UPTEC STS 11 013
Examensarbete 30 hp
Mars 2011

Collaborative Filtering on Familjeliv.se

Christian Wennerström

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Collaborative Filtering on Familjeliv.se

Christian Wennerström

The purpose of this project was the improvement of navigation on the forum site Familjeliv.se. The forum contains over two million threads, and site management wanted a system to recommend threads to users, alleviating the problems with navigating all the data. The choice fell on a thread-to-thread similarity based system, recommending threads similar to the one currently read. Two different similarity measures were tried out, with the first (cosine similarity) being abandoned due to performance issues, replaced with a binary vector similarity solution. The data still needed to be reduced to get execution times down to acceptable levels, and sampling and partitioning were used. A live test showed, after a week and about ten million page views, that the system provided a benefit a bit above 10% over a control condition in terms of number of clicks.

Handledare: Erik Sondén
Ämnesgranskare: Kjell Orsborn
Examinator: Elisabet Andrésdottir
ISSN: 1650-8319, 0UPTEC STS 11 013

Populärvetenskaplig beskrivning

Rapporten handlar om projektet att konstruera ett rekommendationssystem för webbforumet Familjeliv.se. Systemet ska bidra till en bättre navigering i forumet genom att föreslå diskussionstrådar som en läsare troligtvis vill se. Projektet är mer tekniskt än vetenskapligt men trots detta har vi två centrala frågor att få svar på: hur räknar man på bästa sätt ut hur lika två diskussionstrådar är, och går det att hålla resultatet aktuellt när användarna hela tiden skriver nya inlägg?

Rekommendationssystem av collaborative filtering-typ (dvs att det är information om användarnas beteende som används för att räkna ut rekommendationerna) kan bygga på att hitta liknande användare, liknande produkter (trådar i det här fallet) eller att mäta kvaliteten på produkterna. Oavsett typ så är den viktigaste frågan hur man hanterar de stora datamängderna, det måste gå förhållandevis fort att räkna ut vad som ska rekommenderas, och framförallt måste metoden vara skalbar, dvs tiden det tar att räkna får inte öka explosionsartat när mängden data ökar.

Pga att bara ca 10% av användarna som läser forumet är inloggade så är det inte möjligt att rekommendera baserat på liknande användare då man i regel inte vet vem användaren är. Alltså valdes att räkna ut likhet mellan trådar, för att därigenom kunna rekommendera liknande trådar till den en användare läser. En första plan valde cosinuslikhet som metod att definiera likhet, ett mått som tittar efter liknande proportioner i vilka som skrivit mer och mindre i en tråd. Det kräver dock att hela måttet räknas om så fort någon skriver något nytt i tråden, och av den anledningen förkastades lösningen. Genom att förenkla måttet och bara räkna antalet användare som skrivit i båda som likhetsmåttet mellan två trådar kan det uppdateras snabbt och lätt genom att bara lägga till 1 på rätt ställen när en användare skriver i en tråd för första gången. För att ta vara på den mer specifika information som fås av användare som är mer restriktiva med hur mycket de skriver så viktas användare ner om de skrivit i väldigt många trådar.

Måttet räknades ut genom att gå igenom alla användare och notera alla par, men att göra detta rakt av gav alldeles för mycket data som resultat, nästan 6 miljarder par, från ca 2 miljoner trådar. För att minska datakrafts- och utrymmesbehovet användes två tekniker, dels ignorerades användare som skrivit i mer än 300 trådar, eftersom de gav upphov till mycket data och data av låg kvalitet. Dessutom minskades antalet trådar som kunde jämföras med varandra genom att begränsa jämförelser till trådar som låg i samma kategori – forumet har 361 underkategorier. Med detta blev det ca 700 miljoner trådpar, många men ett hanterbart antal.

Systemet testades live på forumet, och effektiviteten mättes genom att rekommendera tio trådar i en ruta intill varje gång en tråd visades, där det var slumpmässigt utvalt hur många från det faktiska rekommendationssystemet som skulle visas (från inga till alla tio) där resten var helt enkelt de nyaste trådarna i underkategorin. Resultat efter en vecka och tio miljoner

sidvisningar var att systemet utgör en förbättring i termer av antal klick på rekommenderade trådar. Med alla tio trådar från systemet blev det ca 13,5% fler klick än med alla tio de senaste från underkategorin.

Svar på huvudfrågorna blev att cosinuslikhet är det bästa likhetsmåttet kvalitetsmässigt, men att det har effektivitetsproblem för stora datamängder, och en förenklad summering av antalet användare gemensamt duger fullgott. Det är samtidigt möjligt att designa ett system som går att uppdatera i takt med att ny data tillkommer, men det sätter vissa begränsningar på vilka likhetsmått som kan användas.

Table of Contents

1 Introduction

1.1 Summary

This project was undertaken with the aim of improving navigation on the site Familjeliv.se. The focal point of this site is the forum, which contains over two million forum threads. Site management felt that there should be some sort of system in place to recommend threads to users, to alleviate the difficulties inherent in navigation through so much information, in the belief that this would increase traffic and user satisfaction. The system was to be of collaborative filtering type.

The first major decision was that the system would be based on thread-to-thread similarities, and recommend similar threads to the one a user was currently reading. With support from the literature on collaborative filtering a plan based on cosine similarity was developed. The first major plan was rejected for performance and updatability reasons, and a second was developed. The second plan was based on binary vector similarity and was superior in terms of performance and primarily updatability.

Two different algorithms were tried out to calculate these similarities between threads, one taking one thread at the time and the other one user at the time. The second one proved to be faster. Still, performance was a problem due to data abundance, and some data reduction was needed. First some directed sampling was tried, which was acceptable but suffered from quality loss. Instead a partitioning approach proved superior.

By the initial online test, additional performance issues surfaced. But a modification and recalculation of the similar threads table managed to solve this problem at very little cost. The test eventually showed that the final system did help navigation in the forum and gathered more clicks than the baseline navigation system.

1.2 About the project

Familjeliv.se is a community-style website dedicated to family life, the largest such in Sweden. Topics treated include, among others, relationships, marriage, pregnancy and child rearing. The site has various features but at the center sits the discussion forum. It is the main attraction to the site, and it boasts hundreds of thousands of registered users, millions of discussion threads in hundreds of subforums, some but not all dedicated to family-related matters.

This abundance of material means that users cannot easily get an overview of what is available to peruse. New material is created quickly, too quickly for users to comfortably keep up and survey it, since there are hundreds of new threads every day. It is therefore believed by site management that users are somewhat prevented from finding and reading the discussion threads they are the most interested in. If there was a way to more efficiently navigate the forum, making it easier to find the things you prefer to read, users might read more, increasing the number of page views. This would lead both to higher advertising revenue as well as increased user satisfaction. Therefore management wanted to develop a discussion thread recommendation system to accomplish this. This report is about the execution of that project.

The purpose of this project is to increase the number of page views on the familjeliv.se forum by finding a way to recommend threads to users by analyzing patterns in the usage data. The recommendations should help users find threads that they want to read in a more efficient way than what is currently the case. Using an increase in clicks as the only goal is somewhat short sighted, and the actual goal is broader. It is to create a tool that makes navigation easier and more enjoyable, and this more or less means creating a system that site management finds satisfactory. An increase in number of clicks was chosen as a performance goal because it is quantifiable and clear, not because it's the only thing that matters.

At the time of the project starting, there was a basic recommendation system in place. This system simply recommended threads universally that had been the most popular to participate in during the last 24 hours. In the end, the new system was supposed to be an improvement on this, in terms of click through rate.

In theory this could be done is several ways. Recommendation systems can be based on the content of the things recommended (called content-based) or be neutral or oblivious to this and only use the behavior patterns of users to infer the proper things to recommend. This latter strategy is usually called collaborative filtering and it was explicitly stated that the recommendation system were to be of this type, and not about textual analysis of the content of discussions.

1.3 Report structure

After section 1, which introduces the project, section 2 will outline the theoretical background that will be relevant to the project, which will primarily include proximity measures and means of data reduction. Section 3 will give information about the site, its structure, content and traffic. Section 4 describes the planning phase of the system, where main approach is chosen, and one plan being developed and discarded before a final plan is made. Section 5 is about the execution and implementation of that plan into a completed system. Section 6 outlines the background, execution and result of a live test. Finally section 7 contains some concluding remarks.

1.4 Main questions

The scientific perspective is less than prominent in this project, since it is about constructing a system rather than researching a scientific issue. This means it is more about surveying and choosing techniques rather than finding any new knowledge.

However, the project has some central challenges that can be thought of in scientific terms, and whose eventual solutions can be thought of as answers to scientific questions.

How do you capture similarity between forum threads in the best way possible?

The data set in question is constantly changing, how does this affect which solutions are possible and appropriate?

2 Background

When describing how the project went about, a number of concepts, methods and terms will be used. This section outlines these, whose importance to recommendation systems range from the highly theoretical to the practical.

2.1 Collaborative filtering

Collaborative filtering is an umbrella term used to describe ways to use the behavior of large number of users to infer patterns in the tastes and interests of people and use this information to make it easier for users to find things they like. "Collaborative" refers to the means: the use of the behavior of many users, each of which in a sense collaborates to create order among a multitude of options. "Filtering" refers to the end result: that information is sorted, ranked or ordered in some way. The term itself was coined by the makers of one if the first such systems, called Tapestry (Su, 2009).

The use of collaborative filtering systems has exploded in recent years. Fast internet connections and fast computers on the one side, and an increasing availability of entertainment for consumption on the other have made number crunching in the service of navigating the new jungle of possibilities a reality. Before, when endeavouring to find something to enjoy in music, film, books or the like, we had to rely on professional reviewers who might or might not share our tastes, on friends who might not either, or sales charts that are informative only to the extent that our tastes mirror that of the average person (whoever this 'average person' is). Not anymore; as the available options grow, both among the traditional cultural forms (books, music, films) and newer (discussion forums, Youtube videos, blogs), the need to know what to look for is greater than ever. Luckily, due to the large collections of data that now can be captured and used, the possibility of doing so is greater than ever.

This also raises new challenges. The computers are getting faster but the load they´re expected to bear is also rapidly growing as more and more information is gathered and more and more sophisticated techniques are being deployed in the treatment of that information. An algorithm might produce great results but at the cost of a lot of calculations, which would render it intractable if used on datasets that are very large. *Scalability*, is a central concern for information processing generally and data mining applications specifically. It refers to the property of an algorithm to retain practicality as the dataset grows large. An algorithm that takes a hundred times as long to treat a data set hundred times as large has good scalability I general. An algorithm that takes a billion times as long when the data is a hundred times as large is not, because no matter how fast the computer is the amount of data is likely to increase faster than what can be accommodated. Handling scalability is a major limitation and challenge when developing collaborative filtering tools.

Large and small companies use collaborative filtering systems. The mighty Google is itself a collaborative filtering system, recommending web sites based on votes of confidence from other web sites (Google), movie rental company Netflix held a high-profile competition with a million-dollar prize to whomever could improve their recommendation system by 10% (Netflix), retailer Amazon uses its purchase data to suggest things you might want to buy

depending on what you've bought before and what you're reading about (Linden, 2003). Minor user generated content is filtered by similar systems as well: on comment boards, review boards, discussions and forums, user "upvotes" and "downvotes" are routinely used to gauge the value of contributions.

Before continuing, it's best to define some key terms we'll be using later on. First: *user*, refers to a person, (or possibly some other type of actor) that is the target of the collaborative filtering engine. The ultimate goal is to recommend things to the user that he or she will like. Users have relationships to *items*, which refers to the units that can be recommended, consumed, liked or disliked. Items can be goods, or web pages, or forum threads, or videos or pieces of music. The goal of any collaborative filtering engine is to use the aggregated relationships of users and items to predict relationships between users and items about which data was previously lacking.

2.1.1 User-item relationship data

User-item relationships is easily represented and thought of as two-dimensional data. That simply means that there are two types of things and a relationship is present at the intersection of a thing of one type and a thing of the other. The typical way to represent two-dimensional data is by a matrix. The matrix has one type of thing represented along its rows and the other along its columns, which makes the entries in the matrix represent all the intersections between things of the two types, for instance, the entry in row 16 and column 20 represent the relationship between thing number 16 of type 1, and thing number 20 of type 2. In our case that would be between user 16 and item 20, or item 16 and user 20.

Usually the data matrix is thought of as organized into *cases* (represented by the rows) and *attributes* (represented by the columns) where the cases are the concrete objects that have been observed, and the attributes being different aspects of the cases that have been measured (Tan, 2006). Sometimes what is the cases and what is the attributes in a data matrix is obvious, such as when the cases are people and the attributes such things as the peoples' height, weight, IQ, eye color, political leanings and such. Other times the two types of things are both entities in their own right ('a weight' or 'an eye color' aren't really coherent entities in themselves) such as books and movies (that are). In that case what is considered cases and what is considered attributes depends on the application and the statistical methods used, as we shall see later on.

The data concerning relationships of users to items can be categorized by three major dichotomies. The first is whether the data is explicitly or implicitly given. Data that is explicitly given is usually called ratings, and it means that a user has purposefully given information about their relationship to an item.

When the user has not supplied data on purpose, the relationships has to be inferred from behavior. Such data can be called *consumption data* (or simply *implicit*) because it is frequently the consumption of various items that constitute this data. Consumption data indicates user-item relationships by recording whether a user has *consumed* an item. What consuming means differ depending on the exact nature of the application. Purchasing a thing or viewing a video could be defined as consuming, but so could reading about that thing or

posting a comment about that video. It all depends on how much and what type of data one wants. But a clear definition of consumption is essential, of course not excluding some composite measure.

The second dichotomy is whether the relationships described by the data are *unipolar* or *bipolar*. Unipolar data means that the magnitude of the number representing the relationship indicates the strength or confidence of that relationship. Consumption data is frequently in unipolar form, since consumption of an item tends to indicate a positive relationship to that item, and more frequent or more intense consumption indicates more or stronger liking. What makes data unipolar is, however, lack of information about different *kinds* of relationships. There is information about positive relationships in contrast no relationship, but not about negative relationships in contrast to none (Hu, 2008).

Bipolar data does however indicate positive or negative relationships, and is more common with ratings. To exemplify the difference from unipolar data, a rating of 1 out of 5 for an item does not mean "I like this, but only weakly", but rather "I dislike this strongly". Bipolar and unipolar data must therefore be interpreted quite differently. Most work on collaborative filtering in academia has been done with ratings data that is explicit and bipolar, possibly because of its ease and convenience (Hu, 2008).

The final dichotomy is between binary and gradual data. Binary data is just that, binary, it indicates presence or absence of a kind of relationship, not its strength or confidence. Typical binary data is data on purchases, a user bought an item or not, which would indicate the presence of a positive relationship. Another example of (unipolar) binary data are Facebook "likes" which are not graded according to *how much* you like something. While perhaps not binary in a strict data-representation sense, binary data can be bipolar as well, indicating the presence of a positive *or* negative relationship, but not its strength. The "upvotes" and "downvotes" used in some discussion forums is such data, along with Amazon's and Imdb's use of "x out of y (y>x) users find this review helpful" style filtering of reviews.

These three dichotomies produce eight different permutations, of which only some tend to occur. It's for instance difficult to establish bipolar implicit data.

2.2 Common approaches

Collaborative filtering systems tend to fall into three broad categories. Two of these are based on computing similarities, either between users or between items, and the third on establishing a general quality measure for items. Each of these has their own advantages and drawbacks, and is appropriately used in different situations.

2.2.1 Personal recommendation or user-to-user

Personal recommendation or user-to-user is the method based on computing similarities between users. This can be done in various ways but is usually based on data pertaining to users' relationships to items. This data can be expressly given in the forms of ratings, or implicitly deduced from the consuming or non-consuming of an item by a user. Most of the literature and applications deal with explicit and bipolar data but this approach is possible with other types of data as well.

Commonly a personal recommendation system works by computing some type of similarity measure (such as Pearson correlation, cosine similarity or Jaccard coefficient) between users based on ratings and/or consumption. To recommend an item to user *u*, other users similar to *u* are selected (called the *neighborhood*) and a weighted average of their ratings is used to predict the rating or probability of consumption for items not yet rated or consumed by *u*. The items that receive a high prediction value are then recommended (Adomavicius, 2005).

The main advantage of personal recommendations is high quality recommendations, based on the conservative assumption that users that are similar in their taste toward some items will tend to be similar in their taste toward others. But it does have drawbacks, one being expense. Calculating the similarity between every pairing of users is an $O(n^2)$ operation where n is the number of users, and every step is O(m), where m is the number of items included in the comparison. This renders an algorithm of $O(n^2m)$ complexity in the typical case, which is expensive but not necessarily prohibitive. Even though, the need to recalculate to accommodate new information makes the updatability of the technique less than optimal.

A more serious issue is the dependence on high quality data to begin with. The more data there is on a particular user, and the more overlap there is between the data of that user and other, similar users, the better the recommendations. The flip side is that when there is very little data the similarity measures between users become highly uncertain, with quality severely suffering. And with new users, or unregistered users where there is no data at all, personal recommendation systems are completely powerless.

The quality they do provide, when data is available, does demand some collaboration from the user, most often in the form of ratings of items, at least for the well-developed techniques that require bipolar ratings. This is a drawback in situations where it is difficult to persuade users to give ratings in sufficient amounts. Using pure consumption data is a possible solution but tends to erode quality because a unipolar consumption variable has less information than a bipolar rating, and there is no way to distinguish between a lack of consumption due to an item being uninteresting and it being unknown to the user. This also renders the most common similarity measures less useful (Lee, 2007).

The strategy there works best when there are plenty of data for every user and where data is of high quality (such as ratings) and users are willing to participate. Sites using personal recommendation system includes Netflix and Filmtipset (both movie recommendation engines) and early systems such as Tapestry, Video Recommender and GroupLens (Adomavicius, 2005).

2.2.2 Item-to-item

Item-to-item collaborative filtering is in a way the opposite of the personal recommendation approach. Rather than using similarity in item tastes to calculate similarity between users, item-to-item uses similarity between the sets of users that items attract to calculate similarity between items. It uses the ratings/consumption data of a user to find items similar to those the user appears to like, and recommends them.

The item-to-item technique is about as expensive as the personal recommendation technique, but has some advantages. One is the quick start-up; it needs only a single positive data point to start giving recommendations -- that is, if all we know is that a user likes item *I*, we can recommend items similar to *i*. Because even though knowledge about the user is poor, the connections between *i* and similar items is well established.

Another advantage is the possibility of responding quickly to new information. To update itself and generate new recommendations after a user has rated or consumed an item, the similarity measures between the user and other users must be updated in turn when using a personal recommendation system. That can take some time, which is costly when the user expects the new information to be considered immediately. When using item-to-item, however, the system can respond immediately, since the adjustment to item-to-item similarities that new user data generates is less time critical and new recommendations based on the recently added item can be given right away (Linden, 2003).

Drawbacks can include worse quality than with personal recommendations, since recommendations can be based on similarity with only one rated/consumed item (though this can be alleviated through the use of weighted averages, then again requiring more data).

Item-to-item are preferably used when data is sparse and insufficient for the personal recommendation strategy, for instance in retail when people might not have bought so many items, or when quick updatability is paramount. The best known example is also Amazon, basing their simple yet powerful "Customers who bought X also bought Y" on this method, as well as Youtube's "Related videos"-system.

2.2.3 General quality

A third option does not rely on any sort of similarity at all. Rather, a general quality approach tries to rank items by their quality through some treatment of user behavior data. General quality can be used when the data necessary for other methods isn't present. Google's PageRank algorithm is in its most basic form general quality-based (Google). This makes sense since finding similar users among all people in the world or finding similarity between all web pages on the internet is a ludicrous proposition. But it also highlights the fact that general quality needs to be thematically supplemented (in Google's case with search terms).

And general quality is indeed most appropriate when 1) items are many and the data on them in relation to users is poor and 2) there is some complementary filtering by theme. Often this means internet forums or message boards where quality is applied to post or threads, most commonly to posts where it is very difficult to know whether a user likes or consumes an individual post, and new ones arrive quickly with no data on them. That they're present in a certain thread ensures some higher level thematic coherence with makes it possible to speak of general quality. Quality can be measured, and it most commonly is, by simple voting by other users (for instance by counting 'thumbs up'/'thumbs down' or equivalent).

2.3 Performance issues and solutions

Collaborative filtering schemes are often the solution to a perceived overabundance of items. After all, if the amount of items were small enough for the user to examine them all,

recommendations would be of limited use. So, the more items there are the greater the need for filtering systems, and the more users there are to supply data, the better these filtering systems perform, quality wise. Taken together, this tends to inflate the size of the datasets used in the systems. When users and items number in the thousands, performance is not an awfully important concern using today's fast computers. When we reach into the millions, however, things are different, especially for algorithms whose execution times grow faster than linearly. Facebook, for instance, has over 500 million users, doing some sort of user-similarity calculation among such a large group would require an astronomical amount of computations.

Important issues to consider when implementing a collaborative filtering algorithm therefore include performance, in an execution time sense. Sometimes a system simply needs to provide results quickly, and sometimes it doesn't, but we still need to make sure than increasing the number of items or users by, say, an order of magnitude, won't cause the system to break down completely. There are ways to address an overabundance of data by somehow reducing it. Data reduction mainly takes three forms (Linden, 2003).

2.3.1 Dimensionality reduction

Dimensionality reduction aims to reduce the number of units to compare by aggregating them. This can be done by a number of statistical techniques. Principal component analysis can tease out patterns in data such as ratings and find regularities that help represent the data with fewer variables, cutting down on execution times. Other techniques include clustering methods, where users are arranged into clusters and receive recommendations for the whole cluster rather than for each individual user. This can also work the other way, of grouping items and then recommending, the whole cluster. Dimensionality reduction can greatly reduce the number of calculations that the system need to perform to get recommendations, but the clustering or principal component analysis can take quite a bit of time themselves, and quality inevitably suffers (Tan, 2006).

2.3.2 Sampling and feature selection

If dimensionality reduction works by disregarding some data deemed to be less important, sampling disregards data in a more straightforward way. Assuming data abundance, we can also perhaps assume that that data contains a lot of redundancy. Maybe we don't need 100,000 people saying that item A and item B are similar if having 1,000 people saying it is quite enough. Sampling then selects only part of the data for use. We might decide item similarities based on only the data from a small group of (randomly or not) selected users, or find similar users based on only the ratings of a small number of items.

The other side of sampling is feature selection, and what we're dealing with depends on what is treated as cases and what is treated as attributes in the data matrix. If we are selecting cases we're using sampling, and if we are selecting attributes it is called feature selection. But when the assigning of cases and attributes can be done either way there is no need to keep these separate. I'll call it sampling for now on, regardless if we're talking about sampling some users or some items (Tan, 2006).

Sampling can be done straight, such as a pure random sampling, which is of course safest when it comes to avoiding insertion of unwanted biases in the data selection. Or it can be directed, aimed at using the users or items we think are the most important or the most diverse (uncorrelated data providing more information than highly correlated). Trying to find similar users, we might for example want to focus on rare items, as they may say more about the users than, say, their purchase or liking of bestsellers.

Obviously, sampling leads to quality loss, and to what extent depends heavily on the specific circumstances and the structure of the data. It is therefore difficult to evaluate generally, noting only that some quality loss is inevitable and its main advantage to dimensionality reduction is cheapness and simplicity.

2.3.3 Partitioning

The third approach tries not to reduce the amount of data used but rather to reduce the amount of data produced by the system. Maybe all users or all items don't need to be compared to each other. Perhaps we only compare children with other children or the middle aged people with other middle aged people since we think pairs of very similar users tend to lie in the same age bracket, or be from the same country, or whatever. Partitioning users might sound politically suspect (even tough customer segmentation based on a number of demographic factors is hardly uncommon), and a more natural and uncontroversial approach is to partition items into groups. If you, like Amazon, are an online retail company and keep a list of item pairs that are similar to each other, you might refrain a priori from looking for pairs between clothes and books, or shoes and dvd:s, since the small number of interesting findings you're likely to get aren't worth the much more data trawling you'll need to do – it might be better to use separate systems for different product categories.

Partitioning can reduce quality, like all data reduction, how much depends entirely on the data structure and the diversity of users/items. The upside is mainly that it, through the cost of removing some interesting unexpected findings, can get rid of a lot of useless information production (Linden, 2003).

2.4 Mathematical concepts

In the following report about the development of a collaborative filtering system for familjeliv.se, a few central mathematical concepts will be used or discussed. To save having to discuss both their nature and their appropriateness to the project at once, they will be presented here first.

2.4.1 Vector proximity

It is important to keep similarity and dissimilarity measures apart, often they can be easily converted, but it's not always obvious how this should be done. A commonly used term that means both similarity and dissimilarity/distance is *proximity*, which will be used here when appropriate.

When defining how similar or dissimilar two scalar numbers are, there are several possible measures to use. One way to define proximity for the scalar numbers is the absolute value of the difference between them; or the square of that; or the difference between the numbers'

normalized position in the distribution of all numbers in the set; or the difference in their rank in the ordered set, and so forth. When it comes to defining the proximity between two n-dimensional vectors a bewildering array of possibilities opens up. The Euclidian distance between the two points in n-space seems natural. That is:

$$d_{x,y} = \sqrt{\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{y}_i)^2}$$

Euclidian distance is merely one variant of the more general Minkowski distance between two points in n-space:

$$d_{x,y} = \sqrt[k]{\sum_{i=1}^{n} (\boldsymbol{x}_i - \boldsymbol{y}_i)^k}$$

Often Euclidian or some other Minkowski distance is perfectly suited to the application at hand. But sometimes not. Minkowski distances measure the absolute distance between points, not taking into account the distribution of data, or the relationship between components of the vectors. Mahalanobis distance takes into account the distribution of the data by (loosely speaking) multiplying with the covariance matrix of the components. But if a covariance matrix is costly to obtain and the more pressing concern is finding the internal relationships between the vectors' different components without it being obscured by differences in the vectors' overall size, cosine similarity or Pearson correlation is a better choice.

Cosine similarity ignores the overall distance from the origin when comparing two points. Multiplying one of the vectors with a constant factor does not change the result. This is desirable when we're mostly interested in the relationship between component sizes. For instance, if we have users rating items from 1 to 5, user A might be generous and award many four and fives, and user B stingier, giving mostly twos and threes. With a Minkowski distance, this difference in general generosity would mark the users as quite different, even though it is quite possible that user A:s fours correspond closely to B:s twos and fives to threes respectively, so that their tastes are similar in the respect that they tend to prefer the same things. Cosine similarity handles this, caring only about the relationships between component sizes. Intuitively speaking, it is the cosine of the angle between two arrows drawn from the origin to the two points whose similarity we're interested in (which is 1 for perfectly aligned vectors, 0 for orthogonal and -1 for those perfectly opposed). It is less vulnerable to the curse of dimensionality when there are a lot of 0-0 matches, since those are completely ignored. Its value is given by:

$$s_{x,y} = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{|\boldsymbol{x}||\boldsymbol{y}|}$$

Pearson correlation, on the other hand, is a measure of the strength of the linear relationship between the two vectors' components. It's defined as:

$$r_{x,y} = \frac{\sum_{i=1}^{n} z_{xi} z_{yi}}{n}$$

Where z are normalized vector components. Pearson correlation remains invariant to addition of constants to all components, due to the fact that the normalization of components automatically centers the data. That makes the center of the values the mean, while when using cosine similarity, the 'center' of the values are always 0. Whether to use Pearson correlation or cosine similarity therefore depends on what sort of values are to be interpreted as "high" or "low", as well as data sparsity. Pearson takes all data into account, whether it is present or not, meaning 0-0 matches are treated as information (it can of course be forcibly ignored). Cosine on the other hand, only looks at nonzero values, meaning it is superior if 0-0 matches are seen as a lack of information. Generally, if data is sparse, cosine is better than Pearson or Minkowski (Tan, 2006). Cosine is also generally preferable when we have only implicit data, since that often mean that data is unipolar and there is no "negative" data around, making Pearson's automatic data centering undesirable (Su, 2009).

2.4.2 Binary vector proximity

The previous measures concern vector similarity and distance where the vectors have scalar components. That means they are appropriate in treating data where there is a gradation in the relationship between users and items, like ratings. High ratings indicate a strong relationship, and low ratings a weak or negative relationship (whether low ratings indicate weak or negative relationship is the Pearson-Cosine question in a nutshell: if weak, cosine; if negative, Pearson).

But if we don't have ratings, and instead rely on consumption data, there is often no gradation. We only know whether a user has consumed an item or not, whether there is a positive relationship or not. The data then come in the form of binary vectors, where 1 counts as a relationship and 0 the absence of one.

For binary vectors there is another set of proximity measures. The simplest is to count how many components of the vector that have the same value (counting 1-1 and 0-0 matches) and divide by the total number of components. This renders a value between 0 and 1 and is called Simple Matching Coefficient. Of course, a value of 0 might mean several things, it could mean that the user is disinterested in the item, or it could mean that the user is unaware of it. That two users share the property of not having consumed an item is perhaps not a very strong indication of their similarity, especially if there are a lot of items. A 1-1 match might be more important, indicating some similarity in taste more strongly. A modified Simple Matching Coefficient could count only 1-1 matches. When a 1 value is considered as information of a higher quality than a 0 value, the attribute is considered asymmetric, and requires special treatment as outlined above. SMC is usually not appropriate in this case (Tan, 2006).

A commonly used variant is to ignore the total amount of items available (the dimensionality of the vector) and only look at the proportion of the vectors' nonzero components that overlap. Jaccard coefficient does just that, counting 1-1 matches and dividing by the total number of components where at least one of the vectors has a nonzero value. As a rule of thumb Jaccard works well on sparse data (Tan, 2006). Like with vectors consisting of scalar

values, which similarity measure is appropriate depends on the interpretation of the data. Simple Matching should be used when 0-0 matches are significant. If it isn't one needs to judge how important 1-0 dissimilarities should be, (using Jaccard if they're judged to be important, possibly a modified Simple Matching if not) which again depends on the interpretation of a 0 value.

2.4.3 Set proximity

When one way to conceptualize a type of data as a binary vector of asymmetric attributes and the attributes are essentially of the same kind (as they are here, where the attributes on a binary vector would be the presence of various users), another conception of the data and its proximity measures is possible. Instead of having vectors with lots of zeroes and a few ones the data can be thought of as a set, a set consisting of the attributes in which the vector value is a one. The vector (0, 1, 0, 0, 0, 0, 1, 0) when the attributes are (A, B, C, D, E, F, G, H) can be thought of as the set {B,G}.

There are some ways to define set proximity, such as the number of members in common divided by the number of members in total, or the number of members *not* in common (distance) or simply the number of members in common (Tan, 2006).

2.4.4 Term frequency, inverse document frequency

When comparing two text documents or comparing one document to a search phrase, the 'term frequency-inverse document frequency' scheme or a variation of it is a common strategy. It is based on converting a text document into a set of word-number pairs. These numbers can then be used to compute the similarity between documents by pairing up numbers associated with the same word and using a vector similarity measure as outlined above, or match documents to a search phrase by adding up the numbers associated with the searched-for words.

The important part lies in how these number are calculated. Tf-idf basically takes two things into account. The first part, term frequency, is a count of how common the word is in the document, compensating for its length.

$$tf_x = \frac{n_x}{n}$$

Term frequency for word x is the number of occurrences of x (n_x), divided by the number of occurrences of words altogether (n).

To compensate for some words being a lot more common than others, for example, "the" being a lot more common than "euclidian", wanting differences in the term frequency of "the" to not swamp more important differences in the term frequency of "euclidian", inverse term frequency is used. Inverse term frequency is defined as:

$$idf_x = \log\frac{n}{n_x}$$

Where n is the number of documents in a set used to represent the language in general (a *corpus*), and n_x is the number of documents where the word x is present. The less common a

word is in general, the higher idf becomes, and the more important its presence in a document is judged to be (Salton, 1988).

2.5 Relational databases

A relational database is a type of software that implements a data structure designed to hold large amounts of data that can have complex relationships to each other. The data is stored in tables with rows that signify individual records and columns that signify different attributes of the records (these attributes each have names, and are associated with a value of a specific data type). Through the use of id codes ('keys') for individual records, they can be cross referenced with other tables so that a set of tables can form a complex structure (loosely called "a database") where information can be retrieved from various tables.

2.5.1 SQL

To fetch information from a database in the form and with the filtering you need, as well as updating or entering data, a way to interface with the database is needed. SQL stands for Structured Query Language and is a standard language for managing database servers (the software that manages databases) (SQL.org). More similar to natural language than most programming languages, it is *declarative* rather than *imperative*. SQL offers the opportunity of, through simple keywords, creating, altering, deleting, updating, looking up, inserting, filtering and ordering data representations. Sometimes SQL is used "in the raw" but most frequently it is embedded into applications that use it to read and write data to databases under the hood.

3 Site information

3.1 Specs and numbers

The discussions in the forum receive about 20,000 new posts every day. These posts are made in old and new discussion threads, whose total number at the time of the project reached about 2,2 million, increasing by hundreds every day. These discussions are conducted by approximately 320,000 registered users. The site has close to a million page views per day.

3.2 Forum structure

The main topical focus of the forum is, as one might surmise from the name, family related matters. Much of the forum is allocated to various sub-topics within this, and some is about more general areas of discussion, such as work, politics, travel, news, sex or religion. To accommodate this topical breadth, the forum is divided into 15 categories (called 'communities'), such as 'Pregnancy', 'Parenting', 'Relationships', and 'General'. These communities are in turn subdivided into 361 even narrower categories (called 'forums'). Every thread belongs to a forum and every forum belongs to a community.

3.3 Main system architecture

The system runs on a MySQL database server with PHP-generated HTML as a front end. The database consists of lots of tables but some are more central to the project (MySQL).

3.3.1 Important database tables

The database contains many, many tables. Only a few of them were directly relevant to the project, and in those only some of their fields. Here they are described with their respective relational schema: relation(table) name and attributes(columns) depicted in parenthesis.

The 'forum_messages' table contains all messages in all threads;

```
`forum_messages`(`message`, `parent`, `subparent`, `owner`,
`created`, `name`, `icon`, `body`, `active`,
`source`,`reply`)
```

Fields that are important to the project are 'message', which is message id; 'parent' which is the id of the thread the message is in; 'created', which is its creation date; and 'owner', which is the user id of the poster.

The table 'forum_threads' contains thread representations where thread id is the id of the starting message;

```
`forum_threads`(`message`, `community`, `forum`, `icon`,
`subject`, `notify`, `latest`, `replies`, `nick`,
`created`, `created_by`, `active`, `poll`, `hits`, `db`)
```

Important fields are 'message', the thread id (really the message id of the first message); 'community' and 'forum' indicate the community and forum where the thread is located; 'subject' is the headline; 'latest' is the date of last reply and 'created_by' the user id of the thread starter.

The table 'forum_views' is information about which threads registered users have read and when;

```
`forum_views` (`id`, `thread`, `viewed`, `user`,
`message_viewed`)
```

Here 'thread' and 'user' indicate thread and user id, while 'id' is simply the row id of the pair. 'Viewed' is the date of viewing and 'message viewed' a list of which pages of the thread that were viewed.

Tables 'forums' and 'forum_communities' represent forums and communities as described in the above section.

```
`forum_communities`(`id`, `community`, `parent`, `created`,
`title`, `active`, `description`, `meta_keywords`,
`meta_description`, `position`, `db`, `priv`, `listicon`,
`sex`, `public`, `anonymous`, `polls`, `hidden_in_menu`)

`forums`(`forum`, `community`, `parent`, `created`,
`title`, `active`, `description`, `position`, `db`, `priv`,
```

```
`sex`, `public, `anonymous`, `restrictions_post`,
`restrictions_read`, `restrictions_reply`,
`meta_keywords`, `meta_description`)
```

Most important, however, is the ‘forum_member_usage’ table. At the start of the project, this table existed but was not used for anything specific. This type of situation was common in the database, with fields or tables being deprecated, of unknown purpose (due to the database having been gradually designed by different database administrators in succession without much documentation), redundant or generally unnormalized. It this case it was lucky, however, since ‘forum_member_usage’ proved useful. It contains information regarding which users post in which threads. Its fields specified a user id (‘member’), a thread id (‘thread’), the number of posts in that thread by that user (‘cnt’), and a date for the latest post (‘latest’), along with a flag indicating whether that user was the thread starter (‘is_ts’). Due to it not being used, I was unaware of its existence when constructing the first suggestion for the system detailed in section 5.1. Later on, new and modified tables were created for the project, they will be discussed later.

```
`forum_member_usage`(`id`,`thread`,`member`,`is_ts`,`cnt`,`
latest`)
```

4 System Architecture

4.1 Overarching goal

The project to introduce a collaborative filtering system to familjeliv.se has, ultimately, one clear goal: to increase traffic to the site. It is believed that the torrent of new material constantly generated is preventing users from finding discussions they would otherwise be interested in. For instance, a user might enter the forum once every three or four days, by which point there are many pages of new threads that she isn’t willing to trawl through in the hope of finding something interesting. The user is likely to skim only a subset of all new threads (or, in the case of a new user, a subset of all previously existing threads which is far beyond the capability of any individual to examine completely) and will therefore be less satisfied with the experience, and is likely to read a smaller number of threads in total.

Alleviating these concerns is thought to increase the number of page views and by that advertising revenue. The system is therefore supposed to assist users in finding discussion threads they want to read with a minimum of effort on the part of the users.

4.2 Basic technical requirements

What does this entail? What demands must we put on the system for it to accomplish the goals set for it? The most obvious criterium for success is that the system produces relevant recommendations. Avoiding philosophical ruminations about how relevance might be defined, in this case we consider relevant recommendations those that a user actually reads when recommended. Knowing in advance whether a recommendation is relevant is somewhat difficult, of course, and requires testing. But before we can do any testing, a system must be

designed, and for that simple judgement must be used to find a way to harness the behavior of hundreds of thousands of users into relevant recommendations.

Besides relevance there are other important considerations. In theory, every last drop of information could be extracted from the data through advanced statistical techniques, textual analysis and psychographic profiling of users. But in a situation where we have millions of threads and hundreds of thousands of users (registered users, that is), we need to be more economical with the computations. In short, the system needs to be quick enough. This need for quickness has two parts; it needs to be quick online, that is, recommendations need to come immediately, and it needs (to a lesser extent) to be quick offline, that is, calculations done in advance of the actual 'recommendation act' must not take an inordinate amount of time, i.e. not many weeks, months or years.

A third condition has to do with the type of data the forum is. It is dynamic, it's constantly changing and growing as new material, new threads, are created and new posts fill them. In contrast to a static data set, a system based in this forum must be able to incorporate new data in real time. This means that it needs to be able to update itself to include new threads and new information at least as quick as this new information is created. As it turns out, this demand puts certain limitations on what mathematical techniques are appropriate.

4.3 Main approaches evaluated

At this point, some basic choices must be made. The overall structure of the system should be designed around the technical goal. Now, the overarching goal is plain – to increase traffic through improved navigation. The technical goal must be more detailed – exactly what should be quantified and calculated in order to achieve the main goal? The three main approaches to collaborative filtering are all contenders to this.

The first, and generally most powerful approach would be personal recommendations. The system would then be built around finding users who read and post in the same threads and then recommend each other's threads to them. This wasn't used, however, and it is because of the composition of visitors to the site. As it turns out, only a small minority (about 10%) of users are actually logged in to their accounts when they visit the site. The other 90% are either not logged in or don't have accounts at all. A system with personal recommendations would have no capabilities at all when it comes to unregistered users, it would not help these important 'unattached' guests to become habitual visitors. This means that we cannot recommend threads to users based on knowing anything about the user, rendering the techniques most well developed in the technical literature unusable.

The opposite tactic to personal recommendations – 'impersonal' recommendations or valuing the general quality of threads, was dismissed at an early stage. Though the forum has a focus on family related matters, its size and number of different sub-forums ensure that discussions and the interests of those who take part in them are diverse. Assuming that all users will find the same set of threads as of high quality is dubious.

Instead, the choice fell on an item-to-item or item-similarity based approach. It would find pairs of threads that are similar to each other based on which users posted in and read it, and

recommend one when a user read the other. This would create a thematic web of threads tied together by links of similarity that a user could start traversing once they had made their own selection of an entry point. This was a safe choice as it seemed clear that the situation for familjeliv.se is structurally quite similar to that of Amazon.com, except that the site 'sells' threads, not goods. Other than that, there are few differences. Using the same type of system would lead to, when reading a thread, there being a box saying something like: "Users who read this thread also read:" and a list of recommendations. This works just as well for unregistered as for registered users. It is also easier to define a good similarity measure through user behavior than a quality measure, which runs the risk of being little more than a list of 'most popular items'. A quality-based system that works well, Slashdot, relies on a rather complex multilevel feedback system that requires explicit quality ratings by users, not only of the quality of items but of the accuracy of the ratings themselves (Ball, 2003). Item-to-item needs none of this and has few drawbacks in comparison. One being the risk that some items get no good matches, an issue that will be discussed later.

Item-to-item does, like other schemes, tend to have a problem with new items (Su, 2009), but since new threads receive high exposure in the forum by virtue of being placed first, we are likely to get data on them relatively quickly.

4.4 First plan

At the beginning of the project, my first assignment was to develop a fairly detailed account of a suggested system for approval by my supervisor/client. I will continue to refer to my supervisor/client as this to reflect the double role as my supervisor on the project as a degree project, and my client since it also has features of a consulting project.

4.4.1 Similarity measure

To design a proper thread-to-thread similarity measure we need to establish what sort of user-item relationship data is available. It is obvious that no explicit bipolar ratings exist, since users to not rate the interestingness of threads. It would always be possible to create such a system but it's likely it would produce very sparse data since most users would not bother with the ratings, and there is also a library of over two million threads, most quite old and now infrequently visited at all.

That leaves unipolar ratings/consumption data of an implicit kind. The client's/supervisor's first instinct was to look at what users posted in what threads, assuming that a user posting in two threads indicate some similarity between them. Using user postings opens up questions about what sort of data this posting really represent.

Could number of posts in a thread be used as an implicit bipolar rating? The more posts, the more interesting the thread? Not really, the problem being that even the most prolific posters only post in a small proportion of threads, and the selection of those threads is biased to only include some of those that are the most interesting. That is, we only get some of the upper tail end of the ratings, having no way of distinguishing between "completely uninteresting" and "just not interesting enough to post in" threads. We can't use a bipolar ratings model since the negative ratings don't really exist.

But a unipolar ratings model is possible. We can view the number of postings in a thread by a particular user as a measure of the strength or confidence of a positive relationship. There is not really information about negative relationships in postings data, only absence or presence and strength of a confirmed positive relationship.

What this means is that if we represent two threads as two vectors with *n* components (where *n* is the total number of users), component *i* being the number of posts that user *i* has made in the thread, what we want is to calculate a vector similarity. Since all nonzero values indicate a positive relationship, and zeroes indicate little more than the lack of a confirmed positive relationship, and the data is very sparse (even the most popular threads have only a tiny proportion of users posting in them) we should use a similarity measure that ignores the multitude of 0-0 matches. Cosine similarity provides a measure of the similarity of the composition of the discussion, by which is meant which users take part and how much they contribute in relation to each other. No negative scores are available, which leaves Pearson correlation an unsuitable candidate.

Since all values are nonnegative, all vectors representing threads will lie in the first quadrant (including edges). That makes zero the minimum similarity score, which occurs when two threads have no posters in common, making the dot product of the vectors zero. It makes sense that this minimum value is zero and indicates a lack of any relationship whatsoever. In theory, there could be a negative relationship between threads, but that could be assumed only if the number of posters in common was smaller than what would be expected purely by chance. But since only a tiny proportion of users post in any given thread, the expected number of common users for two threads is miniscule and doesn't differ significantly from zero. The exact calculation is messy, especially since posters are not equally prolific and it is difficult to decide exactly how to model the situation (what information to take into account and so on), but it is clear that similarities below zero, unlikely to be found at all even among millions of threads, would be of tiny magnitude and completely devoid of value. We, after all, want to find high similarities, not low or negative ones.

So, using cosine, the similarity score will be between zero for thread pairs with no common posters, and one for pairs with the exact same posters, each contributing the same proportion of the posts in each thread.

Tf-idf will come into play here. But it won't be in its standard form, describing a text document. It will be co-opted to compensate for some users being a lot more prolific than others. Some users post all the time, having a low threshold to participating in discussions. Other will be more restrained, (or lazy) and post less often. For someone who only posts rarely, a few postings in a thread represent a stronger indication of a positive relationship than for someone that writes a lot of posts all the time. We want a way for the system to appreciate that not all users' postings are 'worth' as much. Here is where tf-idf comes in. Using those techniques, we can conceive of each thread as a document. But that is a document that consists of the author names of every post in the thread. A thread with five posts and three posters (Alice, Bob and Chris) could be equivalent to the short document "Alice Bob Alice

Chris Bob". From this term frequency could be collected, and the inverse document frequency would be taken from the "corpus" consisting of all threads.

Instead of having pure post count for each user, the components of the vector representing a thread would have the tf-idf scores of each user, in short, the number of posts in the thread times the logarithm of the ratio between the number of threads in total and the number of threads the user has posted in. Note that the numerator is the absolute number of posts, not the ratio of that user's posts to all posts in the thread, as it would be in a pure tf-idf application. The reason for it is that its purpose would be to compensate for thread length, which translates to vector magnitude. But cosine similarity already compensates for that, so it isn't necessary.

4.4.2 Suggested implementation

The next step is to sketch how to implement this measure technically. An algorithm needs to be created and a program needs to be written. How would you calculate and store the cosine similarity scores, and how will they be updated to accommodate new threads and posts?

The first and easiest choice was language selection. It was going to be Java, since that's what I know the best by far and there was no particular reason to choose something else. Performance might be slightly better with some other option, but the overhead required for me to learn it mid-project made anything like that untenable. To interface with the database I chose Java Database Connectivity (JDBC), again mostly because of familiarity and no reason to believe that using something else would be better.

The idea was to extract thread information from the main MySQL database of the site, perform calculations in a Java application, and write the information specific for the filtering engine into its own parallel database to be accessed by the web server (figure 1).

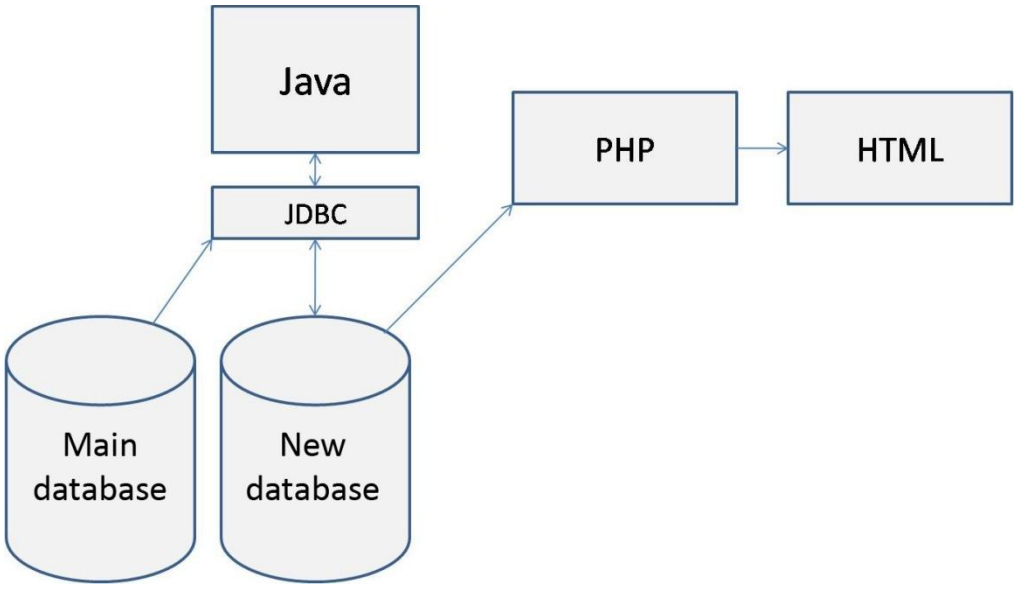

Figure 1, overview of proposed system structure.

The reason for creating a parallel new database was simply ease and the possibility of improved performance if we were to access the main database less. However, since they were both to be hosted on the same MySQL database server there was no performance benefit according to the database administrator and this idea was scrapped, and the new tables were created in the main database.

Calculating the similarities and keeping them current was going to be accomplished through three parallel Java processes: thread representation, complete calculation, and heuristic calculation.

The purpose of the thread representation process was to package the relevant information about the thread in a format that was easy and fast to access. Its operation would be to, at startup, get a list of all thread id:s, go through each of those and call up all posts that belonged to that thread, count the number of posts by every user who had posted, and then store this information in a table called "thread_representation".

```
`thread_representation`(`id`, `posts_info`, `time`)
```

'id' would be the thread id, 'time' would be the date and time the information was stored (the latest time it was guaranteed to be accurate), and 'posts_info' would be a string consisting of a set of pairs like [user_id]-[number_of_posts] separated by spaces. After having completed, the thread representation process would continue to run indefinitely. Now calling up (by date) all posts that had been created since the program started, updating the information on all threads in which these posts had been made - and after that, again calling up all posts new since the last iteration, and so on. This would guarantee that thread information would be as current as possible.

Actually calculating the similarity scores is a different beast altogether. Representing 2,2 million threads in the manner outlined above can take a while. But at least it is O(n) regarding thread count. Calculating pair similarity is $O(n^2)$ if we are to examine all possible pairings. 2,2 million threads means almost 5 trillion pairs, an unmanageable number. Fortunately, as we've previously discussed, data is sparse. Most pairs of threads have no common users, and thus the vast majority of pairs have a similarity score of zero. A lot is won if we can avoid the zeroes altogether, only representing nonzero values.

The complete calculation process was to do this, and it could be started as soon as the first iteration of the thread representation process was completed. It was, just like the thread representation process, supposed to call up a complete list of thread id:s and step through it, one thread at a time. For each thread it finds all other threads that has a nonzero similarity with it, calculate that similarity and store it in a table called "sim" with fields 'first' (the lowest id of the two threads), 'second' (the other thread id) and 's_score' (the cosine similarity).

```
`sim` (`first`, `second`, `s_score`)
```

How does it find all those other threads with which the similarity is nonzero? What it means to have a similarity more than zero is at least one poster in common. The list of such threads can be found with a nested SQL query:

```
SELECT DISTINCT parent
FROM forum_messages
WHERE owner IN (SELECT DISTINCT owner
                FROM forum_messages
                WHERE parent = [current thread id])
              AND parent > [current thread id])
```

In (relatively) plain language, this query gets the unique thread id:s (higher than the current thread to avoid calculating the same pairs twice) from all posts made by users that have made posts in a particular thread. This is a rather heavy query when there are many millions of posts. Doing this a million times would take its time, especially since the cosine similarity calculations require some steps. It would, when it had the two thread id:s in the pair, get the posting data from the thread_representation table, parse the posts_info string and compute the cosine similarity. To get the right weighting of vector components by idf score, it would also need to query how many threads each participating user had posted in, with the query:

```
SELECT COUNT(*)
FROM thread_representation
WHERE posts_info LIKE '%[user_id]-%'
```

It would be unwise to rely on this whole business to be redone over and over again quickly enough to update the similarity scores properly. So what to do? Attempting to alleviate this performance problem, I thought of the heuristic calculation process. A few changes to the complete calculation process make the heuristic one possible. First, another field is added, "time" indicating the date and time the similarity was calculated.

Second, not all nonzero pairs are stored for each thread, but only the highest scoring (that may be 10, 20 or 50, a judgement call), saving space. The heuristic calculation process gets its name from it relying on a rule of thumb: we don't expect the similarity scores to change very quickly. By that I mean more concretely that we don't expect a thread being in, say, top 5 in similarity to a certain other thread if it wasn't in the top 20 or 50 the last time we checked. I we want to find the top 5 and recommend them, recalculating the scores of those who were top 20 some time ago is probably good enough.

The heuristic calculation process goes through the complete list of threads over and over again, looking if they have received new posts since their stored paired similarities were calculated, updating these scores in that case.

4.4.3 Evaluation of first plan

This scheme was suggested by me and discussed with the client/supervisor at a meeting in early October, and it was deemed to have its strengths and its weaknesses. Using the cosine similarity would make the most of all the available information and give us fairly high quality

similarity judgements. Also, the heuristic calculation would help alleviate some of the performance issues with the heavy complete calculation.

But overall, the scheme had too many drawbacks. The database administrator expressed concern that the heavy SQL queries used to find nonzero thread pairs in the complete calculation would lock up the central database if constantly running (as it was planned to do) reducing performance for the whole site. We also came to the conclusion that having updatability depend on the complete calculation running in a reasonable time was unwise, and that the similarity table would still have 20-100 million thread pairs for the heuristic calculation to go through, possibly making it too slow as well. It was also quite questionable to have the complete process calculating lots of similarities that it didn't save. That power could be used better.

The system hadn't been built, let alone tested, but the recognition that it was quite sensitive to likely performance problems led to its abandonment at an early stage. Luckily, we could conclude that most of the need for excessive constant recalculation to keep the data updated rested on a single decision. A decision we weren't very hesitant to change.

4.5 Second plan

Following the meeting where the first approach was rejected, my job was to craft another suggestion, addressing the shortcomings of the first.

4.5.1 Similarity measure

The second similarity measure represented a change in the balance between two values. The first was fine distinctions between various threads when it came to similarity with a given other thread, the other was the possibility of updating values quickly.

In this application there are a lot of items and the number increases constantly. In addition, the newest items are the most important, since it's them that users are most likely to be reading (so there must be matches similar to them available). The weakness of the first approach was updatability, which needs to be better. The strength of the first approach was the quality of the rather subtle similarity measure. Using cosine similarity enables us to compare the relative contributions of different users and properly rank the thread-to-thread similarities where the number of posters in common is equal or close to equal. But this ranking might not be critical. If there are an equal number of posters in common between to threads, and we recommend ten threads every time, their exact ranking isn't of the greatest importance. Cosine similarity requires that the similarity score is completely recalculated when a new post is made in any of the threads, since the different users' contributions to the total score is not independent of each other. This necessarily reduces update quickness, its only upside being a minor quality improvement we might not need.

The natural simplification is then to ignore the number of posts by each user in each thread and just count any posting as "consumption" and consider the data binary consumption data. Presence of a user in a thread counts as a confirmed positive relationship between the user and thread, absence counts as nothing. Considering the large number of threads this binary data is necessarily very asymmetric.

Given this, the data becomes easily updated. Every time a user posts in thread *t* for the first time, all threads that user has previously posted in gains in their similarity score with *t* and it can be updated without concern for its previous composition. No intricate dependencies, no need to recalculate scores completely. And there is no need to update the score for every post, only for the first one for a user in a thread.

The similarity measure we need is similarity between binary vectors, since every thread can be a binary vector with one component per user, indicating whether that user has posted or not. Then the question is, what measure. Simple Matching Coefficient counts all matches equally, and 0-0 matches (which would be the vast majority even for the most populous threads) represent information of a much lower quality than 1-1 matches (it is barely information at all). So SMC is out due to the strong asymmetry, Jaccard coefficient counts only 1-1 matches, and divides by total number of non-0-0 components. In plain language, Jaccard coefficient for two threads would be the number of users having posted in both threads, divided by the number of users that have posted in any of the two threads.

Whether this is good or not is difficult to know. Jaccard Coefficient somewhat penalizes long threads (threads with many posters) since any poster in one thread that doesn't also post in the other thread reduces similarity between the two. Jaccard is simply a measure of the "proportion" of posters that the threads have in common, and shorter threads have it easier (so to speak) to have a high proportion of posters in common just by chance. To instead use a modified SMC or equivalently conceive of the threads as sets of users and use a simple set proximity measure of users in common, would on the other hand penalize short threads, not being able to have as many 1-1 matches or absolute amount of users in common. A middle way was chosen at first, where Jaccard was used, but not dividing by the number of posters in both threads, but by its square root.

If I may skip ahead, this decision was changed later, after some preliminary testing. There were several reasons for this. One was that the length compensation didn't change the order of threads that much, indicating that its benefit wasn't great. Skipping length compensation completely (that is, having the measure simply be a count of posters in common) also reduced computation time significantly (by about 25-30%), since without it there is no need for the querying and insertion of thread lengths into every thread pair representation, and no need to perform the division and inserting of final score for every row in the database (which reaches the hundreds of millions) both in the first calculation and every subsequent update. We also saw some justification in this, with respect to the concerns about quality and favoring long threads. There isn't necessarily anything wrong with favoring long threads, since they have shown to be popular. Also, short threads are not short because of some external constraint, but because they have attracted less interest. Saying that simple match counting is unfairly biased against short threads because they can't get as many 1-1 matches isn't really right because nothing prevents them from getting more, they just haven't got them. Additionally Lee, et al. (200x) showed that for implicit data vector magnitude compensation isn't necessarily a good idea and simple dot product with no upper limit can outperform cosine and Pearson.

So far I have described the measure that was used (what I called simple match counting) as just counting the users in common between two threads. That isn't what was actually done. Just like in the first approach, some idf-inspired user weighting was used. Weighting users mean that, when adding up all 1-1 matches (remember, a 1-1 match refers to a user having posted in both threads), it won't be a sum of a number of ones but a sum of a number of user weights, weights being smaller for prolific users. How should users be weighted? Looking at a basic idf-inspired weighting

$$w_u = \frac{1}{\ln t_u}$$

where w_u is the weight and t_u the number of threads consumed, graphed, we concluded that it somewhat overvalued the importance of very rare consumers since it declines fairly sharply in the beginning and only mildly after that. We were after a more gentle, more linear slope, and instead chose the other

$$w_u = \frac{1}{1 + \frac{t_u}{h}}$$

where h is a parameter. The h value indicates the point where weight would be halved in comparison to a hypothetical user with no threads. Compare slopes in figure 2 (h is set to 50, the value that was eventually used):

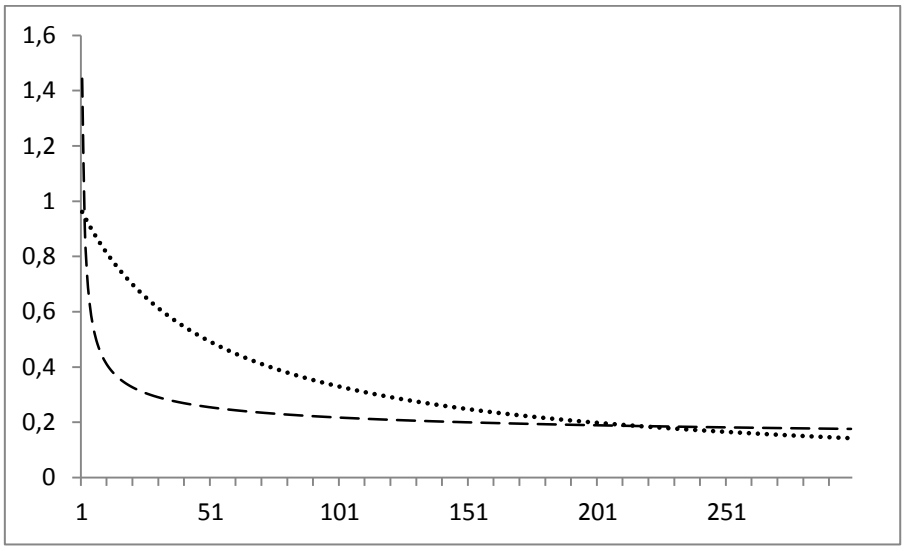

Figure 2, comparison of user weighting depending on number of threads between inverse logarithm (dashed) and other (dotted).

Now, weighting users like this may raise some legitimate concerns. If posting in a thread for the first time reduces a user's weight somewhat, shouldn't the weight from that user that constitutes part of already calculated similarities also be reduced, if the similarity measure is always supposed to follow the stated formula? In theory, yes. But, we're using a rule of thumb here: every new posting doesn't change the weight very much. Cutting some corners in this way, not changing weights already baked into calculated similarities immediately, were

simply judged to be worth it since details on that level are not time-critical and the performance benefits greatly. But there's a limit, and a policy of redoing the complete calculation from scratch every now and then to keep weights reasonably accurate is prudent. Unlike the first approach, however, the basic updatability of the system does not depend on this recalculation.

5.2.2 Suggested implementation

The basic system structure, language, tools and such, was retained from the first approach. In addition to the similarity measure, the processes running also had to change.

Unlike the first approach, the second is implemented as two separate processes, not three. And they do not depend on each other running concurrently. The first is the complete calculation, which can be done very simply, thanks to the forum_member_usage table mentioned earlier (thanks to that, there would be no need for a thread representation process like the one outlined in the first approach). Each row in that table represent a user posting in a thread, and accessing it would easily allow the calculation of user weight (just count the rows with that user and plug the number into the formula). It's also easy to list of all the threads posted in by a certain user, as well as all the users posting in a certain thread. Exactly what algorithm provides the most efficient calculation of all nonzero thread-to-thread similarities was something of a later question discussed in the next section.

When finished, the next issue would be to keep the large table of similarities updated -- to add to similarities that were strengthened, and insert those that were created, by new posts. This monitor process would take over after the complete calculation was finished. It relies on the useful characteristic of the forum_member_usage table; it being a user-thread pairing, adding a new row every time a user posts in a thread for the first time. Since first posts are what matters, updating is easy: get the new rows since the last update and add user weight to all new pairs, wait for a moment, then repeat. The monitor process would only need to keep up with the rate of new posts making any performance problems there very unlikely.

5 Implementation process

As mentioned, the system was written as a Java application. It went through several stages with many changes on the level of implementation details, and only those parts of its development with theoretical considerations will be treated here (unlike user interface issues, error handling and such which took the majority of the actual development time).

The first stage was development of the complete calculation process. I set up a development and testing environment at home, installing a MySQL database server and filling it with test data from 1,000 threads in the forum, represented in the forum_member_usage table. I also installed data from the same threads from the forum_views table, possibly being able to use views as well as posts, with the same treatment of data. Everything mentioned above about posts and posting can also be applied to views and viewing, depending on how we define consumption. Views are of course more numerous, and getting the system to work with posts only was first priority. Views were indeed used in the end, after a new table having been

constructed, the "forum_member_views" table, in the same form as the forum_member_usage one, so they could both be used easily.

Along with the MySQL server I installed the JDBC library and driver, and used NetBeans IDE 6.8 to write the application.

One algorithm was used at first, which was then replaced by another that performed better. The first was based on going through the threads, the second by going through the users.

5.1 Thread-by-thread algorithm

5.1.1 Structure

The first way to write the complete calculation process was to use an algorithm straight from an Amazon industry report (Linden, 2003). It was developed as a way to find only the nonzero pairs when constructing a similar items-table with many expected zeroes. It went:

```
for all threads
    for all users that posted in thread
        for all threads posted in by user
            calculate thread-thread similarity
```

Implementing this in Java was fairly straightforward. The program started by getting a list of all thread id:s in order and stepping into one thread at a time, creating an int-to-double hashmap in which to keep a set of all other threads with which it had nonzero similarity. After that the hashmap was filled by going through every user, calculating their weighting, getting a list of their other threads and inserting the weight as the value where the thread id was the key. If the thread was already present in the hashmap (if another user had posted in both threads before) the value was added to the already extant one rather than inserted.

When one thread had been completed like this, an iterator over the hashmap was created and the thread id:s and similarity values was exported to the database. Then followed the next thread.

5.1.2 Performance

This algorithm finished in manageable time – about 25 seconds when applied to the test data using posts. Views took about three minutes, on my low-powered netbook laptop. It didn't seem troublesome at first, but some back-of-the-envelope calculations showed there was cause for concern.

The complexity of the algorithm is above linear. For every thread we call up every poster, and for every poster we call up every thread they've posted in. The execution time is after that proportional to the number of threads that user has posted in, stepping back, this is done for every user in the thread, stepping back again, this is done for every thread. This makes the whole algorithm proportional to the number of threads (t), times the average number of users in every thread(n), times the average number of threads posted in for every user(m). Or:

$$O(tnm)$$

If we go from 1,000 threads to two million, then obviously *t* increases 2,000-fold. Slightly less obviously *n* remains constant, since the test data was 1,000 complete threads. But *m* increases, possibly by as much as 2,000-fold as well, if the test data was a perfectly random sample of threads. It wasn't, it was a thousand of the latest, which means that if user's postings are not uniformly distributed in time, a full 2,000-fold increase in unlikely. But, worst case scenario, *m* is roughly proportional to *t* and the complexity is therefore

$$O(t^2 n)$$

where *t* is going to increase by a factor of 2,000 as we move from the test environment to the real thing. Attempting to find ways to improve performance I examined the time used to perform certain parts of the calculation. Test showed that about 75% of the time was spent making database calls. After an overhaul, utilizing precompiled SQL statements where appropriate and only committing the changes after each thread was complete (and not after every pair insertion) execution time went down about 10-15%. This was something but not enough, and it was because of these performance problems that the second algorithm was developed.

5.2 User-by-user algorithm

Realizing that the most time consuming element in the calculation was the database calls, I tried to find ways to reduce the amount of calls, preferably by eliminating redundancies. I found one, which was that in the first algorithm, the list of threads posted in by a user was queried lots of times. For every thread a user had posted in, the same list of his or her other threads had to be retrieved. Would it not be easier to call each user's list only once?

5.2.1 Structure

The Amazon-inspired algorithm went through the threads in a row, finding and calculating all pairs involving each thread before moving on to the next. As said, this involved querying the list of threads for a user as many times as the number of threads they've posted in. The second algorithm instead focused on taking on one user at a time. It was:

```
for all users
    for all threads posted in by user
        for all threads with higher id
            add user weight to thread-thread pair
```

The implementation of this did away with the hashmap system. It instead took the thread list of every user and stepped through it, on every step inserting the user's weighting to thread-thread pairs consisting of the current thread and every thread behind it in the list. It didn't insert the pairs at the end, but immediately as the list was traversed, committing after every user therefore considering insertion of one user's pairs one transaction.

5.2.2 The monitor

The monitor process, that which was going to keep the database table updated once it was finished, didn't go through the same sort of evolution. Its structure remained basically the same since it wasn't vulnerable to performance problems (se end of section 5.2.2).

The monitor worked by querying from the forum_member_usage table, all rows that had been added since the last iteration of the monitor. That returns a list of thread-user pairings. The process would take one row at a time, query the list of threads the user had previously posted in, calculate user weighting, and go through that list, adding the weight to pairs consisting of on the one hand the new thread just queried, and on the other each of the earlier ones. In short:

```
for all new user-thread rows
    for all threads previously posted in by user
        add weight to thread-thread pair
```

The complexity of that process would be proportional to the number of new rows, which is the number of first posts in a thread by a user, and for every row complexity would be proportional to the number of threads the user has posted in. It's

$$O(rt_u)$$

for the algorithm, where r is the number of new rows. None of these increase faster than linearly, and none of them are likely to be very large (t_u only very rarely reach the thousands). There is some possible performance improvements to make if one wants, for instance, if the same user is present more times than one in the new rows, that users' list of old threads is called up once for every time she is represented. Putting these together could help a little. But since performance issues here are unlikely it's not needed. And if the time between successive iterations is small (default was set to one minute), then such situations are rare.

5.2.3 Keeping track

When the system is running, there needs to be some way of keeping track of what is going on. If the complete calculation halts for some reason, it should be able to pick up where it left off. When we redo the complete calculation to keep the similarity values current, we need to be able to do this while the older version is being used, and so forth. To store all information about what the system is doing, a special database table was created. It has a set of string-int tuples that function like key-value pairs. I will go through the meaning of each.

```
TABLE `sim_info` (
  `name` varchar(25) NOT NULL,
  `value` int(11) NOT NULL)
);
```

```
'use_tn'  /  'comp_tn'
```

These have values 1 or 0, and are short for "use table number" and "complete calculation table number". The similarity table can be called either "sim0" or "sim1", and if a new complete calculation starts the name of the new table will be "sim" plus the "complete calculation table number". The other will be the "use table number" and the php page that queries the recommended threads will first find that number to access the right table.

```
'last_completed_calcP' / 'last_completed_calcV'
```

These two hold the numbers of the last users to be completed during a complete calculation, P for posts-wise and V for views-wise. Calculation with respect to posts and views would be running in parallel. This enables the complete calculation to pick up from where it previously left off without problems, since the database changes from a user's contributions are only committed when a user has been completed.

`'last_id_compP' / 'last_id_compV'`

When the complete calculation runs, it again and again asks for rows from the forum_member_usage table and the forum_member_views table. But these tables are filled with new rows while the calculation is running, meaning that in the end not the same set of rows has been used to calculate all similarities. Because of that, there is no definite point from where to start having the monitor performing updates. The solution is to note the highest row id present when the calculation starts, and not use rows that are inserted after that. These two numbers store the last id:s to use for the forum_member_usage and forum_member_views tables, respectively. They both start at 0, in which case the complete calculation processes find the highest id and insert it. If it starts again it will use the number from the table.

`'last_id_monP' / 'last_id_monV'`

These two values are used by the monitor processes to know which rows are new since the last iteration. Every monitor iteration saves the id number of the last row it processes, querying only rows with higher id:s the next time.

`'comp_in_progress'`

This is a 1/0 flag indicating if a complete calculation is in progress. When the complete calculation program is started, it examines this flag and if it has a 1, it starts working from the last completed user as indicated by last_completed_compP or last_completed_compV. If it has a 0 then it checks the comp_tn value and creates a new table with that number, then starts work on a new complete calculation.

When the complete calculation is finished, the user is prompted about switching over to use this new table. If yes, a number of things happen: use_tn and comp_tn are switched and the old use table is dropped; last_id_monP and last_id_monV are replaced with the last_id_compP and last_id_compV values, before those two are in turn set to 0; last_completed_compP and last_completed_compV are set to 0 as well; finally comp_in_progress is also set to 0.

5.2.4 Performance

Apart from only having to query a user's list once, the second algorithm only had to calculate user weighting once. There was one downside however, if thread-thread similarity was based on more than one user, then the weights would be sent to the database once per user, not once per thread-thread pair, possibly increasing the number of calls.

Whether the advantages outweighed the drawbacks depended on which of these effects were larger and which of them were likely to grow the most as the amount of threads grew from

1,000 to two million. The advantage was fewer calls to find users' thread lists, the number of such calls in the first algorithm being proportional to the average number of threads posted in by each user, that is, to *m* in the formula above. In the worst case this means proportional to the number of threads, *t*. The drawback of the second algorithm was the increased number of calls per pair, which is proportional to the average number of 1-1 matches (every insertion of a user weighting to a pair means there's a 1-1 match in those threads' binary vectors). That is not going to increase as fast as *t*, since average thread length is going to put an absolute ceiling to the number or possible 1-1 matches, and average thread length remains the same.

The second (user by user) algorithm cut processing time by about a third on the test data. The database calls' share of the execution time climbed from 75 to about 90%, also indicating that hashmap overhead had been considerable. The first was never tried on the full data, but the reasoning above suggests it would have been disastrously time-consuming.

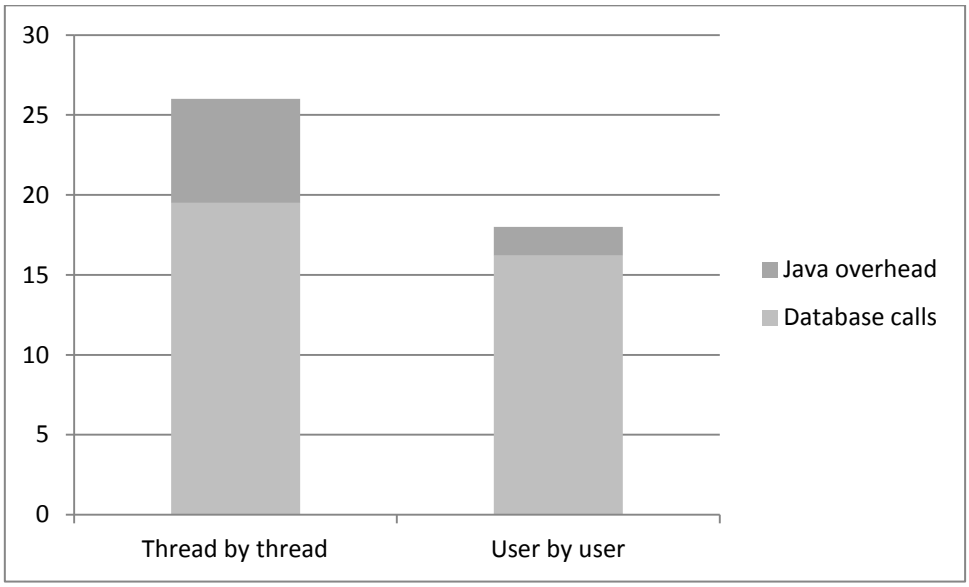

Figure 2, comparison of approximate execution times on 1000 threads (in seconds).

At this point a second test was performed, this time with 10,000 threads. This took an unexpectedly long time, about one hour and ten minutes. That was more than the expected hundredfold increase that constituted the worst case scenario (as *t* increased tenfold). Two things explained this; one was that this time querying and inserting of thread lengths was done for the first time (remember, this thread length compensation was only abandoned at a later stage and at this time a weighted variant of Jaccard coefficient was still used), as well as the division operation being done on every pair. The other reason that execution time grew more than expected was that it would grow proportional to m^2 only after *m* was 2 or more. In the 1,000 thread test data, many of the users had only posted in one of the threads, therefore not having their weighting added to any pair at all. One thread leads to no pairs, two threads leads to one pair, and 0^2 is much less than half of 1^2.

The complexity of the new algorithm was proportional to the number of users, plainly, and for each user proportional to the square of the size of that user's thread list--the number of pair one can make from a thread list of size *s* being $s^2/2$. This means

$$O(um^2)$$

when it comes to complexity (u is number of users). Not that different from the first. It was still likely that m was going to increase 2,000-fold. The difference lay rather in the efficiency of the implementation than the complexity difference of the algorithms. A weakness was that u also was going to increase when all of the data was used, but only by a factor of about 30 (from about 10,000 users in the second test set to about 300,000 in the full data), and the proportionality was only linear. That the second algorithm held the edge over the first, was mainly because of efficiency in the implementation, as said, but also from the fact that m is, in absolute terms, so much smaller than t.

By the first test with the full data of 2,2 million threads, it became abundantly clear that some form of data reduction was needed. Even in the beginning, the pairs generated from a single very prolific early user (id: 2) took over fifteen minutes to insert and numbered to more than 250,000. Some technical difficulties prevented further exhaustive testing for a while, but when those wrinkles were sorted out and a hiccup-free test run was conducted it ran for a couple of days without interruption and was halted deliberately after having processed a quarter of the users and inserted one and a half billion rows, this time running locally on the high-powered main database server of the site.

With this we estimated a total running time of possibly several weeks and a massive table of some 6 billion rows in the end, representing about 3,000 pairs per thread on average. This being based on posts only (before we even touched the much larger views table), data reduction seemed necessary.

5.3 Data reduction

After some research and hard thinking on data reduction techniques, the three main possibilities were considered in turn.

5.3.1 Dimensionality reduction

One possible route to take would be to somehow reduce the number of threads or users in the data set by grouping them. Performing Principal Component Analysis on the data would bring about a set of variables being composites of users, rather than users themselves. This would reduce the amount of components in the vectors representing the threads, essentially going through a series of weighted groups of users rather that single users, improving performance. But the PCA itself would be computationally expensive and complicated and it's unclear whether it would be preferable to the full calculation itself. If the PCA would only be performed once it could help performance on subsequent recalculations, but those are not time critical anyway. This in addition to inevitable quality loss, PCA did not seem to be the right choice.

The other main type of dimensionality reduction would be clustering, of either users or threads or both. Clustering users would mean that we could go through hundreds of clusters of users, rather than hundreds of thousands of users. Data sparsity could be alleviated, since there would not need to be a 1-1 match of an individual user to add 'similarity points' to a pair. A 1-1 match between users of the same cluster would suffice. This would lead to pairs

getting scores based on more generously interpreted data, meaning more pairs per thread but a lower average quality of those pairs. There's a trade-off here, and user clustering could be worth it if execution time and sparsity were to be judged more important than quality. But it is uncertain if time would be saved, because the clustering itself--like PCA--would be costly, going through all 2,2 million threads noting user-user similarities much like the complete calculation is intended to do for threads; and on top of that, using a hierarchical clustering algorithm to group the 320,000 users from pairwise similarities.

Clustering threads would be even worse, since it would necessarily start by establishing thread-thread similarities, which is what we want in the end. There would be little point in following that with a hierarchical clustering only to, in the end, get a lower quality version of what we already had.

5.3.2 Sampling

Sampling the data might be more promising. Maybe selecting 10% of the users randomly and use only them would be enough? But maybe not. If the number of items were smaller than or at least roughly equal to the number of users it might work, if the number of users having consumed each item were large. But since there are about seven times as many threads as there are users, and the number of users per thread not always being very large, using a small random sample of users would lead to severe data sparsity problems, where many threads (possibly most) would be completely without matches.

The other option could be to sample threads. Random sampling wouldn't make sense, but directed sampling were only the most popular threads were part of the system could be possible. One drawback would be that there would be no basis for making recommendations when a user was reading a (not very popular) thread that was not part of the system, neither would such threads ever be recommended even though they ought to sometimes. The same problem occurs if we'd followed through with another possibility, limiting the system to fairly new threads. Lots of users come to the site through search engines, finding old threads, and the system should preferably work in such situations.

What we eventually did try, was directed sampling of users. The thought behind it was this: very prolific users, having posted in many, many threads, have two qualities relevant to the system: their weighting score is low, because them participating in a thread does not give as much information as in the case of a more restrained user; and the load, performance wise, is heavy for these users. A user with many threads in their list gives rise to a number of pairs proportional to the square of the number of threads, while the weighting is low. Therefore, if a thread count limit was imposed, above which the user weighting was "rounded down" to zero (i.e. those users ignored), only low quality information would be lost and performance improved disproportionally much. Also, by adjusting the limit, the trade-off between quality and performance could be controlled. Another boon would be that a limit could be used to render the large amount of views data usable with a well selected limit.

One might imagine a ratchet-like system, where you'd start with a low limit, getting a table of reasonable quality and use that while calculating another one with better quality, switching to

that in turn and start on an even better one, and so on. In that case an execution time in weeks or even months would not necessarily be fatal.

5.3.3 Partitioning

Another approach altogether would be partitioning. In this case, not every pair of threads would be inserted into the database, but only those were both are in the same "category" (the definition of category being open for discussion). This would reduce the number of thread pairs that needed to be considered and inserted into the database (the most time-consuming set of operations), saving time and reducing the massive table size.

The question was into which categories threads should be sorted. Not wanting to do expensive calculations to establish categories, there were three main ideas. From "top" to "bottom": The first option was to subdivide by community. Recall, communities are the main sub-forums on the site, 15 different broad areas of discussion. The second option was a bit more complicated. I thought it might be a good idea to cluster the 361 individual forums. These forums are subdivisions of the larger communities and concern more specific topics. Treating these forums like threads and going through the users calculating forum-forum similarities in the familiar way could, if followed up by a hierarchical clustering algorithm (cheap for 361 data points) provide good clusters to use as partitioning categories. The risk was, of course, that using this technique would fail to remove many thread-thread pairs and not improve performance that much. I wrote a program to calculate forum-forum similarity, but it was never used since the third approach was deemed good enough.

The third approach was simply to partition based on the individual forums. The vast majority of forums were large enough for this, and there was a certain sense to it. If a user has clicked on a certain forum, are we not to assume that they are interested in that topic? Recommending only from that forum reduces the risk of recommendations being "way off" and makes it more likely that they are decent, even if the similarity score isn't that high. It also meant that there wouldn't need to be any special rules made for some of the forums that were about very serious topics and where recommendations of "frivolous" discussions from other forums would be inappropriate. In more ways than one, this was a "safe" avenue of action. It did require that the forum_member_usage and forum_member_views tables were altered to include the forum id as well, but this was easily done by the database administrator.

5.3.4 Final choice

Data reduction was in the end applied in two different ways. Partitioning by forum was done for both post-based and view-based measures. View data was much greater than post data, so to get comparable execution times, directed sampling was also imposed on the views system with a limit of 400, that is, users with a record of viewing more than 400 threads had their weighting rounded down to 0 and skipped. In the end, with both posts and views used, we got 670 million rows. It should be mentioned that the views information had not been retained ever since the start of the forum, and such data did not exist for the first couple of years. This means the similarity scores involving threads from these years would be underestimated, but this was something we would have to accept. Very old threads are visited much less anyway, so their importance is less great.

5.3.5 Performance after reduction

The time requirement for the complete calculation and the ability of the system to keep itself updated as new data is being added are all examples of offline performance. After the reduction this offline performance was deemed acceptable, as a complete calculation could be done in a matter of days and the updating could keep up. The other side of performance was online. That is, the performance when the system was used, not when it was built. The site has a lot of page views and the proper threads would need to be displayed quickly and not make the site grind to a halt. By the first try to display recommendation lists at all thread showings, it became apparent that it was straining for the database server, it bogged down the site and something had to be done about it.

The problem was this: the similarity scores for a thread pair was split up into two values, one based on posts and one based on views. The ranking was to be made on the sum of these two. The sum was not represented explicitly (since it would require up to 700 million additions) but rendered by the SQL query that fetched the list to be recommended. This meant that the list of pairs with nonzero values (which could be quite large) had to be sorted by a dynamically generated value (the sum of the two columns) for every page view. It was heavy for the database server and inefficient, since if the final score could be indexed, the highest values could be retrieved much faster.

One approach was to introduce the sum into its own column and put an index on that, but this was slow as well and recalculation of the whole table seemed like a better idea. The recalculation would need to fill an altered table however, to not suffer from the same problems. One could be to perform the addition when calculating, but I opted for another approach. The reason that there were two separate columns for posts and views data was that we wanted to be able to try out different weightings between the two. It had now become clear that dynamic adding of the values was untenable anyway, so there was no benefit any more to keep them apart. Therefore a new set of programs were made to add points to the same column, regardless of those points came from post-based or view-based matches. We could then index that column. The recalculation set back the testing schedule a bit of time, but it was necessary.

6 Live testing and evaluation

When it was time to perform the first proper live test (actually using the system on the site and measuring its effectiveness), we had a similar items table completed.

6.1 Test scheme discussion

One concern that had been present ever since the first version of the system was run was how good the recommendations were and how good they'd need to be to be usable. Usable is difficult to define, and getting a good overview of the data wasn't trivial. It could simply be tested, noting what similarity scores were associated with clicks on links to recommended threads. But that would, in the end, serve to establish a lower limit to how high a similarity score would need to be for one thread to be recommended when the other in its pair was being read. That could simply be tested directly, by varying the lower limit randomly and noting

which limit gave rise to the most clicks, on average. The great traffic to the site of about a million page views per day enables quick and accurate randomized trials (for a discussion of the great value of randomized trials, see (Ayres, 2007)).

It was unclear however, what the benchmark would be. Why would possibly having a lower limit (showing more recommendations) result in fewer clicks? It could in the long term if low-quality recommendations erode trust in the system. But this is difficult to test, and a more easily evaluated baseline, or benchmark, would be to compare the system to the current one. The current system is the ”hottest discussions” list that displays the most written-in treads of the last 24 hours in a sidebar. One way to test and calibrate the new system would therefore be to create a similar list of fixed size (the hottest discussions list has 6 entries), fill it with recommended threads with a similarity score of over the lower limit, and if there are spaces left, add the top entries from the hottest discussions list. A randomization scheme, run by the php that creates the pages that display threads, would select one of, say, ten possible lower limits and construct a list based on it. The number of clicks on any of the links would then be counted. This would mean there being ten rows in a special table in which the number of clicks on lists generated using a particular lower limit would be counted.

An example: if the row marked “1” for instance, in this table, had the value “10,000” it would mean that there during the test had been 10,000 clicks on lists where the lower limit 1 had been used. This doesn’t mean much in itself, but if the corresponding row “2” has 20,000 clicks we can assume that 2 is a better lower limit to use (given that a randomization scheme with rectangular distribution would have displayed as many lists constructed from a limit of 1 as from 2).

A high lower limit would mean on average less recommendations from the similarity system and more from the hottest-discussion-list, and a low one the opposite. The optimal lower limit does therefore represent the cutoff point where the quality of similarity based recommendations falls below the quality of popularity/newness based recommendations.

But this scheme wasn’t used. Replacing the current hottest discussions-list with a composite of it and the new system was judged to be undesirable and risky by upper management and we had to construct a less intrusive testing scheme.

There was some confusion about what was to be tested when we no longer could test the cutoff point between the old system and the new. There were a number of possibilities, put forward by both myself on the one hand and the client/supervisor on the other. Some differences in perspective revealed themselves, where the client/supervisor favored a more exploratory approach while I made the case for a more rigorous experimental design that would provide more solid information for system optimization, along the lines of the first scheme. We did however manage to converge and find a good design.

The main options were to, on the one side, measure the efficacy of the old and new system by using them alongside one another and count the number of clicks on their respective recommendation lists. I objected to this design on the grounds that since the lists were displayed at the same time only the relative click rate of them when both present would be

noted, and that this would not help us compare the result of different choices such as not using the new system, using both, or using only the new system. The use of randomized trials can help isolate the consequences of various decisions from each other and not using it would not be the best use of our time.

A suggestion of mine was to (through randomized trials) test five different ways of measuring similarity by weighting posts-similarity and views-similarity through five different weightings (posts only, views only, 50-50 and 75-25 in both directions). This wasn't done for understandable reasons, the client/supervisor wasn't very interested and it would perhaps be a little too detailed.

There was still a good case to be made for establishing a lower limit for similarity scores. But without the possibility of "filling up" with entries from the hottest discussions-list, the list would only be shorter when the limit was high, and how could a shorter list give rise to more clicks? The cost of having longer lists would need to be seen as being more abstract, as eroding confidence in the system if it shows bad recommendations. Finding a cutoff point, not between the new and old system but between the new system and nothing at all was a bit dubious as to its value. I seriously questioned the use of this, perhaps to the point of pedantry as seen by my client/supervisor. But in the end we found a suitable benchmark; the database administrator suggested that we could use the newest threads in the current forum as "filling", this would yield a comparison not between the new system and nothing but the new system and threads from the forum whose similarity to the current thread were essentially random.

The solution to evaluating the new system as a whole (rather than finding the best lower limit) was also a sort of compromise between exploratory and rigorous methods. One option (rigorous) was to alternate and show the list generated by the new system only on half of the page showings (leaving the old system in its place) and see whether the presence of the new system increased clicks compared to its absence. The other option (exploratory) would be to display both systems all the time and count the clicks on each system's list, rendering some idea of how good the systems were in relation to each other. The option that was finally used was exploratory in the sense of using both systems all the time, but rigorous in the use of randomization to compare them as well as possible. It involved putting the two lists in a sidebar, on above the other, but using 50-50 randomization to determine which was above and which below to compensate for a possible placement effect.

6.2 Test architecture

The site had a finished system for click-recording in place which we were able to use, which meant that we could save time not having to write it from scratch. We were to select ten different minimum similarity scores that would act as candidates to a final cutoff point. It wasn't obvious which ones to select since the similarity scores had no maximum value (at this point the similarity scores were simple sums of weights of the users in common). The solution was to cast a wide net in the beginning and do a more focused second test based on the results of the first.

The first series of cutoff values tested were 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5 and 100. The top value of 100 was there to provide a baseline where no threads at all were to be shown from the system. The results were to be recorded in the table

```
version_hits('name','version','url','clicktime')
```

where 'name' indicated whether the click was on the list of the old system ('hottest discussions') or the new; 'version' which cutoff was used, 'url' which thread was the target and 'clicktime' was, quite self-explanatory, the time. With this scheme we could easily, by the end of the test period, count the clicks on different variants.

6.4 Test results

The test was run for one week and over a total of ten million page showings. The ten different variants where randomly displayed, meaning they were shown about a million times each. The evaluation metric is CTR ('click through rate') – the share of showings that leads to a link being clicked. The results clearly show that the new system is superior, with a CTR roughly double that of the old.

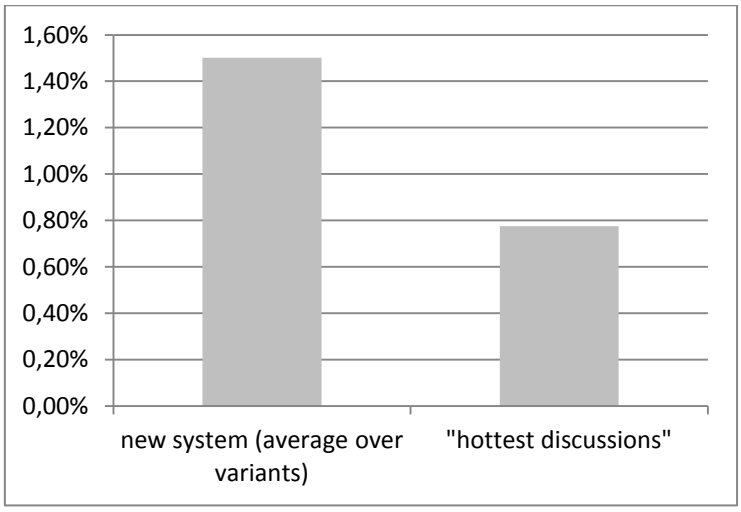

Figure 3, CTR of new system compared to extant

But not so fast. This might look great, but the difference could be due to the "hottest discussions"-list having more or less the same threads showing for long periods, showing a lower variety of threads. If the new system is about a 100% improvement, how much of that is actually due to the quality?

The ten different cutoff points illustrate this. If quality has an effect, then lower cutoff points (associated with showing many recommended threads and few of the newest ones) should show higher CTR than the higher ones, and especially the control condition where the cutoff point is 100 and virtually no recommended threads are shown.

It turns out that the quality effect is only a small part of the improvement, but it exists. Lower cutoff points are better, and the lowest (0,5) has a CTR about 13,75% higher than the control condition.

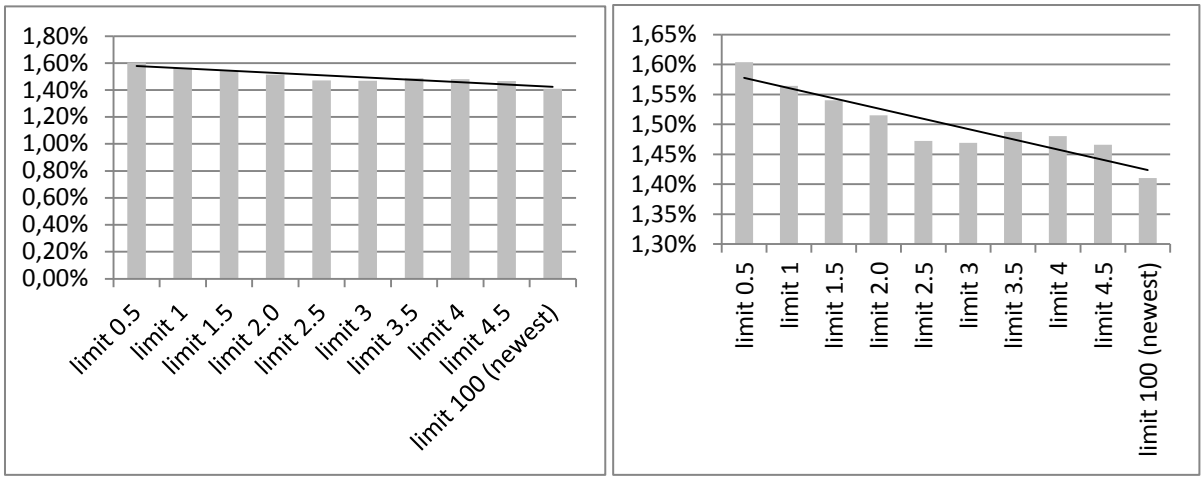

Figure 4, CTR by variant (left), with differences magnified (right)

7 Conclusion and future work

The system does represent a large improvement. Most of that improvement, however, is apparently due to the increased variety of threads being recommended. The benefit of the system itself, is most fairly estimated at about 13,75% percent, counted in CTR increase.

It's difficult to determine what sort of expectations I had regarding the effectiveness of the system. Spontaneously, an 13,75% improvement feels like a moderate success. If just increasing the variety of recommendations with respect to the previous system's configuration yielded such great improvement, then 13,75% seems like a bad return on investment. But taken alone it is certainly acceptable.

What more could be done? Other things could be tested, such as the effectiveness of post-based and view-based similarity measures (rather than composite) or different user weightings. If a new system were to be created from scratch, perhaps a general quality-based system would be a good idea, considering that the newest threads from the current forum (essentially random) get a good result based on variety only. Perhaps selecting random threads from the same forum, their likelihood of selection weighted by some general quality measure could improve the system even more, at a smaller cost in terms of computing power and time.

7.1 Answers to questions

Where does that leave our questions? Are we now able to answer them? Yes, perhaps.

The previous discussion led to the conclusion that the available data lent itself ideally to cosine similarity over such measures as correlation or Minkowski distances. With unlimited capacity and performance a non-issue, cosine would be preferable. But here is where the constantly changing data comes in. This constraint in combination with data abundance means that we need a similarity measure where 'new similarity points' can simply be added onto existing ones, not requiring complete recalculation of the score the way cosine works.

The modified Simple Matching Coefficient that was eventually used adressed this issue without compromising quality significantly. Not compensating for thread length (and thereby not using Jaccard Coefficient) was prompted by the demands that come from working with changing data, but can also be said to bring its own quality benefits. Similarity doesn't go from 0 to 1, but from 0 and up. This makes numbers hard to interpret, but interpretation is not the primary function of the number. It's comparison. And this scheme weighs in both the strength of the similarity and the certainty of the estimate. A high number means a lot of users in common, not just a high proportion, and that favors more certain relationships.

8 References

(n.d.). Retrieved 3 15, 2011, from SQL.org: www.sql.org

Adomavicius, G. T. (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* , volume 17, issue 6.

Ayres, I. (2007). *Super Crunchers: How Anything Can Be Predicted.* John Murray.

Ball, P. (2003). Critical Mass: How One Thing Leads to Another. Heinemann.

Google. (n.d.). *Google Technology Info.* Retrieved 12 15, 2010, from http://www.google.com/corporate/tech.html

Hu, Y. K. (2008). Collaborative Filtering for Implicit Feedback Datasets. *IEEE International Conference on Data Mining.*

Lee, T. Q.-T. (2007). A Similarity Measure for Collaborative Filtering with Implicit Feedback. *Lecture Notes in Computer Science* , volume 4682.

Linden, G. S. (2003). Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE* , volume 7, issue 1.

MySQL. (n.d.). Retrieved 10 5, 2010, from MySQL: http://www.mysql.com

Netflix. (n.d.). *Netflix Prize.* Retrieved 10 16, 2010, from http://www.netflixprize.com/

Salton, G. B. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management* , volume 24, issue 5.

Su, X. K. (2009). A Survey of Collaborative Filtering Techniques. *Advances in Artificial Intelligence* .

Tan, P.-N. S. (2006). *Introduction to Data Mining.* Pearson Education.