaptive rank aggregation.

Clearly, 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. As we have said before, algorithms are only as strong as the data they can leverage. In other words, methods for personalized search will 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.

— 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 next chapter will discuss implications and limitations of the results.

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	943	1,682	100,000

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 our technique 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, p9), Lemire and Maclachlan (2005, p4), Adomavicius and Tuzhilin (2005, p1) and Herlocker et al. (2004, p2)). 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 a subset of the entire MovieLens collection, with 100,000 ratings from 943 users on 1,682 movies.

The MovieLens dataset also comes with meta-data on each user, such as gender, age and occupation. There is also meta-data on each movie, such as its title, release date

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

and genre. In Experiment 1, we are only interested in the ratings matrix of this dataset. However, the titles of each movie 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, p14), Adomavicius and Tuzhilin (2005, p5) and Ahn and Hong (2004, p30)). This dataset is considerably larger than our first set, consisting of 1,832,275 ratings from 24,983 users of 100 items. Notably, the number of items is quite smaller than in the other dataset.

Jester also have a different ratings scale compared to the MovieLens dataset: while each movie is 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 evaluation results, both datasets were split into multiple disjoint subsets. We need disjoint subsets in order to perform cross-validation. 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:

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

Each $base_x$ and $test_x$ are disjoint 80% / 20% splits of the data in each subset. 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, each *base* set is further split using bootstrap aggregation, into random subsets for training each standard recommender model. The entire base set is then used to train the adaptive error estimating recommenders. Each corresponding *test* set will be used to evaluate our performance on each subset.

Before diving into each experiment, let us take a closer look at the different types of recommender systems each will use.

4.2 Recommenders

We need a number of recommender systems for each experiment. Standard recommenders will be used for both the basic predictions, and the accuracy estimations, as described in Chapter 3.

(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	knn	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 each of the recommender methods used in our experiment. Each recommender is used in every experiment. See Section 2.3 for more information.

Naturally, we need a large number of different recommenders, preferably ones that consider disjoint patterns in the data. Table 4.2 gives a short overview of the recommender systems we shall employ. Each experiment will use every recommender in this table. This section only gives a short introduction to each recommender. See Section 2.3 for more information, and Appendix A for details on how these were implemented in our 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. These recommenders each look at the data in different ways to arrive at predicted ratings. We chose this wide range of recommenders for just this reason: as previously explained, the performance of aggregate recommenders are more dependent on the dissimilarity of the basic recommenders than their individual performance.

Let us briefly explain how each basic recommender 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 2.3.4). 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.

The KNN 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 KNN 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 data patterns.

4.2.2 adaptive Recommenders

The second type of recommenders are the aggregation methods, that combine the result of each of the basic recommender systems (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, traditional 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 each basic recommender. All the aggregation approaches, including our technique, use 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 model, as seen in Listing 1. In other words, we have as many possibilities for choosing the adaptive recommenders as the basic recommenders.

For our experiment, we went with SVD recommenders for each of the adaptive models. That is, each basic recommender method gets a secondary accuracy predicting recommender, which in this case is a standard SVD recommender. The SVD recommender is a natural choice in this case, since we wish to uncover latent patterns of accuracy for each model. Examples of these patterns include groups of items or users a specific

recommender works well for.

It is important to note that the same configuration of recommenders was used for all three experiments. In other words, neither the basic nor the aggregate or adaptive recommenders were heavily tailored to each dataset. To be sure, higher performance could probably have been achieved by tailoring each recommender to each dataset. However, as our goal is to compare our finite set of methods, all we care about is how they perform compared to each other.

As with the basic recommenders, the same SVD recommender configuration was used for the adaptive layer in each 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 2.3.4 for more information on how SVD 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 predictions. 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, p17), Adomavicius and Tuzhilin (2005, p13) and Bell et al. (2007b, p6)). We shall use this measure to estimate the performance of our adaptive prediction aggregation algorithms. The RMSE of a set of estimations $\hat{R}$, compared the correct rating values R, is defined as

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

where 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. In other words, 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 the 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 soon see. Because of this, regarding hypothesis H3, we are most interested in examining how personalization with adaptive recommenders affects the initial rankings from an IR method.

4.4 Prediction Aggregation

Our first hypothesis, H1, states that: *the accuracy of relevance predictions can be improved through adaptive recommender aggregation.* The second hypothesis, H2, states: *adaptive aggregation can achieve higher accuracy than global and generalized aggregation methods.*

In order to verify these hypotheses, we performed adaptive prediction aggregation through adaptive recommenders on the two datasets previously described. 5-fold cross validation was performed to further verify each result.

Table 4.3 gives the results from Experiment 1 (MovieLens). Table 4.4 gives the results from Experiment 2 (Jester). Each cell corresponds to the RMSE values for each 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 each experiment are given in the last parts of Tables 4.3 & 4.4. The statistical values are the minimum, maximum and mean values for each of the methods. We also include the standard deviation (σ) for each method, across each collection 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 performance in Experiment 1, along with the 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 see, the performance of each of the aggregation methods, as well as the best performing standard recommender, follow each other closely. At the same time,

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	knn	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	knn	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): Each cell gives an RMSE value, on the scale 1-5. The first table gives errors for each subset of the data (d_x). Lower values indicate better results. Bold values indicate the best result in each 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	knn	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	knn	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): Each cell gives an RMSE value, on the scale 1-5. The first table gives errors for each subset of the data (d_x). Lower values indicate better results. Bold values indicate the best result in each column. S refers to singular methods, and A to aggregation methods. σ refers to the standard deviation of each method across the subsets.

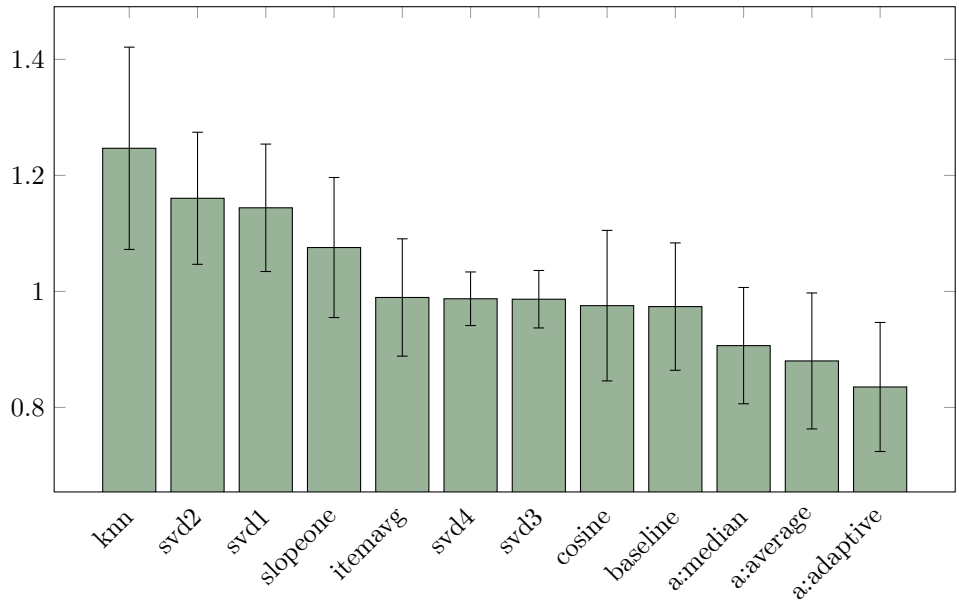


Figure 4.1: Average RMSE plot: This plot shows the average RMSE for each method, and each aggregation method (denoted "a:"). The actual numbers are given in Table 4.3. The error bars indicate the standard deviation of each method. Note the scale on the y-axis — the errors are not as pronounced as they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

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 each of 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 5-fold cross-validation, the results allow us to confirm hypothesis H1 in these conditions, and likely for other, similar scenarios. We shall discuss this 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, adaptive recommenders are adaptive in that each prediction is aggregated based on the current user and item, satisfying the language of H2.

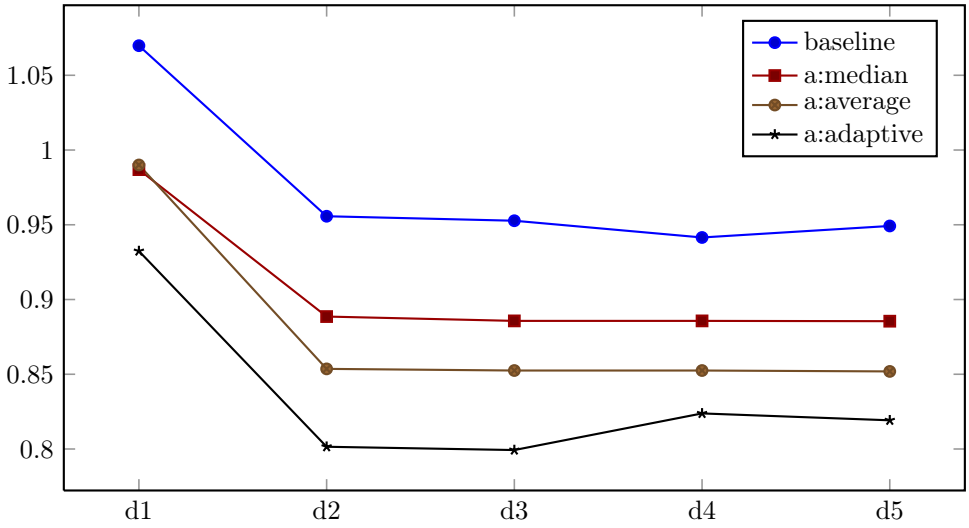


Figure 4.2: RMSE Variations: This plot shows that, while the standard deviation of each method may be high, this has more to do with the selected dataset than with their performance in comparison with each other. The performance of each of the aggregate methods, as well as the baseline standard method,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

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 combination 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 methods 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 second experiment and hypothesis H3.

4.5 Rank Aggregation

While the previous experiment was a quantitative exploration of RMSE values, this experiment will focus on qualitative traits of rank aggregation. In particular we wish to see how our method can perform personalized search.

The MovieLens dataset fit the needs of this experiment. Searching through movies is a scenario where the actual predicted rating of each movie would be a welcome signal for ranking the results. In other words, 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.

Hypothesis H3 states that *the result set from an information retrieval query can be adaptively personalized by layering recommenders.* We wish to check if the prediction algorithm in Listing 4 performs personalized search in a meaningful way.

There are many ways of determining the accuracy of a personalized search algorithm, such as the mean average precision of the results list (the mean of the precision average over a set of queries). However, 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, which we have no access to.

Of course, 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 shall take a look at four use cases with 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.

The IR weight is the ratio multiplied with the IR score before the adaptive recommender scores are incorporated (see Listing 4). The actual choice of weight depends on scale of scores returned by the IR method. 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.

In other words, 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 (1-5).

We will consider the following use cases: (1) searching multiple topics, (2) searching for a series of movies, (3) searching for one particular topic, and (4) searching for a particular attribute.

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

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: This example has a quite complex IR query (["new york" or washington*]). The top-most table shows the results returned by our IR model, defining the item-space for the following tables. The scores are the returned IR ranking scores. 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. The bottom table shows the final scores and positions when the IR scores and predicted ratings are combined. Only the top 10 results are shown.*

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. In other words, each score is considered an input signal, and each signal is weighted the same.*

(1) 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 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) 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 ranking for the query [*star trek*], which refers to a movie 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) 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. When we can be sure that each item 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

#	*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.*

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 each item in the list, based on how well they have performed for each item in the past.

(4) 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 can be said to be perfect answers to the algorithm – each movie was 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 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.

As we have seen in this section, adaptive recommenders can provide personalized search in multiple ways. Because of this, hypotheses H3 is confirmed for these use cases with our dataset. 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

#	*score*	*title*
1	2.7742	Fallen (1998)
2	2.7742	Sphere (1998)
3	2.7742	Phantoms (1998)
4	2.7742	Vermin (1998)
5	2.7742	Twilight (1998)
6	2.7742	Firestorm (1998)
7	2.7742	Palmetto (1998)
8	2.7742	The Mighty (1998)
9	2.7742	Senseless (1998)
10	2.7742	Everest (1998)

#	*rating*	*title*
1	3.8694	Apt Pupil (1998)
2	3.4805	The Wedding Singer (1998)
3	3.1314	Fallen (1998)
4	3.1225	Tainted (1998)
5	2.9442	Blues Brothers 2000 (1998)
6	2.9046	Sphere (1998)
7	2.8842	Desperate Measures (1998)
8	2.8798	Firestorm (1998)
9	2.8633	Vermin (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.

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. Naturally, this weight is a defining characteristic of any personalized search based on adaptive recommenders.

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 work well for personalized search. This positive result for Experiment 3 confirms hypothesis H3, at least for this dataset, this IR system and our chosen recommender algorithms.

The next chapter will discuss the implications and limitation of our results.

— 5 —
Discussion & Conclusion

This chapter will discuss the implications of our results. While our hypotheses may be answered, 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

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 each user and item *should* be modeled. We believe this selection should be left to each user: differing 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. However, 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 each user's personal opinion determines if these connections are used is desireable.

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. Each of 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 each of these algorithms may have varying levels of accuracy for each individual item. Adaptive recommenders can help solve this problem by adaptively combining the recommenders based on individual item performance.

Hypothesis H1 was confirmed by showing that adaptive recommenders can outperform standard single-approach recommenders. We achieved lower total RMSE scores across each of our datasets, which would imply that adaptive recommenders reliably performs better than our tested standard recommenders.

Hypothesis H2 was confirmed by showing that adaptive recommenders can outperform simple, generalized aggregation approaches. While our standard aggregators were

simple, this result is promising. However, the real test would be to use our approach in a situation with even more differing recommender systems.

Hypothesis H3 was confirmed by showing that our approach can be used to provide personalized search. While we did not strictly evaluate the quantitative performance of this approach, our results show that different prioritisations of the IR scores can cope with many different use cases. 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.

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.

5.2 Limitations

There are some important general limitations to our research related to the (1) complexity of our method, (2) our choice of data and evaluation metrics, (3) the general usefulness of this approach, and (4) common issues with recommender systems.

(1) Complexity: 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 each user might place varying importance on, and that may have varying success with each item, 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.

(2) Evaluation: we chose traditional datasets and evaluation metrics to validate the adaptive recommenders technique. While our initial results are promising, it is important to stress that this is only one test on one dataset. Considering the vast scope of applicable data, and the number of ways these results many be evaluated, the results must be seen for what they are: initial and preliminary explorations of a new technique that has yet to be proved useful in the real world.

As mentioned in Chapter 2, the scale of known ratings is another 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 each steps 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.

(3) Usefulness: 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 cases, each of the systems described in this thesis have their use cases. In the end, the requirements of each system must decide which method best suit their needs.

(5) Common issues: 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 each user 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.

While a thorough discussion of privacy and usability is outside our scope, they are both important limitations to considered when using a recommender system.

In addition to the general limitation, each of our experiments carries a few drawbacks.

Hypothesis H1 was only tested against a limited number of standard recommenders. The key word is standard: these recommenders were not heavily customized to fit the available data. As in all 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 only tested against simple aggregators. Many more complex aggregations are possible, for example by solving the problem of finding optimal generalized

weights for each method. While our tests show the basic viability of our approach, more testing against complex aggregation functions is still required.

Hypothesis 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.

5.3 Contributions

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 one 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 which works best is dependent on each user and item in 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 each 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). In other words, 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: by creating error models for each recommender, we can use this to predict its accuracy for each user/item combination. These predictions can then be used to weigh each combined algorithm accordingly.

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 each individual item. 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.

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 a few interesting research topics which should shed more light on the subject.

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. In other words, we wish to identify clusters of users and items for which each algorithm may or may not be suited.

However, 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 each user 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.

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.

However, 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 each user and item 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.

Multiple IR Models as Signals As mentioned in Chapter 4, we only tried rank aggregation in a scenario with one IR model. However, 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.

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 each method in an ensemble, we get a system capable of predicting the accuracy of each method. 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, p7), and other fields of ensemble learning.

5.5 Conclusion

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.

This thesis has explained the *latent subjectivity problem*, and introduced the technique of *adaptive recommenders* in an attempt to solve it. We believe this solution is one possible way to minimize the overload problem. Our technique implicitly, and without any extra work required from each user adapts how the system models users and items based on past performance. Our experiments show that this technique is capable of higher accuracy than standard recommenders and simple aggregation approaches.

Of course, we have only tested our method on 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 truly and internally adaptive user modeling 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: only by creating truly adaptive systems that adapt their algorithms to each user and item can we achieve a system powerful enough to deal with the widely varying users, and vast scope of items.

We believe many AI methods fail to gain wide acceptance by users because they lack this extra level of personalization. As stated through the latent subjectivity problem, systems should not only tell users what has been predicted, but also allow flexible and adaptive usage of its internal algorithms. *After all, a system that insists on being adaptive in one particular way is not really adaptive at all.*

Information is indeed a curious thing, and our only way of taming the never ending torrent of arriving data is to embrace the wide scope of the information and of the people who wish to consume it. Adaptive prediction aggregation takes us one step further towards this goal.

— A —
Implementation

This section 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 *adaptive recommenders* in itself, this should give a short introduction to how this technique can be put into practice.

A.1 Libraries

Naturally, 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 each experiment 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 kNN-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, each of our recommenders come 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

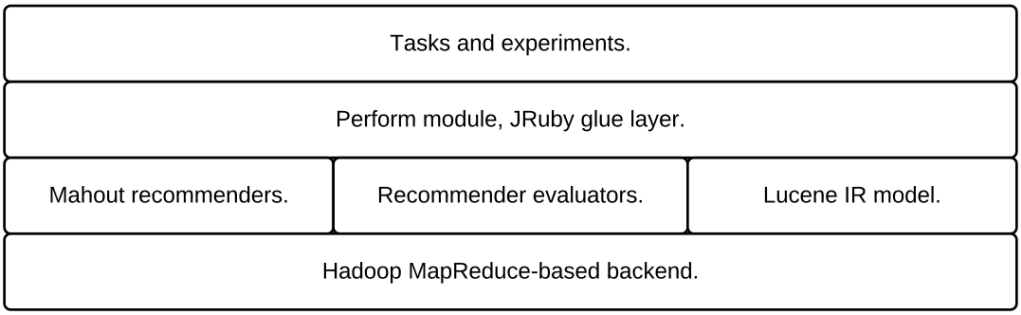


Figure A.1: Library layers: Tasks and experiments are performed by the custom JRuby glue layer. Recommenders are created by the Mahout machine learning library. IR models are based on the Lucene search engine library. Recommender evaluators are created in Ruby. Each of the intermediary layers are built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

using the JRuby implementation of Ruby, Java libraries can be imported directly into 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

In order to facilitate rapid prototyping, 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 and serves as instantiated configurations for 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

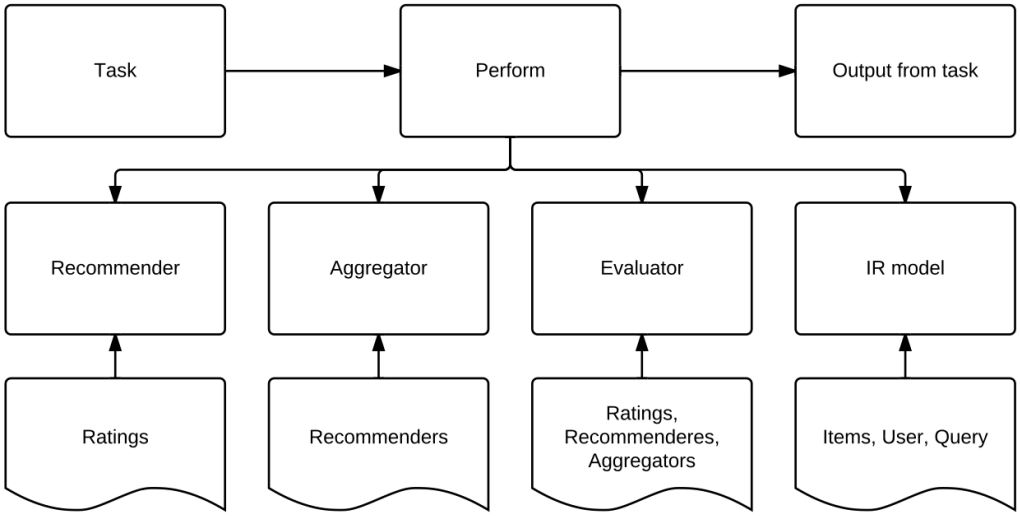


Figure A.2: Task structure diagram: A task (instantiated configuration) is passed to the perform module. This module creates a number of modules: recommenders, aggregators, evaluators and information retrieval models. Each module takes a set of inputs (bottom row), which are specified by the current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

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

Our experiments required us to measure the performance of each recommender, 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 each of the 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 each modeling method. 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 of course 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 code is written in JRuby, just as it is 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 = {
  knn:        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.

A.5 Running the Experiments

To tun 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 conformant to Ruby version 1.9 (e.g. JRuby 1.6). The *rake* library should also be installed, as it is used to start and run each experiment.

To get the system up and running in a Posix-based environment, the following steps should be run. The $ refers to the operator 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 each 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 experiment1.rb
```

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.

— B —
Resources

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

B.1 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.

B.2 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 Recommenders:
              Personalized Prediction Aggregation
              Through Accuracy Estimation}}
}
```

(1) See www.apache.org/ — accessed 19/5/2011
(2) See hadoop.apache.org/ — accessed 19/5/2011
(3) See mahout.apache.org/ — accessed 19/5/2011
(4) See lucene.apache.org/ — accessed 19/5/2011
(5) See www.apache.org/licenses/ — accessed 19/5/2011
(6) See www.jruby.org/ — accessed 09/05/2011
(7) See www.ctan.org/tex-archive/macros/latex/contrib/memoir/ — accessed 19/5/2011.
(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., Wu, X., Fei, B., Xue, G., Su, Z., and Yu, Y. (2007). Optimizing Web Search Using Social Annotations. *Distribution*, pages 501–510.

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.

Bjorkoy, 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.

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*. Pearson Education.

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*.

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. and Maes, P. (2006). Unraveling the taste fabric of social networks. *International Journal on Semantic Web and*.

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., and Schutze, H. (2008). *Introduction to information retrieval*. Camebridge University Press.

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. and Moore, C. (2000). Mean-field solution of the small-world network model. *Physical Review Letters*.

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. (1988). The design of everyday things. *Basic Book Inc./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.

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.

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). *Adaption as a Problem of Design*, pages 59—-84.

Ujjin, S. and Bentley, P. J. (2002). Learning user preferences using evolution.

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. (2008). Towards decentralized recommender systems. *ISBN 363901149X, Vdm-Verlag.*