ASSIGNMENT

Stacked User Modeling

Adaptive Model Aggregation
Through Accuracy Estimation

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.

Contents

List of Figures

List of Tables

— 1 —
Introduction

TODO: This chapter is just a collection of notes, the introduction has not yet been written.

In 1971, Herbert Simon said the following on the topic of information overload: "What information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention and a need to allocate that attention efficiently among the overabundance of information sources that might consume it." Greenberger (1971).

In 2009, Alon Halevy, Peter Norvig, and Fernando Pereira, wrote: "Perhaps when it comes to natural language processing and related fields, we're doomed to complex theories that will never have the elegance of physics equations. But if that's so, we should stop acting as if our goal is to author extremely elegant theories, and instead embrace complexity and make use of the best ally we have: the unreasonable effectiveness of data." Halevy and Norvig (2009).

Higher orders of training data: Some algs perform better than others, depending on amount of data. Banko and Brill (2001)

Simple algorithms, lots of data, many algorithms, user-combined.

Data-driven learning: http://archive.ecml.at/projects/voll/rationale_and_help/booklets/resources/menu_booklet_ddl.htm

"Predictive accuracy is substantially improved when blending multiple predictors. Our experience is that most efforts should be concentrated in deriving substantially different approaches, rather than refining a single technique. Consequently, our solution is an ensemble of many methods." — Bell et al. (2007b) Teams with different insights, each team votes on answers. Breakthrough in netflix prize challenge – race of ensembles.

Previously, something else was the problem

Today, biggest problems on the web

Information overload

Content discovery

Search

User modeling

Often generic methods

the modeling problem: model+prediction

the core problem: estimating preferences

getting past 80%

the efficiency of data

This paper: A more personal approach

Aggregated user modeling methods for truly personal predictions.

Hypothesis

Contributions

Outline

— 2 —
Background Theory

This chapter will introduce some basic theory needed to develop our approach to user modeling. We will first describe our stated enemy, the information overload problem, before delving into how user modeling and, more specifically, recommender systems, is currently used to solve this problem.

This chapter will also introduce the notion of personalized search, a field where our user modeling method will be especially applicable. The next chapter will use these theories to build an a method for *adaptive rank aggregation* called *stacked user modeling*.

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). Needness to say, this is a widespread phenomenon, with as many definitions as there are fields experiencing the problem. Examples include *sensory overload*, *cognitive overload* and *information anxiety* (Eppler and Mengis, 2004).

Two common tasks quickly become difficult in this situation: Consuming content that is known by the user to be relevant can be drowned out by irrelevant noise. Orthogonally, discovering new, interesting yet unknown content also becomes difficult because of the sheer amount of available content. 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 movies because those most relevant are never discovered.

The 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): "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) points to studies linking extreme cases to various psychological conditions related to stressful situations, lost attention span, increased distractibility 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). To evade information overload is then to maximize available attention, allowing more focus on the most important items of an interface.

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, describing 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.

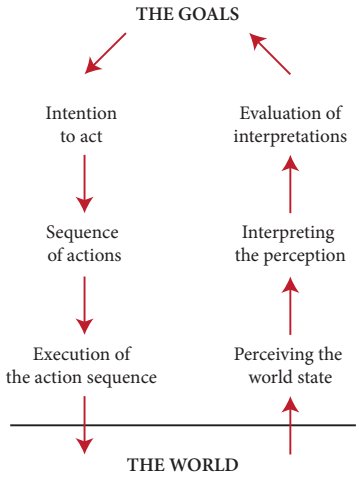


Figure 2.1: Stages of Action

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.

2.1.1 Online Overload

The Web is a common source of information overload, and presents a good example of when and how the problem occurs.

Online information overload is especially pervasive when considering *content aggregating websites*, i.e. sites that collate information from multiple other sites and sources. Online information retrieval (i.e. search engines), fall into this category, as does online newspapers, feed readers and portal websites. As mentioned, the wealth and scope of data are natural culprits of online overload, as well as the varying qualities of websites publishing the information.

Graph theory presents applicable models of the Web that characterize how people navigate between websites, and show how content aggregators form important hubs in the network. These models also show 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), 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 webpage 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 different heuristics to evaluate each link, often resulting in a quite good path to where we want to go. So why is the Web still quite challenging to navigate?

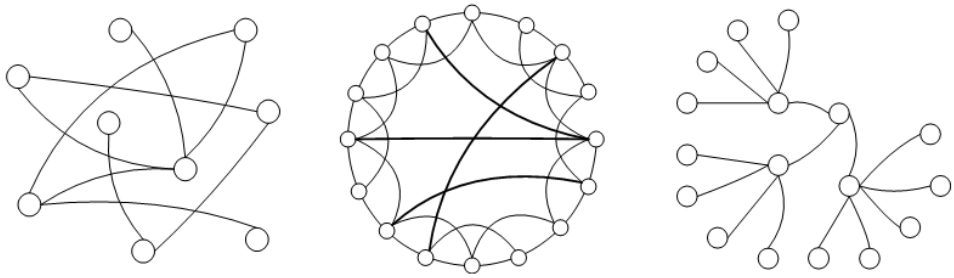


Figure 2.2: Complex Networks, from the left: A random *network, a* small-world *network and a* scale-free *network (which is a type of small-world network). Figure adapted from Huang et al. (2005).*

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 $\prod k_i$ in Equation 2.1, where k_i is the number of links pointing to node i.

$$\prod_i k_i = \frac{k_i}{\sum_j^N k_j} \tag{2.1}$$

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. Factor in the existence of multiple sub-graphs, or continents, and we can intuitively see that navigating the Web is not as easy as it might appear from simple models.

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.

So far we have established that information overload is a pervasive problem, especially on the web. The question now becomes how to best solve this issue. This is where user modeling comes in.

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 and how to implement these methods, 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 layout of interface control elements (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.

We are interested in adapting the content of an application since the source of our

overload problem often comes down to a mismatch between presented content and desired content. Examples of such user modeling include:

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

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), 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. Luckily, according to Kobsa (2001), recent research has blurred the lines between the AI and HCI in user modeling.

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 paper is concerned with the AI approach to user modeling, and in particular, the use of *recommender systems (RS)*. As our goal is to combine different RSs into one coherent

user model, we will now describe what an RS entails, and introduce some of the many algorithms they employ.

2.3 Recommender Systems

The name might seem constraining, but 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 products like books or movies 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.

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 anything: Documents, movies, music, places, people, or indeed other users. A recommender system is best described by graph and graph operations, even though the underlying algorithms might not use this as the representation. Mirza and Keller (2003) explains 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 strenghts of hypothetical connections between nodes that are not explicitly connected.

For example, if a user has rated some of the movies in a movie recommendation system, we use these ratings to predict how well the user will like unseen movies, based on a movies ratings from users similar to the one in question. 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. Recommender systems are then powerful methods for user modeling, personalization and fighting information overload, because of their ability to infer how relevant and item (or another user) will be to the current user.

Formally, a recommender system can be seen as a quintuple, $\mathrm{RS} = (I, U, R, F, M)$, 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 connections, for example explicit preferences given by users for certain items, or connections in a social graph. F is a framework for representing the items, users and ratings, for example a graph or matrix. M is the actual user modeling method 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), M is seen as a utility function $f : U \times I \rightarrow S$. Here, f is a function that maps the set of users and items into a fully ordered set of items S, ranked by their utility (i.e. rating) 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:

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

The utility function u 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 u 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 u 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 rated themselves.

Another popular way of describing, and implementing an RS is using a simple matrix. Here, one dimension represents users, the other dimension represents items, and each cell corresponds to an explicit 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 (i.e. most of the cells are empty). Consider that while there may be a large number of users and items, each individual user only rates or connects to a few number of items. 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, the recommender system must be able to produce results from a matrix where only 1% of the cells have meaningful values.

Naturally, this is the defining characteristic of many recommender systems: the ability to extract meaningful patterns from sparse data, through dimensionality reduction, neighborhood estimation and other methods, as we shall see.

Recommender systems face many challenges other than the 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, 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. 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 continous scale.

2.3.1 Predicting Ratings

The crucial part of any RS is how it predicts unknown ratings. (Note that altough we use "ratings", "utility", "preference", "relevance" and "connection strength" depending on the context, they all basically mean the same.) Because of this, each method is best categorized based on a few dimensions of its predictive capabilities (see Table 2.1). In our taxonomy, these dimensions are: *data*, *method*, *granularity*, *temporality* and *agents*.

The *data* variable represents what data the RS uses to perform predictions. Content-based methods use only the items, inter-item relations, and an individual user's past history as predictive of future actions (Pazzani and Billsus, 2007). 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). The problem of having to do complex

inference from little data, as is often is in content-based predictions, is often called the *sparsity problem* or the *cold start* problem. This is closely related to the problem of *overfitting* data, where the algorithms creates models that match the training data, but not the actual underlying relationships. A lot of research looks at ways to overcome sparse data, i.e. achieving "warmer" cold start. When using content-based predictions, the utility function $f(u, i)$ of user u and item i is extrapolated from $f(u, i_u)$, where i is an item similar to i_u and $f(u, i_u)$ is known.

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 $f(u, i)$ of item i for user u is extrapolated from $f(u_j, i)$ where u_j is a user similar to u.

Because of *the new user problem* of content-based prediction and the *average user problem* of collaborative prediction, many systems use a hybrid approach (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. For example, creating a Bayesian networks from past interaction is a model-based approach. The other category is the *heuristic* or *memory-based*

Variable	*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).

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

The *granularity* variable tells whether this approach creates models for the canonical user, stereotypical users or individual users. 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 sterotypes. Each stereotype has a set of *facets* which is then used to match books and users.

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))

2.3.2 *Examples of Recommender Systems*

Because our solution will combine different recommender systems, we need a short introduction to some of the approaches we will combine. 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 next chapter. 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.

(1) *Baseline ratings* are the simplest family of recommender systems, based on item and user rating averages. The data is content-based, and used to compute heuristic predictions. This is done on a per-user, individual basis and collects long term knowledge

so far as the user is rational in most of his or her ratings. 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 system rating, 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 assesments, 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.

Another approach based on simple averages is the *Slope One* family of collaborative filtering 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,

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

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

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

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

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

The limited neighborhood size becomes the statistical sampling size, and is a useful way of placing an upper bound on the complexity of computing a neighborhood. n does not have to be a raondom 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. Similarity metrics from information retrieval, such as *cosine correlation* from the *vector space model* (VSM) are also popular, often 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 neirest 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 introduce.

(3) *Dimensionaliy reduction* is an oft-used technique when creating recommender systems. The ratings matrix is factored into a set of lower dimension matrixes, 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* (LSI) in information retrieval

(Baeza-Yates and Ribeiro-Neto, 1999, p44). Formally, SVD is the factorization $M = U\Sigma V^*$. 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^* (the conjugate transpose of V) is an $n \times n$ factor ("stretcher"). Σ is a $m \times n$ diagonal matrix ("aligner").

The dimensionalty of the ratings space is performed by truncating the factor matrices each to a number of rows or columns, where the number is a paramenter depending on the current domain and data. 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 k dimensions, the result looks like this:

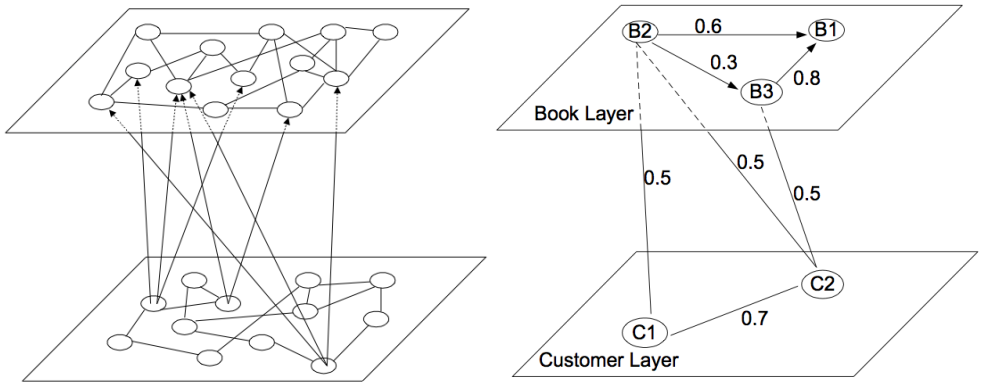
$$\begin{bmatrix} \\ R_{m,n} \\ \\ \end{bmatrix} \Rightarrow \begin{bmatrix} \\ U_{m,k} \\ \\ \end{bmatrix} \begin{bmatrix} \Sigma_{k,k} \end{bmatrix} \begin{bmatrix} \quad V^*_{k,n} \quad \end{bmatrix}$$

Two important transformations happen in this 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.

There are many ways of using the reduced factors. We can for instance use the resulting reduced factors to find similar users by their cosine similarity (see Section 2.4). 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 asymetrical (directed edges) symetrical (undirected edges) relations, or linking items based on some similarity metric.

Huang et al. (2002) used network traversal to create a simple system for recommending



(a) *(b)*

Figure 2.3: Network Traversal: (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).

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 2.3).

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 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 $\mathbf{p}^{(t)}$ stores the long term probability rates of each node, where $\mathbf{p}_i^{(t)}$ represents the probability that the random walk at step t is at node i. $\mathbf{S}$ is the column-normalized adjacency-matrix of the graph, i.e. the transition probability table. $\mathbf{q}$ is a column vector of zeroes with a value of 1 at the starting node (that is, $\mathbf{q}_i$ 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

$$\mathbf{p}^{(t+1)} = (1 - \alpha)\mathbf{S}\mathbf{p}^{(t)} + \alpha\mathbf{q}$$

when it is run to convergence (within a small delta). Then, the *relatedness* of nodes x and y is given by $\mathbf{p}_y$ 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 user modeling and recommender system. In both situations, we wish to predict how relevant an 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) show the user items similar to another item, and (2) allow discovery of relevant items the user did not know exist. The purpose of *search* is a bit different: 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 in this chapter, the two fields employ a lot of the same 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 to no describing structure, and the queries are short user-initiated descriptions of an *information need* (Baeza-Yates and Ribeiro-Neto, 1999, p19). Among other domains, this model describes web search engines, where the documents are web pages and queries are short sentences or a few 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 each document that each term appears in. 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)
- Lemmatization (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 masures 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 succintly define what a search engine must to to be successfull: retrieve many relevant documents and few irrelevant documents. Failing this test is to neglect the main purpose of an IR: preventing information overload by allowing people efficient access to relevant parts of an otherwise overwhelming information repository.

To us, however, the most interesting part of any IR system is the *ranking function*. This function maps queries to documents by a scalar score, signifying how well a match each document is to a query. The relation to recommender systems should be self-evident, and indeed, IR systems use many of the same metrics to measure query/document similarity.

A common framework for storing and ranking documents is the previously mentioned 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 of the 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):

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

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. $freq_{t,d}$ 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 successfull probabilistic weighting algorithms (Robertson, 2010)).

There are as many IR models as there are domains that need search, and even the basic vector space model can be constructed in a myriad of ways. There is also the simpler *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 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 similar traditional metrics may form the foundation of modern search engines, a lot more thought go into the final results. Different types of rankings 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 *reranking* or *rescoring* 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, another popular search engine, 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 typset words.

Signals can also depend on the query. Manning et al. (2008, p145) describes a system that

(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

takes multi-word queries, breaks them up into different permutations and 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 percieved 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 weighted 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:

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

where B_x is the set of pages linking to page x, and $\mathrm{Links}(y)$ 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 (PS) attempts to solve these problems by introducing individually catered 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 be "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 algoriithm*.

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:

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

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) aggregator (Xu et al., 2008, p3), which is nothing more than a weighted combination of the rankings:

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

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:

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

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:

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

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* (AM) 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 masured 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 the 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 preictions, the dimensionality reduction finds "regional effects", major structural patterns in the data. As they say: "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 are aggregators. This is mostly a question of semantics and implementation. Burke (2007, p4) 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 approaches to model aggretation (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 combinining predicted scores one-by-one, creating an aggregated result for each element that should be evaulated. 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 metasearch algorithms based on the Borda and Bayesian models usually outperform the best input system and are competitive with, and often outperform, existing metasearch 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 dabbeled 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-performes 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 lables, 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.

This chapter has introduced the basic theory we will need to develop our take on user modeling. The next chapter will build this model, While Chapter 4 presents our

evaluation experiments.

— 3 —
Methods & Implementation

This chapter will build our approach to user modeling. We will first explain why a new approach is needed, and outline a few hypotheses that will test our reasoning. We shall then develop *stacked user modeling*, a novel approach to adaptive user model aggregation. The last parts of this chapter shows how this new approach can perform prediction aggregation in a recommendation scenario, and rank aggregation in an information retrieval 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, 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.

However, both simple predictions and aggregate predictions have a fundamental problem. There exists an underlying, misplaced subjectivity to relevance prediction that is seldom discussed. When a method is developed or selected for use in a prediction system, a concious descision of which approach to use is made. The researcher or developer select which method (or methods in the case of aggregation), they think should be able to model users of the system. Consider the following relevance judgements:

- 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.
- When providing personalized search results, one ranking signal may be the social connections of the current user. Items deemed relevant by the user's peers will then recieve a boosted ranking (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 defines their 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, *latent subjectivity* is not desirable.

For example, While one user might appreciate a 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. The exact differences are not important — what is important is that they exist.

Another way of explaining latent subjectivity is that *user modeling methods are dependent on the subjective assumptions of their creators*. In other words, each modeling method use some aspect of available data to make predictions, and the importance of each of these methods is determined by whoever creates the system, or by the on average best error-minimizing combination, not by each individual user.

Aggregate modeling methods face the same problem of misplaced subjectivity: Aggregation is done on a generalized, global level, where each user 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.

We propose a method where these descisions are left to each user, providing an extra level of abstraction and personalization. This leaves the subjective nature of modeling method selection where it should be: In the hands of each individual user. If each method is *only used* based on how well it performs for each user, any possibly applicable user modeling method suddenly becomes a worthy addition. 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?

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 method* that *some users*

might find helpful. 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.

The issue of latent subjectivity can also cause problems when considering the scope of items. Should we really use the same metrics for judging every item's relevance score? Needless to say, items are often quite different from another, along a myriad of dimensions. Consider the World Wide Web: If each webpage 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. We are left with the same two questions as before:

1. What combination of which methods will accurately model every item?
2. Which methods could possibly help predict a score for an item?

As before, an approach where we only need to consider the second question is desireable. 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 issue of latent subjectivity. 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. We will now express our goals as a three hypotheses.

3.2 Hypotheses

To minimize any latent subjectivity, we would like a method for prediction aggregation that works on a per-user and per-item level (*per-element*). By automatically adapting each modeling method based on which user and which item is currently being considered, we should be able to produce an improved result. This paper will consider three hypotheses (H1-H3) to establish whether adaptive aggregation through stacked user modeling is a viable technique:

H1*: The accuracy of relevance predictions can be improved by blending multiple modeling methods on a per-user and per-item basis.*

It stands to reason that if a recommender system is indeed impaired by the subjective selections of modeling methods, an adaptive blend of these methods should outperform each of the individual approaches. Second:

H2*: An adaptive aggregation method can outperform global and generalized blending methods.*

If our assumption that model aggregation inherits the subjective nature of its chosen parts, a per-user aggregation without such misplaced subjectivity should outperform a generalized blending. Third:

H3*: The result set from an information retrieval query can be personalized by stacked user modeling, where the retrieval scores are considered standard input signals.*

As described in Section 2.4.2, every measure that influence the final ranking of results from a search engine can be seen as individual signals. Each signal, be it an IR score, something like PageRank or the result of a user modeling method, contributes to the final ranking.

This fits well into our thinking that multiple modeling methods will leverage multiple patterns in the data. The information retrieval ranking function becomes another metric, just like the user modeling methods, allowing us to do IR and personalization with the same algorithm.

3.3 Stacked User Modeling

Stacked User Modeling (SUM) is a way of combining recommender systems in an effort to answer two related 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 stacked user modeling as *adapting a set of user modeling methods with another complementary set of user modeling methods* (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 SUM can use recommender systems for both these tasks, as we shall soon see. A system for stacked user modeling is specified by a 6-tuple:

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

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 $i \in I$. These 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 later in this 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 rating:

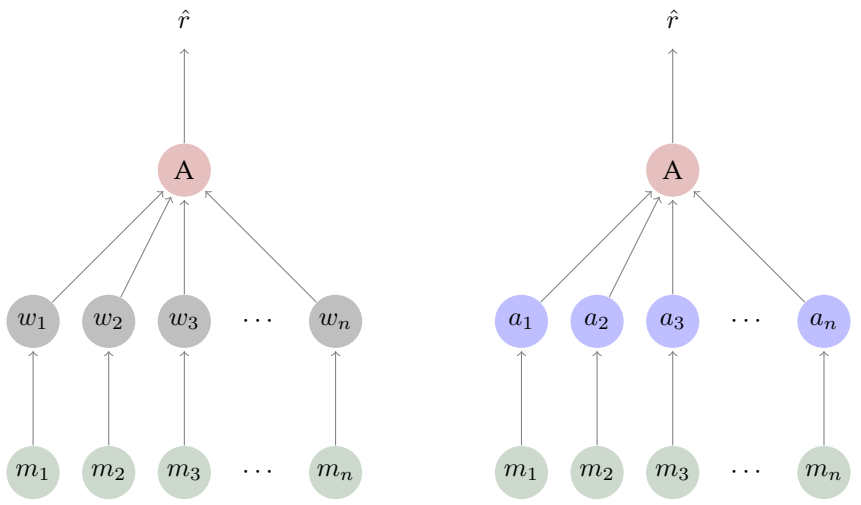


Figure 3.1: Comparison of aggregation and adaption: (left) modern aggregation approaches uses a set of pretrained weights to prioritize each modeling method. The weighted predictions are aggregated into a final prediction $\hat{r}$. (right) Stacked user modeling employs secondary modeling methods instead 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 are dealing with multiple approaches to user modeling, we also have a set of modeling $Methods$, that each create their own user models. Each model $m \in M$ are used to compute predictions. In our case, these methods are recommendation systems. As demonstrated in Chapter 2, there are many different recommendation algorithms, that consider differents aspects of available 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 2.3.2). These methods predict unkown 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

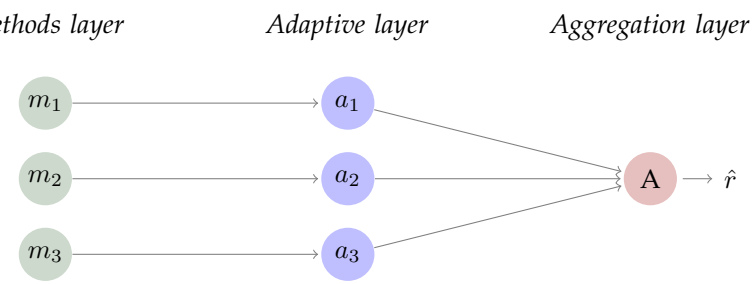


Figure 3.2: Stacked user modeling: The method layer consists of three ordinary modeling methods, each predicting the rating between a user and an item. The adaptive layer estimates how well each modeling method will perform for the current user and item, and weighs the predictions accordingly. The aggregation layer combines these weighted combinations into one final score.

methods that look at disjoint patterns, i.e. complementary predictive parts of the data (see Section 2.5).

The $Adapters$ part of our 6-tuple refers to the second level of user modeling methods. In traditional prediction aggregation this is a single function for combining the different predictions, for example by precomputing 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.

In stacked user modeling, the $Adapters$ are themselves user modeling methods (see Figure 3.1). These methods predict how accurate each of their corresponding recommender methods will be. This 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 measure to produce predictions. When presented with an item and a user, these methods produce a predicted rating $\hat{r}_{ui}$ based on an algorithm.

2. *The adaptive layer* is another set of corresponding modeling methods, that work a bit different. These methods take an item and a user and estimates how well its underlying method will perform this prediction. These estimations are then

used to create a combined prediction score by the aggregation layer. Each of these adaptive methods do not have to employ the same algorithm as their corresponding methods, they are only similar in that both are recommender systems.

Another way of describing (and implementing) the two modeling levels is through application of the map and reduce functions of functional programming. When performing *prediction aggregation* (scores), this estimation can be expressed as

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

First, each modeling method is applied through the map function, with the current user and item for which a rating should be estimated. This operation returns a set of scalar prediction values. These values are then combined in the reduce method, which takes the predictions and current user as input. In our case, this is the set of adaptive recommender methods. If we wish to do rank aggregation (i.e. sorted lists), the equation is a bit different:

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

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.

Expressing ourselves in terms of map and reduce now is helpful, as this will later guide our implementation of these operations in a proper MapReduce framework for parallell computation (as explained in Manning et al. (2008, p75)). The map function may apply each modeling method in parallell, as these are independent computations. The modified reduce function, which takes the resulting predictions map and the current user as inputs, serve as our adaptive layer.

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 avereage of all predictions made by the different methods (e.g. Aslam and Montague (2001, p3)):

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

Here, $\hat{r}_{ui}$ 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, most recommender aggregators attempt to weigh each method differently (e.g. Claypool et al. (1999)):

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

Here, w_m is the weight applied to modeling method m. These weights fall in the range $[0, 1]$ and sum up to 1. As described in 2.5, these weights can be estimated through different machine learning methods. Most often, the ratings data is divided into two sets: A training set for training the recommender systems, and a testing set for finding the weights that produce the optimal combination of predictors. However, as discussed in Section 3.1, this is still a generalized result, averaged across every user. The system assumes that the best average result is the best result for each individual user.

In order to leverage as many data patterns as possible, and remove the latent subjectivity, we need *adaptive weights* that are computed spcifically for each combination of a user and an item. However, just as we do not know the rating of every combination, we do not know the optimal weight. In other words, these adaptive weights have to be estimated, just as the ratings themselves:

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

We have now reduced our mission to creating a method that can estimate the adaptive weights: $p_w(m, u, i)$. As mentioned, we wish to use the same algorithms for both of these predictions. For recommender systems to be applicable here, we need to create a matrix (or graph) that stores known estimations of how accurate some of the predictions will be.

The key insight is that *the accuracy of a method is the inverse 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). Consider the following *error matrix*:

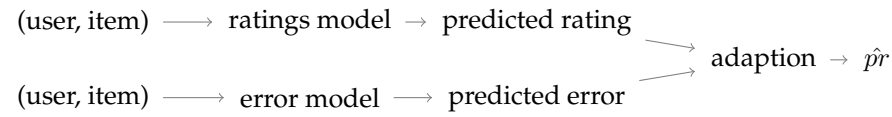


Figure 3.3: Adaptive weights: The data flow through the adaption of a single recommender method. The current user and item is fed into two distinct models: the ratings model, which predicts unknown ratings, and the error model, which predicts how accurate this rating will be for the current input. The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

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

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. 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. Notice how similar this matrix is to the previously introduced ratings matrix. This similarity is what will allow us using the exact same modeling methods 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 a meta modeling method (error model) based on the error matrix.

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. Normalize the errors so that the error vector sums to 1.
4. Compute a weighted combination where each weight is $1 - e_{m,u,i}$.

The next section will explain these steps in detail. For now, we have our approach to stacked user modeling:

$$\hat{r}_{ui} = \sum_{(m_e, m_r) \in M} (1 - p(m_e, u, i)) \cdot p(m_r, u, i),$$

where p returns a prediction from a model for a specific user and item, m_r is a standard recommender model, and m_e is its corresponding error model. For this equation to work as expected, the weights must be normalized:

$$0 \leq p_e(m, u, i) \leq 1 \quad \text{and} \quad \sum_{m \in M} (p_e(m, u, i)) = 1.$$

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 modeling method to perform adaptive aggregation. Hence, the name *stacked user modeling*: each standard modeling method is stacked under another accuracy estimating modeling 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 Stacked 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 stacked user modeling:

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).

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

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.

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 strutures 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

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** $error\_model$

first create the aforementioned error matrix by estimating predictions for each known combination in the ratings data. The NewModelingMethod call simply creates a new, untrained recommender model of some prespecified $type$ (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 a set of stacks, 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 stacks 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.

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. $\quad ratings \leftarrow \text{Predict}(rating\_models_m, user, item)$
5. $\quad errors \leftarrow \text{Predict}(error\_models_m, user, item)$
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. $\quad prediction \leftarrow prediction + weight_m \cdot ratings_m$
12. **end for**
13. **return** $prediction$

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, 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 Stacked 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 aggregtion, 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 ach 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.

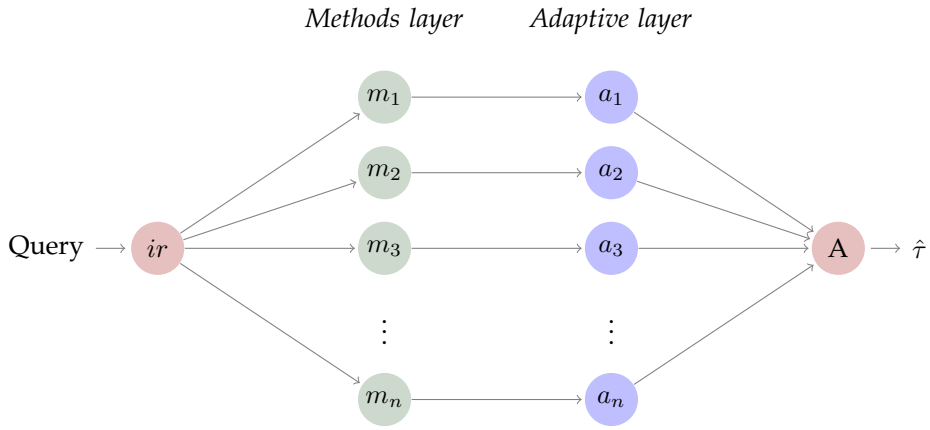


Figure 3.4: Stacked Rank Aggregation: An IR method returns a results list of possibly related items, each with a ranking score. The methods layer estimates ratings for each item in the results list. The adaptive layer predicts how accurate ach of these ratings are likely to be. Finally, the ranking scores,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

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, rescoring 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 preictions for each signal, as seen in Figure 3.4. Let us now see how the modeling and prediction phases are performed in adaptive rank aggregation.

3.5.1 Modeling Phase

We shall only deal with settings where we have a single IR method. While multiple IR methods and corresponding error models 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.

The values in this implict 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. 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.

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

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. $prediction \leftarrow ir\_score$
13. **for all** $m \in rating\_models$ **do**
14. $weight_{item} \leftarrow 1 - error_{m,item}$
15. $prediction \leftarrow prediction + weight_m \cdot ratings_{m,item}$
16. **end for**
17. $item_{prediction} \leftarrow prediction$
18. **end for**
19. $results \leftarrow \text{SortByPredictions}(results)$
20. **return** $results$

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 peformed 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 substract 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.

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.

In this chapter, we have seen how to perform stacked user modeling. It is now time to see how this model performs. The next chapter will test the viability of our model and answer our three hypotheses.

— 4 —
Experiments & Results

This chapter will perfom experiments to find out whether or not *stacked user modeling* is a viable technique. We will first explain which evaluation metrics will be used, and the datasets we will use to test the technique. Then, we will test each of hypotheses given in Section 3.2. The next chapter will discuss the implications of these results.

4.1 Evaluation Metrics

To evaluate how our model performs, we need a measure for computing the total error across a large number of predictions. The canonical measure for estimating the error of a predictions from a recommender system is the *Root mean squared error* (RMSE) measure (e.g. 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 to a set of known ratings R, is defined as

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

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. In other words, 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.

While the RMSE works well for evaluating scalar predictions, we need another measure for considering a predicted sorting from rank estimation 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. Because of this, we are interested in examining how personalization with stacked user modeling affects the rankings from an IR method.

4.2 Datasets & Recommenders

To test our model, we need an applicable dataset with users, items and ratings. We chose the MovieLens dataset[1] of movie ratings. This dataset is often used to test the performance of recommender systems, for example in Alshamri and Bharadwaj (2008, p9), Lemire and Maclachlan (2005, p4), Adomavicius and Tuzhilin (2005, p1) and Herlocker et al. (2004, p2). The dataset consits of a set of users, a set of movies, and a set of movie ratings on the scale 1 through 5, and is available in two sizes:

- A set of 100,000 ratings from 943 users on 1,682 movies.
- A set of 100,000,209 ratings from 6,040 users of 3,900 movies.

Each collection 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 and genre. For prediction aggregation, we are only interested in the ratings matrix extracted from this dataset. The titles of each movie will be used to experiment with personalized search.

To achieve reliable evaluation results, the dataset should be split into multiple disjoint subsets, so that we can do cross-validation. This entails running the same experiments across all the subsets, and averaging the results. The MovieLens dataset comes with a preset number of splits for this kind of testing. In the set with 100,000 ratings, the data is also split into five disjoint subsets, which are again split into training and testing sets:

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

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.

In addition to this dataset, we need a number of recommenders, that can predict unknown ratings between users and items. As we have seen, standard recommenders will be used for both the basic predictions, and for the accuracy estimations for each basic prediction. Naturally, we need a number of different recommenders that consider disjoint patterns in the data. Table 4.1 gives a short overview of the recommender systems we shall employ. See Section 2.3 for more information on the different types of recommenders,

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

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

Table 4.1: Stacked modeling methods: A short description of each of the recommender methods used in our experiment. See Section 2.3 for more information. For the SVD methods, the factorizers refers to algoriithms used to factorize the ratings matrix. 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.

and Appendix A for information on how these were implemented in the following experiments.

The important part is that each of these methods produce predictions in different ways. They look at different aspects of the data to arrive at each of their predicted ratings. As seen in Table 4.1, we have two types of methods: The first type of methods are standard recommender systems. The second type are the aggregation recommenders, that combine the result of each of the standard recommender systems. In addition to our stacked user modeling method (denoted with the key *stacked* in Table 4.1), we have the median- and average-based aggregators. While not complex in nature, these methods will help us see how our method compares to simple, traditional aggregation techniques.

To answer our three hypotheses, we have performed two experiments. The first experiment evaluates our method when used for adaptive prediction aggregation, comparing it to the methods given in Table 4.1. This will help us answer hypotheses H1 and H2. The second experiment will evaluate our performance in adaptive rank aggregation, in order to answer hypothesis H3.

	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	stacked	**0.9324**	**0.8015**	**0.7993**	**0.8238**	**0.8192**

Table 4.2: Results from Experiment 1: Each cell gives an RMSE value for a method on a subset (d_x) of our dataset. As these are error measures, on the same scale as our predictions, lower values indicate better results. Bold values indicate the best results for each dataset.

	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	stacked	**0.7993**	**0.9324**	**0.8352**	0.2225

Table 4.3: Statistics from Experiment 1: This table shows the minimum, maximum and mean RMSE for each of the methods in Table 4.2. The last column shows the standard deviation (σ) of each method. As before, lower values indicate better results, and bold values represent the best values in each column.

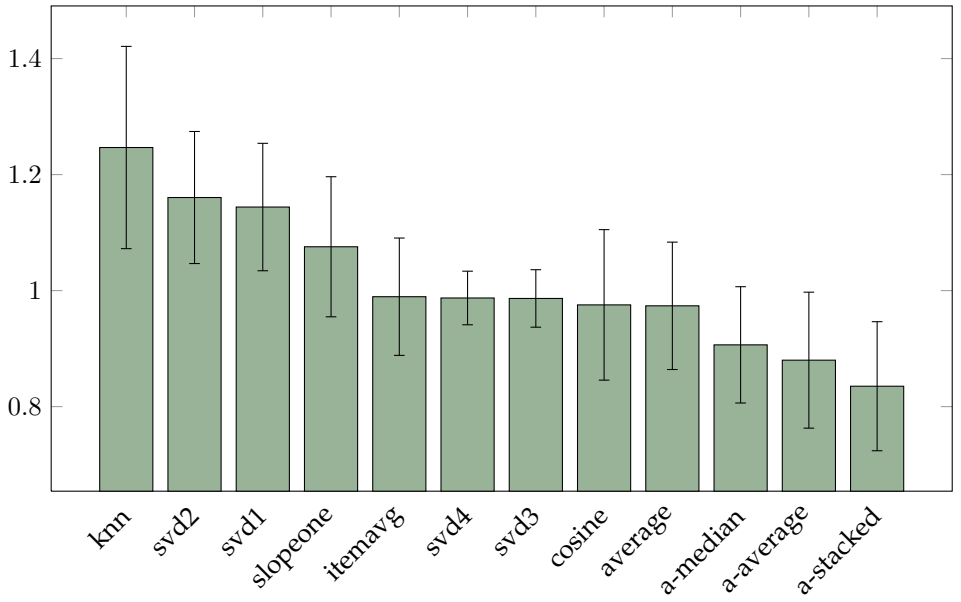


Figure 4.1: Average RMSE plot: This plot shows the average RMSE for each method, and each aggregation method (denoted "a-"). The actual numbers are given in Table 4.3. The error bars indicate the standard deviation of each method. See also Figure 4.2.

4.3 Adaptive Prediction Aggregation

H1

H2

4.4 Adaptive Rank Aggregation

Test H3

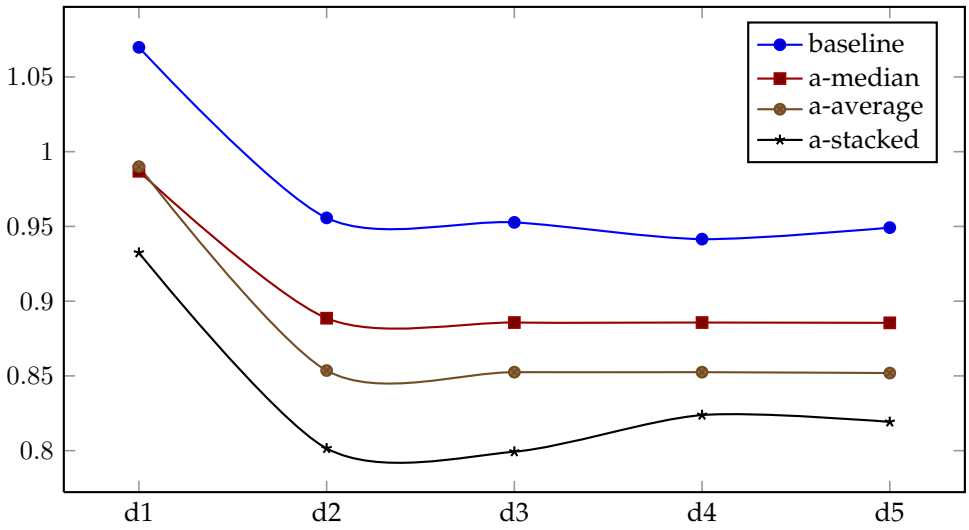


Figure 4.2: RMSE Variations: This plot shows that, while the standard deviation of each method may be high, this has more to do with the selected dataset than with their performance in comparison with each other. The performance of each of the aggregate methods, as well as the best performing standard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

— 5 —

Discussion & Conclusion

5.1 Discussion

5.1.1 Results Analysis

5.1.2 Conceptual Findings

5.2 Contributions

5.2.1 Findings

5.2.2 Answering Hypotheses

5.3 Future Work

5.3.1 Complex Individual Functions

5.3.2 Applicability of Different Methods

5.3.3 ...

5.4 Conclusion

— A —
Implementation

This section describes how the system outlined above was implemented, in order to perform the experiments of the next chapter. 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 *stacked user modeling* 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.

To quickly get a large number of recommenders up and running, the system was linked with the *Apache Mahout machine learning library*[1]. Apache 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[2]. This is important to the performance of our system. As mentioned, a lot of the operations performed in stacked user modeling 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*[3] is an open-source

(1) See http://mahout.apache.org — accessed 09/05/2011
(2) See http://hadoop.apache.org/ — accessed 09/05/2011
(3) See http://lucene.apache.org/ — accessed 09/05/2011

search engine, also built on top of Hadoop, gaining the same performance wins as Mahout. Lucene provides powerful methods for creating indexes of items, and for querying these indexes.

Mahout, Lucene and Hadoop are all written in the Java Programming Language, and runs on the Java Virtual Machine (JVM). To facilitate rapid prototyping, the Ruby scripting language was chosen as a "glue" language, for interfacing with the libraries. By using the JRuby[4] 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 sytem, which carries them out. Tasks specify what the system should do, which dataset should be used, and other options. See Appendix A for an example.

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 and Prediction

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

(4) See http://www.jruby.org/ — accessed 09/05/2011

As we shall soon explain, our experiments require us to measure the performance of each recommender, and the stacked user modeling 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.

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

— B —

Resources

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.

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.

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.

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.

Greenberger, M. (1971). *Computers, communications, and the public interest*. Johns Hopkins University Press.

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.

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.

Kobsa, A. (2001). Generic User Modeling Systems.

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.

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

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

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, 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*.

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.

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.

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

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.

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.

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.