Integrating Solutions to Solving the Cold Start Problem in the Wikipedia Recommender System

Casper Kristiansen

Daniel Shanti

Kongens Lyngby 2013
IMM-M.Sc.-2013-17

Technical University of Denmark
Informatics and Mathematical Modelling
Building 322, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Wikipedia is a free online encyclopedia that can be edited by anyone. Due to its open nature, it can be difficult for readers to assess the quality of the articles, since the articles may contain errors due to vandalism or simply due to lack of knowledge by the editors. Wikipedia Recommender System (WRS) is a collaborative filtering system that provides readers with individual rating predictions for Wikipedia articles, based on ratings given by other users for that article. The ratings are weighed according to the similarity with the reader's ratings on other articles. The purpose of the ratings is to give the reader an indication of the quality of the articles and thereby increasing trust in Wikipedia articles.

However, the system is currently a prototype and as such it has no real users or any user generated data. When readers start using WRS, the data in the system is expected to be sparse and as Wikipedia contains a tremendous number or articles, a reader may be required to rate a high number of articles, before WRS is able to provide useful predictions for the reader. This phenomenon is well known within collaborative filtering systems, and is known as the *cold start problem*. The purpose of this thesis is to investigate solutions to the cold start problem.

This thesis focuses on integrating two techniques for mitigating the cold start problem. The first is to use data from the external service WikiTrust which calculates ratings for Wikipedia articles by determining and computing a trust value for the content. The second technique is to depend on the concept of *similarity propagation*. This is utilized when a reader requires a predicted rating for a given article but the reader has given only a few ratings, and none of the user's ratings are similar to those of someone who has rated the article in question. In this case WRS may determine that the reader is similar to another WRS user who is similar to a third WRS user that can be used to predict a rating.

The result of this thesis is the prototype of an improved version of WRS which is far more capable in the cold start situation. With this improved version of WRS, readers of Wikipedia are given an advanced tool that can help them determine the quality of Wikipedia articles in advance and the tool has the potential for helping the millions of internet users that visit Wikipedia every day.

Preface

This thesis was prepared at Department of Informatics and Mathematical Modelling, within the Technical University of Denmark in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering.

The project was completed in the period from November 9th, 2012 to March 15th, 2013 under the supervision of Associate Professor Christian Damsgaard Jensen.

Lyngby, March 2013

_____

Daniel Shanti

s082941

Lyngby, March 2013

_____

Casper Kristiansen

s092816

Contents

List of tables

List of figures

List of code listings

1 Introduction

1.1 Introduction

Wikipedia is a multilingual, web-based, free-content encyclopedia project operated by the Wikimedia Foundation and based on an openly editable model[1], which means that anyone can add or edit articles. Using this model, Wikipedia has grown to be immensely popular since it was launched in 2001, and at the time of writing, wikipedia.org is the 6th most visited website in the world according to statistics from the web information company Alexa[2].

The number of articles in Wikipedia grows rapidly and today the English version alone contains more than 4.1[3] million articles. Much of the popularity of Wikipedia stems from the open nature of Wikipedia, which allows anyone to add or edit articles in the encyclopedia. Wikipedia relies on a number of automatic measures to avoid vandalism and defacing of articles, and also a large number of voluntary editors who review both new articles and changes to existing articles.

However, this openness also presents major problems for the trustworthiness of Wikipedia articles, because there are no guarantees that the articles were written by unbiased or even qualified contributors. Wikipedia has tried to mitigate the problem of trustworthiness by adding the concept of "featured content". Featured articles have undergone a thorough review by Wikipedia editors and are "considered to be the best articles Wikipedia has to offer"[4]. However, at the moment only 3,772[5] of the

[1] http://en.wikipedia.org/wiki/Wikipedia:About
[2] http://www.alexa.com/topsites
[3] At the time of writing this, 2013/01/02 it contains 4,134,047 articles.
[4] http://en.wikipedia.org/wiki/Wikipedia:Featured_articles
[5] 2013/01/03

English articles are featured. This amounts to about 1 in every 1,090, and thus this concept alone is not sufficient to make Wikipedia as a whole a trustworthy source.

1.2 Wikipedia Recommender System

One way to make Wikipedia's articles more trustworthy is to introduce a simple way of providing user feedback. Wikipedia Recommender System (WRS) is a collaborative filtering system which has evolved over the course of five master theses at DTU. The purpose of the system is to provide predicted ratings for articles that are unknown to the WRS user. The predicted ratings are based on prior ratings by the WRS user.

The current versions of the system use a central server to store the ratings from all WRS users, and utilize ratings given by other WRS users to find similar users. The basic idea is that if user A has agreed with user B on the ratings of articles X and Y, it is likely that they will also agree on the rating of article Z which only one of the two users have rated. WRS also asks the users to decide the category of the article being rated and takes this information into consideration when it attempts to find similar users.

A significant challenge for a collaborative filtering system such as WRS is that it faces the so-called "cold start problem" when introduced. The cold start problem refers to the situation where the system contains no or very few ratings. In this situation the system has no way of predicting ratings, since there may be no other users who have rated the article that a specific WRS user is reading. Earlier theses have focused on trying to mitigate the cold start problem by introducing an external entity into the system which returns a rating for any article [Mihăilă, 2011], or by supporting recommendations which accelerate the expansion of a user's network of known or similar users [Andersen, 2011].

1.3 Objectives

The main objective of this thesis is to integrate and improve previous solutions to the cold start problem in WRS. This integration will be based on the findings of the previous theses concerning WRS, and will focus on the last two versions by Andersen [Andersen, 2011] and Mihăilă [Mihăilă, 2011] that diverge in their implementations in general and their approach to mitigating the cold start problem.

The analysis will ultimately lead to the design and implementation of a functional system which provides the users of the system with meaningful feedback. It should work as a helpful tool in determining the quality of an article on Wikipedia, by delivering realistic article ratings to the user.

The WRS implementation from Andersen is based upon the idea of building trust by not only considering direct interactions, but also indirect interactions i.e. interactions through a chain of users [Andersen, 2011]. As an example, the system could consider an interaction between a user A and another user C, where A has only interacted indirectly with C through a third user B.

The implementation is based on a recommendation concept, and the idea of trust propagation can in principle continue through any number of users. By including indirect interactions the network of users

rapidly expands with an increasing number of interactions and this will accelerate the rate at which the network grows. The increased number of interactions provides a much greater coverage of articles and aids in computing an article rating.

While this network grows much faster, the system still faces the cold start problem every time a new user tries WRS for the very first time, since such a user will have no previous interactions which the system can base predictions on.

The WRS implementation from Mihăilă takes a different approach in trying to solve the cold start problem by including an external entity in the form of WikiTrust[6] [Mihăilă, 2011].

WikiTrust is an open-source reputation system created for the MediaWiki software and is developed at the Computer Science Department at University of California Santa Cruz. WikiTrust provides an API for interacting with Wikipedia, which is the largest project using MediaWiki, and it has been shown to provide a usable article rating based purely on the edit history of the article [Mihăilă, 2011].

The objective of integrating the trust propagation by Andersen [Andersen, 2011] with WikiTrust by Mihăilă [Mihăilă, 2011] should not only improve the quality of the rating calculated by WRS, but also increase the likelihood of the system being able to compute it in the first place. In the previous versions of WRS, the system is only able to predict a rating for a specific article if the trustor reading the article has had positive interactions with one or more trustees who have rated that specific article.

The number of ratings available for predicting article ratings can be modeled as a function of the number of interactions of a trustor. In the previous version of WRS by Pilkauskas [Pilkauskas, 2010], one interaction between the trustor and a trustee will allow for rating prediction based on the number of articles that the trustee has rated (minus the one article on which the interaction was established, but this is disregarded for simplicity). This can be modeled using a simple function for linear growth:

$$f(x) = ax$$

Here *f(x)* is the total number of ratings available for predicting article ratings, *a* is the average number of ratings per user, *x* is the number of interactions that the trustor has had.

The function above clearly results in linear growth which is what could be expected when the number of ratings available for prediction directly follows from the average number of ratings per user and the number of users the trustor has interacted with.

Mihăilă improved this by utilizing the WikiTrust API [Mihăilă, 2011], where even if the trustor has had no previous interactions, WikiTrust can still predict a rating for all articles. This means that WikiTrust provides a constant number of ratings equal to the number of articles in Wikipedia. This is modeled by adding the constant b to the equation above:

$$f(x) = ax + b$$

[6] WikiTrust is a reputation system for Wikipedia, developed at UC Santa Cruz, http://wikitrust.soe.ucsc.edu/

Andersen took another route and sought to include trustees of the trustor's trustees [Andersen, 2011]. This idea can be expanded to include trustees that are an arbitrary number of links away from the trustor, effectively making the number of ratings available for rating prediction grow polynomially. Including additional levels of trustees will change the number of trustees to be a power of *n*, where *n* is maximum number of links away from the trustor that is still to be considered an interaction:

$$f(x) = ax^n$$

This thesis will combine the two ideas into a single solution, which means that the number of ratings available for rating prediction will follow the equation:

$$f(x) = ax^n + b$$

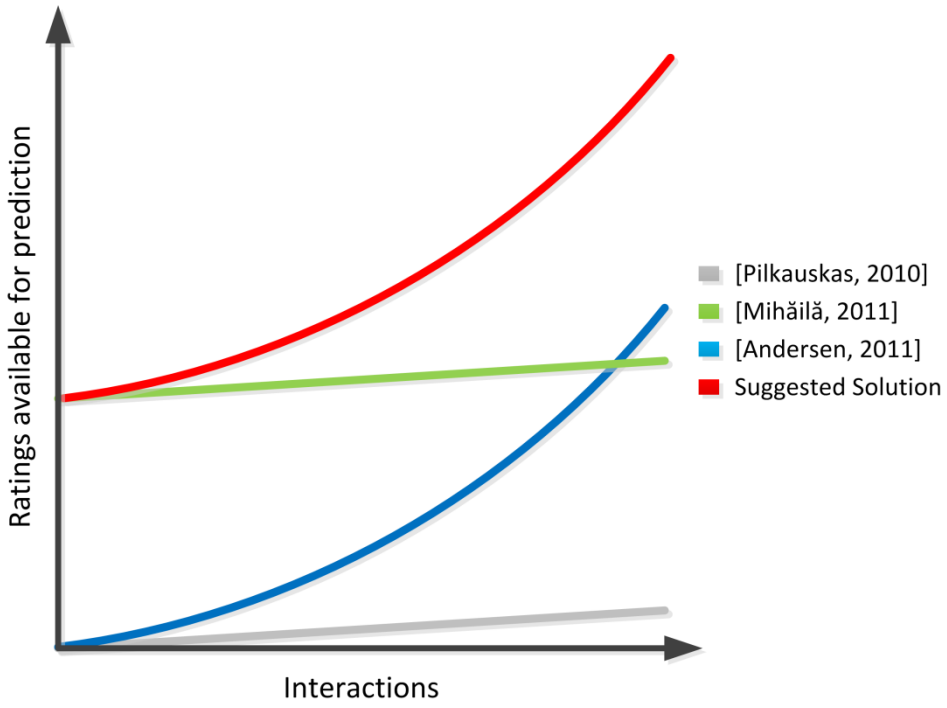

Figure 1 - Growth in number of ratings available for prediction in different versions of WRS.

The final objective of this thesis is to reuse parts of the back-ends of both theses and in particular the front-end in the form of a browser plugin [Mihăilă, 2011]. This plugin proved to be vastly superior to the Java application based on the Scone proxy that has otherwise survived through several iterations of WRS, but experiments have shown it to result in a serious performance bottleneck [Pilkauskas, 2010].

1.4 Structure

This thesis follows the structure outlined as follows:

Introduction describes the WRS project along with the objectives of this thesis.

State of the Art provides an overview of the current state of WRS and the theory behind it. It also describes Wikipedia's new rating system which is an alternative to WRS.

Analysis provides a thorough analysis of the current theoretical foundation of WRS, and highlights drawbacks of the current trust model. It continues to analyze how the idea of trust propagation applies to WRS and presents an alternative to the existing trust model. It also analyzes how to integrate the two previous versions of WRS into a new and improved system.

Design goes through the process of designing a solution to the problem of predicting ratings based on similarity propagation, while considering the complexity of the problem. It analyzes the running time and the space usage of an effective solution to the problem.

Implementation describes our implementation of the improved version of WRS. The most important components are discussed in detail and a general overview of the solution is provided.

Evaluation shows the state of the project after the proposed changes and highlights individual achievements. It also contains descriptions of the automated tests that have been included in the project in order to assess and verify the quality and functionality of the new version of WRS.

Future work and research enumerates the design choices that were not based on a scientific foundation or experimental data, in order to highlight values and choices that may benefit from optimizations when WRS passes the initial phase. It also describes some future tasks that could lead to an even more useful version of WRS.

Conclusion summarizes the results achieved in this thesis.

Appendix contains relevant resources used in the project.

1.5 Definition of Terms

In this thesis it is assumed that the definition of a number of terms is known. These terms are listed below along with a short description.

WRS: An abbreviation for the Wikipedia Recommender System, which is the name of the system that this thesis is centered around. It was originally created by Thomas Rune Korsgaard in 2007 and has since been the center of other theses, namely those by Thomas Lefevre, Povilas Pilkauskas, Mihai Mihăilă and Natasa Popovic Andersen.

Rating: A numeric value in the range of [1, 9] denoting the quality of a Wikipedia article.

Interaction: When two users have rated the same article on Wikipedia, the two users are said to have had an interaction.

Trustor: The user that shows trust in another user.

Trustee: The user a trustor shows trust or distrust in through a previous interaction.

Recommendation: A term introduced by Andersen in order to describe the concept of a user publicly sharing his or her trust in another user [Andersen, 2011].

Trust Value: A numeric value in the range of [-1, 1], showing how much a trustor trusts a trustee, ranging from complete distrust to complete trust.

Web of Trust: The trustor's Web of Trust is the network of trustees that a particular trustor has had direct or indirect interactions with. The Web of Trust is not limited to trustees that the trustor has positive trust in, but also includes distrusted users.

RDBMS: Relational Database Management System; A database management system based on the relational data model, which is the predominant type of databases.

2 State of the art

In this chapter we will describe both the theoretical foundations and the implementations of the previous versions of WRS, being the result of five different master theses at DTU. The first three versions were developed sequentially, with each subsequent thesis incorporating the work of the previous. The fourth and fifth versions were developed in parallel and their implementations diverge from their common starting point.

We will also briefly describe *WikiTrust* and *Wikipedia's Article Feedback Tool* which are two external services that are closely related to WRS.

2.1 Trust model

The purpose of WRS is to provide a trustor with a reliable predicted rating each time the trustor views a new Wikipedia page. The rating should be based on feedback from other users and should be similar to the rating that the trustor is likely to give the article, when he or she has read it. In order to achieve such a rating, the system will consider ratings by other users whom the trustor has previously agreed with.

WRS operates with the concept of *functional trust*, where a trustor may trust a trustee to provide useful ratings. This functional trust arises from similarity of ratings given by trustor and trustee. The trust is considered *functional* because the trustor does not explicitly state his or her trust in the trustee, but the trust stems from the fact that the trustor and the trustee have previously agreed on the ratings of Wikipedia articles.

To model this concept of trust, WRS utilizes a model suggested by Stephen Marsh [Marsh, 1994] which represents trust as a continuous variable over a specific range. In WRS the range is [-1,1] where -1 signifies complete distrust while 1 signifies complete trust.

The trust function in WRS describes the development of trust between trustor and trustee. It is based on the model of trust dynamics by Jonker and Treur [Jonker, Treur, 1999] which has the following three main parts:

1. Initial trust
2. Trust dynamics
3. Trust evolution model

2.1.1 Initial trust

In WRS a trustee which a trustor has had no interactions with is given the initial trust value 0. This signifies that a rating given by this trustee has no value to the trustor. However, as soon as the trustor has interacted with this trustee, the trust value of this trustee will either increase or decrease. An interaction with a trustee occurs when the trustor rates an article that has previously been rated by the trustee. The interaction can positively affect the trust value of the trustee if both trustor and trustee has given the same rating (Or a rating within a certain threshold), or negatively affect the trust value if the ratings are dissimilar.

Once an interaction has taken place, WRS will place the trustee in the *Web of Trust* of the trustor, which means that ratings from that trustee can be considered in the future if the trustor visits Wikipedia articles that have been rated by the trustee.

2.1.2 Trust dynamics

Trust dynamics describe the development of trust over time and as interactions occur. In WRS the development of trust is based on a weighted time definition. This means that the older an interaction is, the less weight is given to the ratings of that trustee. The weighted time definition has varied over the development of WRS. In the current version from Mihăilă, interactions that are less than a month old count 100%, those between one and six months count 50% and the ones between six months and one year counts 25% [Mihăilă, 2011]. Interactions more than a year old are ignored.

2.1.3 Trust evolution model

The trust evolution model consists of a trust evolution function and methods for adjusting the function. A basic trust model function could increase the trust value of a trustee with a specific value each time a positive interaction happened between the trustor and the trustee. This would result in a linear trust evolution function.

However, in order to adjust the trust evolution function according to the behavior of the trustor, the trust function of WRS introduces the concepts optimistic and cautious curves for the trust evolution function. This is achieved by asking the trustor two questions:

1. What is the rating of the article on a scale from 1 to 9?
2. Was the predicted rating from WRS satisfying or not?

The value of the rating will determine if the trust values increase or decrease and the question of whether or not the rating was satisfying or not will give an idea of what the fault tolerance of the user is. This information can be used to adjust the trust evolution function [Korsgaard, 2007].

The trust evolution function of WRS is based on the formula of a super ellipse with radius 1:

$$|x|^n + |y|^n = 1$$

Where *x* is the sum of interaction and *y* is the calculated trust value. The value of *n* gives the curvature of the curve and is initially set to 1. The value is updated based on whether the user is cautious or optimistic. The value *n* changes in steps of 0.1 depending on if the trustor states that he or she is satisfied with the rating from WRS or not.

The following four equations describe the four possible states that a trustor can be in, with regards to a trustee. Which of the four functions is used depends on the behavioral nature of the individual WRS user i.e. if the user is cautious or optimistic and in trust or distrust:

Optimistic curve in trust:

$$|x-1|^n + |y|^n = 1, where\ x\ \in [0.0, 1.0] and\ y\ \in [0.0, 1.0]$$

Cautious curve in trust:

$$|x|^n + |y-1|^n = 1, where\ x\ \in [0.0, 1.0] and\ y\ \in [0.0, 1.0]$$

Optimistic curve in distrust:

$$|x|^n + |y+1|^n = 1, where\ x\ \in [-1.0, 0.0] and\ y\ \in [-1.0, 0.0]$$

Cautious curve in distrust:

$$|x+1|^n + |y|^n = 1, where\ x\ \in [-1.0, 0.0] and\ y\ \in [-1.0, 0.0]$$

2.2 Classification

Apart from the two questions mentioned in the previous section, WRS asks the trustor to decide the category of the article to be rated. This is because WRS is based on the notion that a trustor's trust in a trustee is contextual, and is associated with a category. The idea is that even though a trustor may trust a trustee within for example sports articles, the same may not be the case for articles about politics.

WRS contains two metrics when rating Wikipedia articles: The rating and the category. The idea is that the rating metric defines the quality of the article, whereas the category metric defines the user's understanding of the article.

The rating is the primary metric because the quality of the article is considered to be more important, while category is the secondary as it depends on the individual user's ability to perceive information.

The combination of the two metrics affects the weight of the ratings from each trustee in a trustor's Web of Trust. The weights are defined in the following table:

Weight	Rating	Category
0.5	Agree	Disagree
1	Agree	Agree
-1	Disagree	Agree
-1.5	Disagree	Disagree, and trustee has assigned a category that is different from the one assigned by the majority of raters of the article.
-0.5	Disagree	Disagree, and trustee has assigned the same category as the majority of raters of the article.

Table 1 - Weight table for various combinations of agreement on rating and category

If trustor and trustee agree on both rating and category, the rating carries full weight. If they agree on the rating but disagree on the category, it still carries some weight. Otherwise it carries various degrees of negative weights.

The reason the negative weight is lowest when trustor and trustee disagrees on both rating and category *and* the trustee has assigned a different category than the majority of raters of the article, is that in this situation it is likely that the trustee has misunderstood the article and therefore the trust in this user should become even lower. If they disagree on both category and rating, but trustee has assigned the same category as the majority, it is likely that it is the trustor who has misunderstood the article, and in this situation the rating should not carry as much negative weight.

Povilas Pilkauskas investigated what classification scheme would be optimal for classifying Wikipedia articles [Pilkauskas, 2010]. Through surveys it was found that Open Directory Project - Dmoz classification scheme was superior to the other potential classifications schemes (Citizendum workgroups, Dewey Decimal Classification classes and Wikipedia top-level portals). This classification scheme scored highest in both users' general experience value and also proved to give the most reliable categorizations when several articles were categorized by a group of users.

2.3 Trust propagation

WRS as a tool faces the problem of containing no ratings at all, when it is released. This problem is called the *cold start problem* and the consequence is that WRS cannot give any meaningful ratings to its users until it has been in service for a while.

Even after the system has received ratings on many of its articles, it may still not be very helpful to a new WRS user. This is due to the fact that in order for WRS to be able to provide a rating to some trustor A, there must be at least one trustee B with whom A has had a positive interaction through some article that they have both rated *and* B must have rated the article that A requires a rating for. Considering the number of articles that Wikipedia contains, this may be a somewhat unlikely situation, and the further division of articles into categories does not increase the likelihood of the situation occurring.

Andersen looked into the concept of trust propagation in her master thesis about WRS [Andersen, 2011]. Trust propagation is based on the idea that trust is transitive within some certain boundaries, such as a specific Wikipedia article category. One example could be that a trustor A trusts a trustee B to provide ratings for articles in category X. Using trust propagation, a rating given by a third user C which B trusts within the category X may also have value for A. Andersen discusses the idea of having transitive chains of trust which can dramatically increase the speed at which a trustor expands his or her Web of Trust.

Two existing algorithms, TidalTrust and MoleTrust, are analyzed in order to determine which of the two trust propagation algorithms would be optimal within WRS. The thesis proposes to use the MoleTrust algorithm within the WRS for the following three reasons:

1. Its simplicity
2. Execution time
3. Support for adjusting the trust propagation horizon and the minimal trust value

2.4 WikiTrust

Mihăilă takes another approach to reducing the cold start problem. Instead of relying on trust propagation he includes article ratings from the service WikiTrust when the Web of Trust is of no help to a WRS user. This way WRS can always provide a rating, even for a trustor with no ratings at all.

WikiTrust is an open-source, online reputation system for Wikipedia authors and content[7]. It works by analyzing the actions of Wikipedia authors and determining the *implicit trust* which follows from the actions. A *reputation* is calculated from the implicit trust by examining the *text life* and the *edit life*. The system was examined and determined to provide useful reasonable ratings for both low quality articles and for articles which live up to the highest standards of Wikipedia, such as Featured articles [Mihăilă, 2011].

[7] http://wikitrust.soe.ucsc.edu/

2.5 Wikipedia's Article Feedback Tool

After the two newest master theses were completed, Wikipedia released its own rating system to a wider audience. The tool has actually been in development since July 2010 and was launched in September 2010, although it remained relatively unnoticed as it was only available on approximately 700 articles. Over the course of versions 2 and 3, the number of articles expanded to 3.000 and 100.000 respectively. The 4th and current version is said to be live on all articles on the English Wikipedia[8], although we did find pages which, for unknown reasons[9], did not contain the feedback tool. In the current version the system lets the user rate an article on the following four characteristics:

1. Trustworthy
2. Objective
3. Complete
4. Well-written

On each of these the rating is given on a scale from 1 to 5. In addition to this it also lets the user indicate if he or she is highly knowledgeable about the topic of the article and if so, where that knowledge comes from such as profession, hobby or a college/university degree.

Wikipedia's article feedback tool is conceptually different from WRS because it provides ratings that reflect a global average of the ratings of every person who have rated a specific article. This is in contrast to WRS where a predicted rating is based only on the ratings of similar users.

2.6 Current architecture

This section will describe the current architecture of WRS. Since there are two individual current versions, this section will describe the key features of both versions.

Both current versions of WRS are based on the previous version by Pilkauskas in which Wikipedia.org acts as the central server that hosts the ratings and user profiles [Pilkauskas, 2010]. In order for a user to utilize WRS, the user must have a Wikipedia user profile. The user must also have a client application installed on his/her local computer. This application contains a web proxy which intercepts requests for http://wikipedia.org. The contents of the HTTP response is parsed and modified by the client application and a calculated rating value is injected into the Wikipedia HTML before it is displayed in the user's browser.

The following diagram from Mihăilă's master thesis illustrates the behaviour of WRS in the previous version:

[8] http://www.mediawiki.org/wiki/Article_feedback#Project_history

[9] Pages of current president of USA Barack Obama and former president George H. W. Bush both do not contain the rating tool as of January 16th, 2013, whereas pages of George W. Bush and Bill Clinton both do.

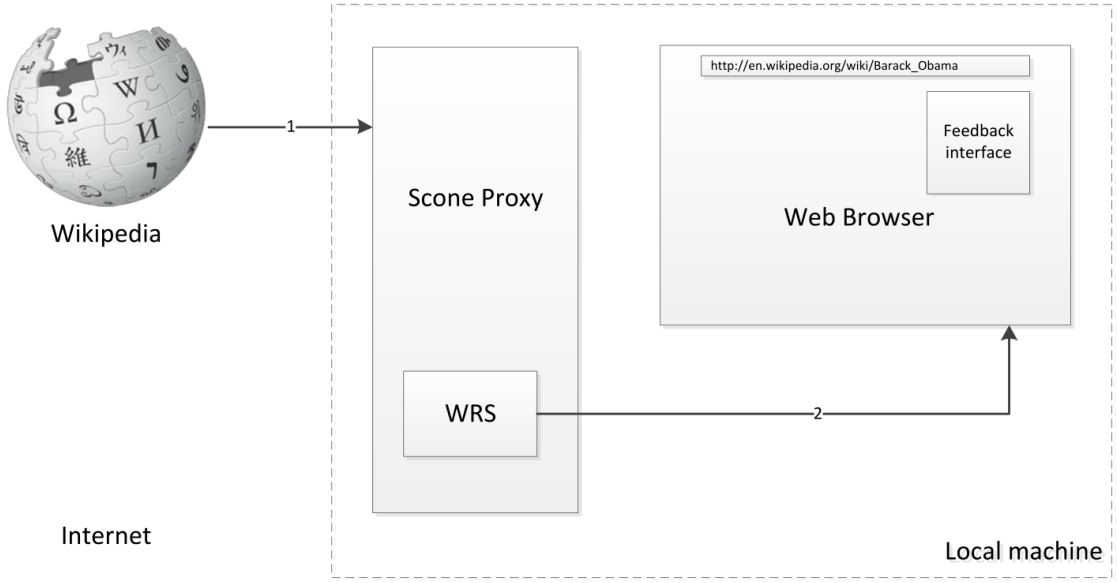

Figure 2 - Behaviour of the previous version of WRS

The HTTP response (1) from Wikipedia is parsed by the WRS module of the Java based Scone Proxy and an altered version of the HTML page is passed on to the web browser (2). In the example the Wikipedia page about Barack Obama is displayed in the user's browser along with a feedback interface which the WRS user can use to rate the article.

In order to provide a rating for the user, WRS calculates a rating value based on the interactions between the user and other WRS-users. Depending on the nature of the interactions WRS calculates a trust value from the user who visited the article last (trustor) and the user who visited the article first (trustee).

However, using Wikipedia User pages as a central repository for ratings was incompatible with the automatic vandalism protection methods of Wikipedia. This eventually led to WRS being banned from editing Wikipedia and both Mihăilă [Mihăilă, 2011] and Andersen [Andersen, 2011] were forced to seek a solution to this problem.

Both Mihăilă and Andersen mitigated the problem by using a web server along with a database as a centralized repository for data storage while exposing data using web services.

However, the exact implementations diverged from a simple MySQL database [Mihăilă, 2011] and to the slightly more advanced solution based on Virtuoso Universal Server[10] [Andersen, 2011].

In the version by Andersen, Virtuoso allowed for the ratings to be stored in a relational database much like MySQL which provides easy means of data extraction using SQL queries [Andersen, 2011]. Virtuoso

[10] Virtuoso Universal Server is an enterprise grade multi-model data server by OpenLink Software. It offers functionality for RDBMS, RDF, Linked Data, Web Application/Services deployment among others.
[10] http://virtuoso.openlinksw.com/

also allows for exposing a set of RDF views using SPARQL[11]. The added RDF functionality was a key point in choosing this particular server technology. Andersen also introduced the idea of recommendations of other users in parallel with user ratings, such that a user can share parts of his or her Web of Trust with other users [Andersen, 2011].

Recommendations of other users seek to accelerate the rate at which a user's network grows, by incorporating the *Friend of a Friend*[12] concept. In this version of WRS, the Scone proxy based implementation was kept as a central part of the system.

In the version by Mihăilă the author paid great interest in moving away from the Scone proxy, by highlighting numerous problems with the technology [Mihăilă, 2011]:

1. It slows down the browsing tremendously as a result of bridging data through the proxy
2. Conflicted with other software running on the client
3. Ignores `No-Cache` in HTML and thus requires manually clearing cache
4. Installing WRS with Scone is not trivial
5. No debugging with IDE, only message dumps to console

In order to replace the Scone proxy, a light-weight browser plugin for the Google Chrome browser was developed. This plug-in provides the same basic functionality while being less intrusive and easier to install and distribute through the Chrome Web Store.

While the plugin itself does not directly address the cold start problem, it improved the overall user experience of WRS. In order to mitigate the cold start problem, the system utilizes the WikiTrust API. This ensures that even for new users, WRS will always return a rating, albeit not based on the previous ratings by the trustor.

The following diagram illustrates the architecture of both systems. The version by Mihăilă [Mihăilă, 2011] is illustrated on the left side and the version by Andersen [Andersen, 2011] is illustrated on the right.

[11] SPARQL is an RDF query language used to interact with databases storing their data in the Resource Description Framework format

[12] The Friend of a Friend (FOAF) project is creating a Web of machine-readable pages describing people, the links between them and the things they create and do. (http://www.foaf-project.org/)

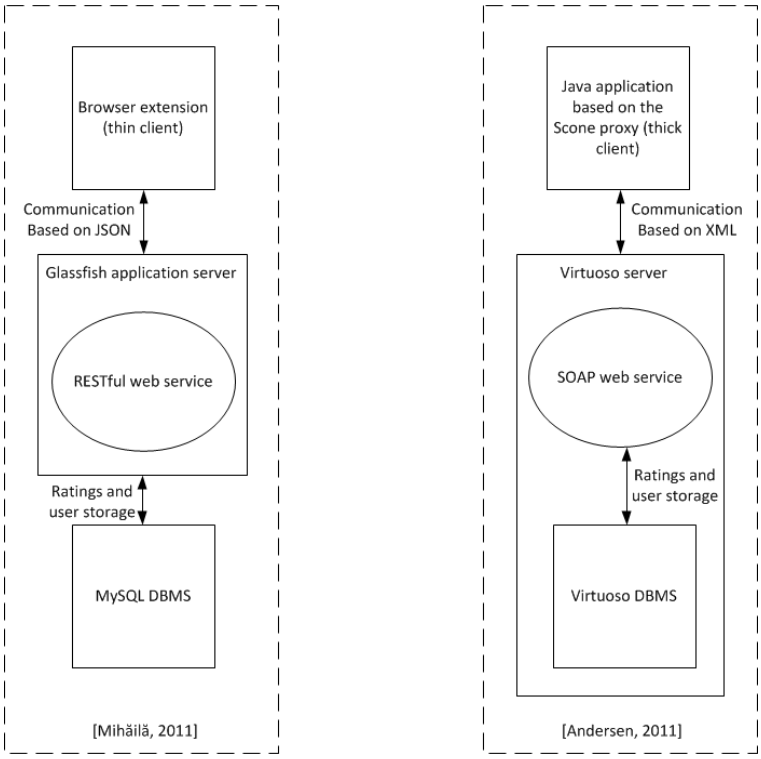

Figure 3 - Architecture of the previous versions of WRS

As evident from the side-by-side illustration of the two architectures, they are quite similar. The two most central differences between the two designs of the system are:

1. The use of a thin client in the version by Mihăilă moves the Web of Trust from the user's computer to the WRS server [Mihăilă, 2011]. This significantly speeds up the browser's loading time and improves user experience but adds more load to the server because all trust value calculations are now done server side.
2. The use of an independent application server and DBMS in the version by Mihăilă makes for a more flexible platform where the servers can be replaced individually in case other servers are deemed more useful at a later point in time. However, using a single server which contains application server, DBMS and many other systems may reduce setup time and avoid any potential compatibility issues between server systems.

However, it is also clear to see that the two systems have diverged significantly on the technological level.

2.7 Evolution of WRS

The first version of WRS was developed by Thomas Rune Korsgaard [Korsgaard, 2007]. It used Wikipedia's article pages to save user ratings.

The second version of WRS was developed by Thomas Lefevre [Lefevre, 2009]. It moved ratings to the user pages at Wikipedia instead of the articles pages. It also introduced rating categories.

The third version of WRS was developed by Povilas Pilkauskas [Pilkauskas, 2010]. It changed the available article categories in order to get a better categorization scheme.

The fourth and fifth versions were developed in parallel by Natasa Popovic Andersen [Andersen, 2011] and Mihai Mihăilă [Mihăilă, 2011]. In the following we denote the version by Andersen as the fourth version and the version by Mihăilă as the fifth version.

The fourth version by Natasa Popovic Andersen [Andersen, 2011] moved ratings to a central WRS server based on Virtuoso Server. This server provided a SOAP based web service which was used by the client application.

The fifth version by Mihai Mihăilă [Mihăilă, 2011] moved ratings to a central WRS server based on Glassfish Application Server and MySQL. This server provided a RESTful web service used by a thin client in the form of a Chrome extension. The application logic of the client application was moved to the server and the client application was scrapped. The cold start problem was reduced by introducing article ratings from WikiTrust.

2.8 Summary

WRS has been developed over the course of five master theses at DTU. The theoretical foundation of WRS comes from a trust model by Stephen Marsh and incorporates a trust function which has support for trust evolution and trust dynamics.

Version four and five restructures WRS in order to avoid using Wikipedia's pages for data storage and moves rating data to a central server. However, the two versions diverge on the technological level and one version introduces the idea of trust propagation while the other introduces the external service WikiTrust in the cold start situation.

Since version four and five were completed, Wikipedia has added their own rating system, however it uses global ratings and is thus still quite a bit different since WRS uses individual ratings.

3 Analysis

This analysis chapter will consist of a theoretical section and a practical/implementation related section.

The theoretical section will start by analyzing the current theoretical foundation of WRS, and will continue to propose a number of changes to this foundation.

In the implementation section we will analyze the two current versions of WRS and describe how to achieve a new version which combines ideas and code from the two versions.

3.1 Theoretical foundation

In *State of the art* we described the theoretical foundation of the existing trust model of WRS. In this section we will further analyze this foundation in order to fully understand the choices and possibly improve the WRS implementation.

The basic trust model consists of the following parts:

- A trust value within the continuous range [-1;1].
- Trust functions
- Trust evolution
- Trust dynamics
- The trust model extensions:
- Classification
- Trust propagation (only theoretical)
- WikiTrust

3.1.1 The basic trust model

The basic trust model is based on four equations which are used to calculate the trust value of a specific trustee from the point of view of a trustor. Which function is used depends on whether or not the trustor is in a cautious or optimistic state, and whether the trustor trusts the trustee or not.

It was based on Jonker and Treur's model of trust dynamics [Jonker, Treur, 1999]. Thomas Korsgaard conceived the idea of using the formula of a superellipse (Also known as a Lamé curve) to model trust [Korsgaard, 2007]:

$$\left|\frac{x}{a}\right|^n + \left|\frac{y}{a}\right|^n = 1$$

Where *n*, *a* and *b* are positive numbers. In WRS *x* in the formula denotes the accumulated trust, while *y* denotes the calculated trust value.

Korsgaard sets *a* and *b* to 1, which results in the following formula [Korsgaard, 2007]:

$$|x|^n + |y|^n = 1$$

He further modifies this formula and splits it up into four different equations which are used within the four different states that trustor and trustee can be in:

Trustor is optimistic *and* trustee has positive accumulated trust (*x* value)

$$|x-1|^n + |y|^n = 1, where\ x\ \in [0,1] and\ y\ \in [0,1]$$

The function $F_{opt\text{-}trust}(x)$ for calculating the trust value based on the sum of interactions *x* for this situation becomes (by isolation of *y*):

$$F_{opt-trust}(x) = (1 - |x-1|^n)^{\frac{1}{n}}, x \in [0,1]$$

Trustor is cautious *and* trustee has positive accumulated trust (*x* value)

$$|x|^n + |y-1|^n = 1, x \in [0,1] and\ y \in [0,1]$$

Function $F_{cau\text{-}trust}(x)$ becomes:

$$F_{cau-trust}(x) = -(1 - x^n)^{\frac{1}{n}} + 1, x \in [0,1]$$

Trustor is optimistic *and* trustee has negative accumulated trust (*x* value). Trustee is in distrust

$$|x|^n + |y+1|^n = 1, x \in [-1.0, 0.0] and\ y \in [-1.0, 0.0]$$

Function $F_{opt\text{-}distrust}(x)$ becomes:

$$F_{opt-distrust}(x) = (1 - (-x)^n)^{\frac{1}{n}} - 1, where\ x \in [-1,0]$$

Trustor is cautious *and* trustee has negative accumulated trust (*x* value)

$$|x+1|^n + |y|^n = 1, where\ x \in [-1.0, 0.0] and\ y \in [-1.0, 0.0]$$

Function $F_{cau\text{-}distrust}(x)$ becomes:

$$F_{cau-distrust}(x) = -(1 - |x+1|)^{\frac{1}{n}}, where\ x \in [-1,0]$$

The *n* value has the following purpose: It changes the curvature of the trust function so a cautious trustor requires a higher number of positive encounters with a trustee before the trust value of the trustee rises to values close to 1. On the other hand, an optimistic trustor requires fewer positive interactions with a trustee before the trustee gains a high trust value.

In the following illustration the four functions are plotted within the specified ranges of the function in the situation where *n=1.5*. The green curve represents $F_{opt\text{-}trust}(x)$, the red represents $F_{cau\text{-}trust}(x)$, the blue represents $F_{opt\text{-}distrust}(x)$ and the yellow represents $F_{cau\text{-}distrust}(x)$.

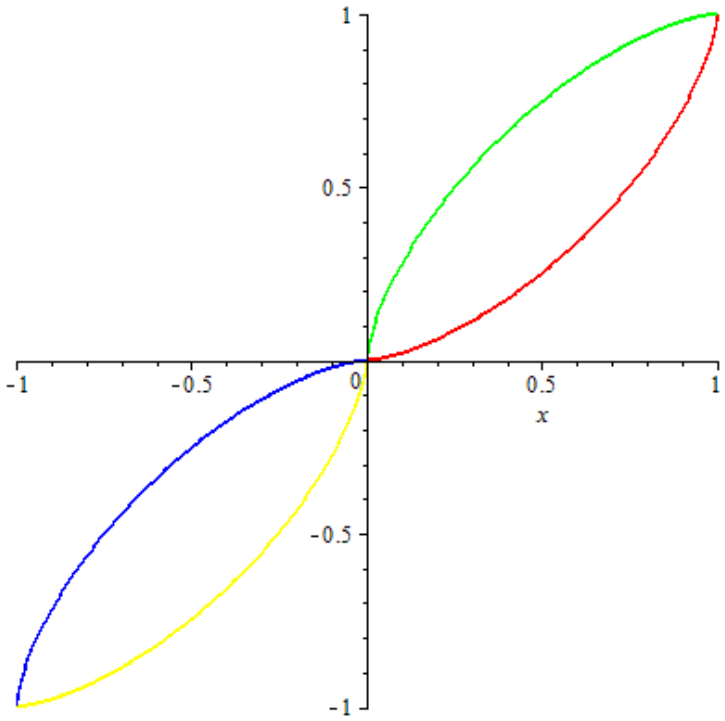

Figure 4 - Plot of the four trust functions for n=1.5

The following plot shows the functions for *n = 2* to illustrate the effect of altering the value of *n*.

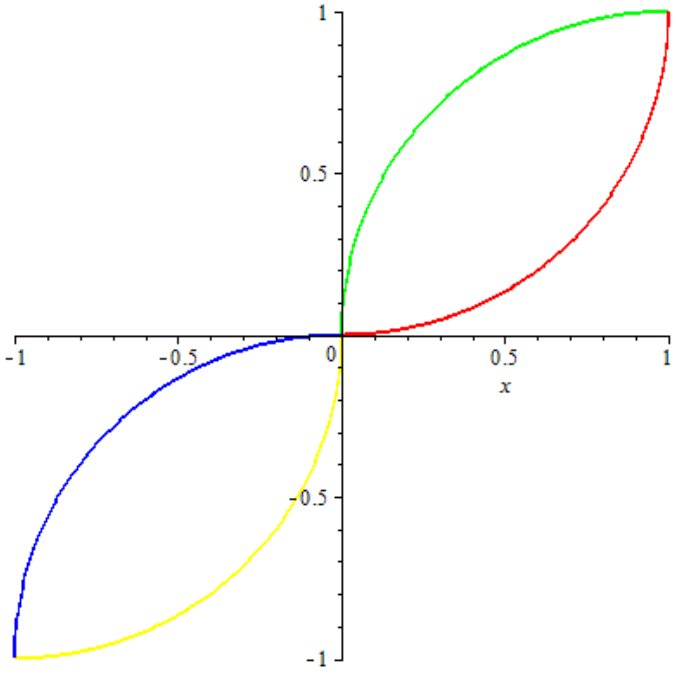

Figure 5 - Plot of the four trust functions for n=2

It is unclear from reading the chapter *Trust Model* from Korsgaard's thesis how the value of *n* changes and in which range it operates for the trust function [Korsgaard, 2007]. It is also unclear when the state of a trustor changes in such a way that another trust function should be used i.e. when the state of a trustor changes between the cautious state and the optimistic state.

When examining the *State of the Art* chapter of Pilkauskas thesis the following clarification is given [Pilkauskas, 2010]:

> *n denotes the curvature of curve, according to curvature, it is possible to determine the characteristics of trustor. The initial value of n is 1 and it could be incremented or decremented by +0.1 or -0.1 subject to the trustor characteristics, this shows whether the trustor is cautious or optimistic. The curve with cautious feature is having n < 1.0, whereas optimistic curve has n > 1.0.*[13]

Mihăilă gives a similar explanation in the *State of the Art* [Mihăilă, 2011]:

> *The n parameter which gives the curvature of the curve starts at 1. This parameter will be updated based on whether the user is cautious or optimistic. n < 1.0 gives a cautious curve, while n > 1.0 gives an optimistic curve.*[14]

However, by analyzing the outcome of the function we find that this interpretation of the original work by Thomas Korsgaard's functions cannot be true. We believe that the proper interpretation is that the *n* value operates in the range:

$$n \in [1, \infty]$$

We base this on the following observations

1. Both the cautious and the optimistic curves can be achieved for values of *n >= 1*, as seen by the illustrations above and similar illustrations in Korsgaard's thesis [Korsgaard, 2007].
2. When isolating the trust value in the four functions defined above, the function will be undefined for *n = 0* because of division by zero.
3. If we plot $F_{opt\text{-}trust}(x)$ for *n=1.7* and *n=0.3* i.e. 7 steps in the optimistic direction and 7 steps in the cautious direction according to Pilkauskas [Pilkauskas, 2010] and Mihăilă [Mihăilă, 2011] we get a function which is asymmetric around the base (neutral) curve where *n=1* as seen on the following plot. The green curve represents *n=1*, the blue represents *n=1.7* and the red represent *n=0.3* for the function $F_{opt\text{-}trust}(x)$:

[13] [Pilkauskas, 2010] p. 21
[14] [Mihăilă, 2011] p. 10

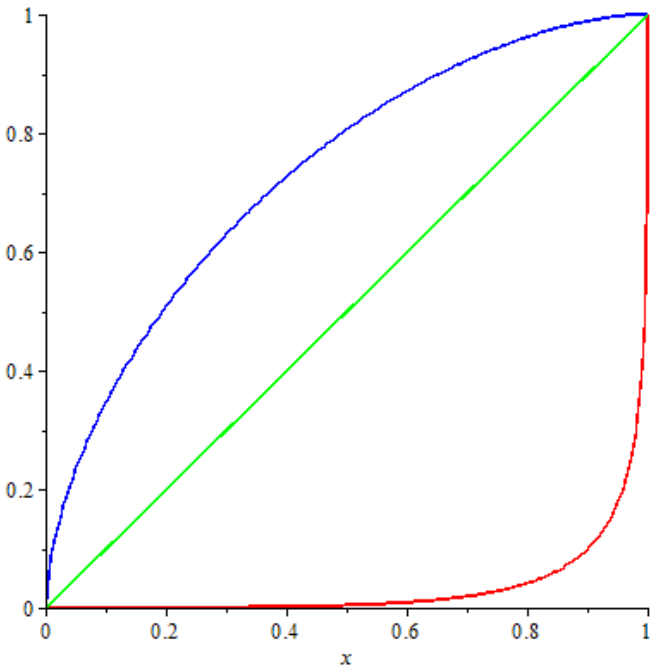

Figure 6 - Plot of $F_{opt\text{-}trust}(x)$ according to Pilkauskas and Mihăilă

In the version of WRS designed by Thomas Korsgaard the value of *x* is calculated as follows:

Each time the trustor has an interaction where he or she agrees with a specific trustee, the accumulated value *x* of this trustee will increase by 0.1. Trustor and trustee agree if the absolute difference between their ratings is at most 1. If trustor and trustee disagree on the rating of another article, *x* is decreased by 0.1. The value of *x* is also affected by the age of each interaction.

The value of *x* determines which of the two pairs of functions is used:

$$f(x) = \begin{cases} F_{opt-trust} \vee F_{cau-trust} & x \geq 0 \\ F_{opt-distrust} \vee F_{cau-distrust} & x < 0 \end{cases}$$

Our analysis of the operating range of *n* can be further supported by examining the source code of the class `Reviewer` related to Pilkauskas's thesis which also contains clues to when and how the optimistic/cautious state of the trustor changes [Pilkauskas, 2010].

The variable `nValue` operates in steps of 0.1 and the curve is initially in the optimistic state and in trust, with *n=1* and *x=0* for an unknown trustee which means the function $F_{opt\text{-}trust}(x)$ is used.

If the next interaction is positive (similar rating and positive *experience*) *n* and *x* will each be increased by 0.1. If on the other hand the interaction is negative (dissimilar rating and negative *experience*) two things will happen:

1. *The value of x* is now -0.1 which means that the trustor is now in distrust about the trustee and one of the distrust functions will be used.

2. Because the trustor is in the *optimistic* state and gets a negative experience, 0.1 is subtracted from *n*. Because *n = 1* and 1 is the minimum value of *n,* the state changes from *optimistic* to *cautious* and 0.1 is added to the *n* value instead, resulting in *n = 1.1* in order to reflect that the curvature has changed in the cautious state.

The general behavior of selecting optimistic of cautious state is as follows:

If the trustor is in the optimistic state and gets a positive experience, the value of *n* increases by 0.1. If the trustor is in the cautious state and gets a negative experience the value of *n* is also increased.

If the trustor on the other hand gets an experience which contradicts with his or her current state the value of *n* is decreased by 0.1 unless *n=1* in which case the state changes to the opposite and instead 0.1 is added to *n*.

The functions as described above include the basic trust model and also the trust evolution, which is handled by the *n* value, which changes according to the stated experience of the trustor.

However, the trust dynamics are not described in the basic functions. The trust dynamics describe how trust changes over time, and in Thomas Korsgaard's trust model this happens in the following way:

When accumulating the *x* value which is used by the trust functions, a recent interaction (up to 3 months old) carries full weight, interactions 3 to 9 months old carries 50% weight, interactions 9 to 12 month old carries 25% weight, while older interactions carry no value at all[15]. The weight of the interaction is multiplied with the original value of the interaction so for example a 10 month old interaction will only count *0.25 * 0.1 = 0.025* in the accumulated *x* value.

3.1.2 Trust model extensions

3.1.2.1 Classification

Classification was introduced in WRS by Lefevre in order to improve the reliability of the predictions that are presented by WRS [Lefevre, 2009]. The idea was that even if two WRS users agree completely on the ratings of articles within a certain category A, they may not agree on articles within another category B.

The classification is considered a secondary metric with the rating value being the primary. This means that if ratings are similar, the trust value will be affected positively even if the two users have assigned different categories. However, in this situation this rating will only carry half weight compared to when the users agree on both rating and category.

The weight is used when calculating the value of *x* used in the trust functions in the same way as described in *Trust dynamics*.

[15] [Korsgaard, 2007] p. 56

As described in the classification table of *State of the Art,* in all situations where the trustor and trustee disagree on the rating, the interaction carries a negative value. Different weights are used when the users disagree on the rating, depending on whether or not they agree on the category and whether or not the trustee assigns the same category as the majority of raters or not.

The extension of WRS related to classification allows the system to provide more reliable ratings, but the price for this is further increasing the cold start problem of WRS. The implication of the extension is that now, in order for WRS to have a foundation for providing ratings, it is no longer sufficient that there exists at least one trustee for which the following is true:

1. Trustee has rated a subset of the articles that the trustor has also rated
2. Trustee has rated the article for which the trustor requires a predicted rating
3. Trustee and trustor has agreed on the rating of at least one article (Or the calculated trust value for the trustor in the trustee is positive)

The third condition is now further refined so that is less likely that the calculated trust value is positive because the category condition makes an interaction where trustor and trustee agrees on the rating but disagrees on the category carry less value. And at the same time, disagreement of both rating and category may carry more negative value than before where the category did not affect the weight.

The result is that it is less likely that relevant trustees with a positive trust value exist, and therefore WRS has less data to use for predicting ratings.

3.1.2.2 Trust propagation (only theoretical)

After careful analysis of the source code of by Andersen we find that trust propagation never actually made it into WRS [Andersen, 2011]. The thesis contains discussions on various algorithms for trust propagation, but only vague descriptions and ideas on how these algorithms could actually work in WRS. An analysis of how to utilize any trust propagation algorithm within WRS is important because it means using an algorithm which is based on explicit trust in a system where this concept does not exist.

3.1.2.3 WikiTrust

WikiTrust is an open-source reputation system for Wikipedia, developed at the Computer Science Department at University of California Santa Cruz.

The following quote from the developers of WikiTrust explains the overall idea behind WikiTrust and describes how it is able to provide meaningful ratings, solely by text analysis.

> "WikiTrust uses a content-driven reputation system: authors gain reputation when their contributions are preserved by subsequent authors, and they lose reputation when their contributions are reverted. The reputation system judges every revision on the basis of many subsequent ones, and it has been built in such a way as to be relatively resistant to spam and vandalism. For instance, users lose only a negligible amount of reputation when vandals remove

> their contributions, much less than they gain from the subsequent revisions once their work is restored.
>
> The trust of a portion of text is computed according to the reputation of the author, and the reputation of the people who subsequently revised the portion of text, and the text immediately surrounding it. Thus, what WikiTrust calls "text trust" is an indication of the degree with which the text has been revised. The trust is computed in such a way that every text change -- insertions, deletions, displacements -- leaves a low-trust mark, which alerts visitors to the modification. It is possible to compute text trust also based on a mix of text age, and number of revisions for which the text has been present; in fact, it is straightforward to modify WikiTrust to do so. The reason WikiTrust uses also the reputation of authors is to prevent a well-organized set of new users from cheating the system, creating content that gains full trust due to their coordinated revisions. Since new users have low reputation, this type of attack cannot be carried out." [16]
>
> - http://wikitrust.soe.ucsc.edu

WikiTrust exposes a public API which can be used to retrieve a calculated rating for an article on Wikipedia solely based on its content and revision history. It relies on the stability of an article and the fact that the longer content remains unchanged over the course of article revisions, then the likelihood of the content being trustworthy increases. WikiTrust traces content back to the revision at which it was introduced or altered and computes a weighted average of the different revisions into a single score from 0-10, which denotes the trustworthiness of sections of an article.

The ratings from WikiTrust cannot be used directly in WRS because WikiTrust operates on content, while WRS operates on articles. WRS comes about this by calculating an average of all sequences of text within the article using the following formula:

$$\frac{\sum_{i=1}^{n} |seq_i| \cdot t_i}{\sum_{i=1}^{n} |seq_i|}$$

Here seq_i is a sequence and t_i is the calculated trust in seq_i. WRS also recalculates from the [0,10] trust scale of WikiTrust to the [1,9] scale of WRS.

In Mihăilă's work it was determined that this way of calculating the rating of an article gave reasonable ratings i.e. high quality pages such as Wikipedia's "Featured Articles" gets a high rating, while obviously low quality articles such as articles about recent or ongoing events get a low rating [Mihăilă, 2011].

WikiTrust was used as a fallback mechanism in situations where the user had no other interactions, and as soon as a single interaction was made, WikiTrust would no longer be in effect. This thesis seeks to improve on this behaviour by analysing how WikiTrust can be incorporated to function alongside user interactions throughout the entire lifetime of the system and thus continue to aid in improving the ratings.

[16] http://wikitrust.soe.ucsc.edu/the-algorithms-underlying-wikitrust

3.1.3 Evolution of the theoretical foundation

In preceding sections we have presented a thorough description and analysis of the existing theoretical foundation of WRS in the versions by Andersen [Andersen, 2011] and Mihăilă [Mihăilă, 2011].

The previous versions of WRS revolve around the concept of trust between WRS users, where trust is described through the trust model and the trust of each trustor is located in the user's personal Web of Trust. However, WRS has never operated with explicit trust where one specific trustor states trust in another specific trustee. Instead the trust model was based on a form of implicit trust which was inferred from the similarity of the ratings given by trustor and trustee.

At the same time, through the evolution of WRS in the versions by Mihăilă [Mihăilă, 2011] and Andersen [Andersen, 2011] the system changed to store all rating data and trust data on a central WRS server, instead of locally at each WRS user's computer and on Wikipedia's user pages. We find that this change in the foundation of WRS is so significant that it requires the core trust concepts of WRS to be reconsidered.

We suggest a solution which moves away from the trust model of previous versions of WRS. Instead rating predictions should be based purely on the correlation of ratings between different users.

This solution will give a number of advantages. Among these, an important one is that that WRS will be able to base predictions on a much larger data set. The Web of Trust based idea faced the following limitations:

1. The Web of Trust of a trustor A was expanded every time a rating was given to an article X. This happened based on everyone else who had already rated the page. However, future ratings to X would not affect the Web of Trust of A, unless he or she returned to rate X again at a later stage.
2. The Web of Trust was only expanded with the trustees whom A had direct interactions with, and therefore does not have support for trust or similarity propagation.

As mentioned, the trust model is an indirect way of calculating the correlation of ratings between two users, where trust closely follows the similarity in the ratings given by trustor and trustee. However, well-documented tools for calculating correlation or similarity can be utilized instead.

Examples of such tools are Pearson product-moment correlation coefficient (in the following referred to as Pearson correlation) and Euclidean Distance or Squared Euclidian Distance.

When using statistical correlation instead of the WRS trust model it is possible to model more fine grained levels of correlation. For example, in the current trust model it makes no difference if a trustee has rated 7 or 1 for an article A, if the trustor has rated 9 for A. Both the ratings 1 and 7 are more than a distance of 1 from the rating of the trustor and the accumulated *x* value is affected in the same negative way. However, the intuition is that the "size of the difference" should somehow be reflected in the calculated trust value, and this is not the case with the current WRS trust model.

Another example is if trustee has given ratings 3, 5, 7 for articles A, B, C while trustor has given ratings 5, 7, 9 for the same three articles. With the WRS trust model the trustee will get a significant negative trust

value even though it is intuitively clear that there is some correlation between trustor and trustee: The trustee could actually be agreeing with the trustor on the quality of the rated articles, but may have different expectations about the general quality of articles and therefore rate these specific articles lower.

Many other examples could be thought up, but the essence of the argument is that the trust model is somehow binary for each interaction: The interaction can be only either positive or negative, and this significantly reduces the quality of the model when compared to a model based on statistical correlation.

3.1.3.1 Other techniques of finding similarity

When calculating similarity in connection with collaborative filtering many scientific articles mention Pearson correlation as the preferred measure for finding similarity between raters ([Breese et al., 1998], [Sarwar et al., 2001], [Guo et al., 2012] and others).

Pearson correlation calculates the linear dependence between two variables X and Y and gives a result in the range [-1, 1]. A value of 1 implies that a linear equation describes the relationship between X and Y perfectly, where all points are on a line for which Y increases as X increases. A value of -1 implies that all points are on a line for which Y decreases as X increases. A value of 0 implies that there is no linear correlation between the variables.

The following formula calculates the Pearson correlation coefficient:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

Here *X* is the set of points from one set; *Y* is the set of points from the other set and *n* is the size of each of the sets. $\bar{X}$ and $\bar{Y}$ are the average values of the sets X and Y respectively.

A feature of Pearson correlation is that it is invariant to scaling and will account for a user's tendency to rate either high or low. If two users rated the same article and the first rated them as the set [2, 3, 4, 3, 1] and the other rated them [4, 6, 8, 6, 2], then by the current trust model in WRS the two users would be considered dissimilar and the trust value would be negative.

Pearson correlation will show that there is in fact a strong correlation between the two users in that multiplying the ratings of the first user by 2 gives exactly the ratings of the second. This shows the linear dependence, since the transformation of the ratings of the first to the second user can be modeled with the following equation.

$$r_{second} = 2 \cdot r_{first}$$

It is also invariant to adding a constant, such that the sets [2, 3, 4, 3, 1] and [5, 6, 7, 6, 4] will also show a linear dependence but this time because adding a constant to every rating will transform the ratings using the following equation:

$$r_{second} = r_{first} + 3$$

However, after careful analysis and experiments with various examples of WRS ratings we find that Pearson correlation is not an optimal measure for similarity within WRS. The primary reason for this is that Pearson correlation does not work well with small sample sizes, which follows from this basic analysis:

1. Pearson correlation cannot be calculated for sets of size 1: By examining the formula it is apparent that attempting to do this will result in division by zero.
2. For any set of size 2 the coefficient will be either exactly 1.0 or -1.0 or result in division by zero

In order to quantify exactly when the Pearson correlation provides useful values we have generated all mutually different sorted sets of integer values in the range [1,9], for sets of increasing sizes. Because the sets are sorted we will never get a correlation coefficient of -1.0. The table is shown below:

Set size	Different permutations (A)	NaN/division by zero (B)	Correlation coefficient of 1.0 (C)	Equal sets (D)	Useful values % (Worst case)
1	45	45	0	9	0%
2	825	285	540	45	5,4%
3	9075	945	1338	165	76,7%
4	70785	2529	2649	495	93,4%
5	429429	5961	4974	1287	97,8%
6	2147145	12825	8669	3003	99,1%
7	9202050	25695	14995	6435	99,6%

Table 2 - Pearson correlation coefficient for data sets of different sizes

The data can be found in the attached appendix section *Pearson correlation and Squared Euclidian distance coefficient for small data sets*.

The percentage of possibly useful coefficient values is calculated based on the absolute number of useful values of each set size:

Number of useful values = A - B - (C - D)

The reason behind this calculation is that we say that the result is not useful when the coefficient cannot be calculated, and if the correlation is 1.0 it is also not useful unless the sets are exactly equal: In this case a Pearson correlation of value 1.0 is considered useful.

It is important to note that with this definition of usefulness we do not actually utilize the fact that Pearson correlation is invariant to scaling and adding constants, and this is why we denote it as worst case in the table. For a higher number of common values, it may actually be possible to use the coefficient, even if it is 1.0 and the ratings are mutually different.

The invariance of Pearson correlation allows for finding correlations that other similarity metrics might not have found, but the problem is that this feature also has its downsides when dealing with the small number of article ratings. In order to actually utilize a suggested rating if there is a correlation when the ratings are different, we have to do some sort of translation from the trustee's to the trustor's ratings.

An example is if the trustee rates three articles [3, 1, 2] and the trustor has rated the same articles higher [6, 2, 4], and thus there is a correlation of 1.0 between them. Since the trustor seems to rate higher than the trustee, simply using the trustee's ratings as is would result in a predicted rating that would be lower than it should be. Instead the trustee's ratings have to be translated to match the rating tendency of the trustor; in this case multiplying by a factor 2 seems to be in order.

By examining the results, one will realize that the correlation coefficient value for sets of size 1 is not at all useful. For the sets of size 2, 5.4% of the permutations have a correlation of 1.0 because the sets are actually equal, and the rest are either not computable or returns a 1.0.

The problem with sets of size 2 is that no matter what the values are, there are only two of them and as such it is always possible to draw a straight line through the two ratings, resulting in a Pearson correlation of 1.0. This effectively makes Pearson correlation useless for sets of sizes less than 3

For sets of size 3 Pearson correlation shows increasing usefulness, but there are still cases where Pearson correlation shows a full linear dependence between the sets even if an observer might not consider the sets of ratings similar. This is because the sample of ratings that the similarity is based upon, may be too small to actually determine whether a trustee genuinely shows a tendency to rate lower or higher than the trustor or if it is simply a matter of coincidence.

We will illustrate this by an example. Say that trustor and trustee each has rated six articles, and they have rated three in common as illustrated in the table below. The letters in the first row denote different articles, and trustor and trustee has both rated articles A, B and C. Trustor has also rated D, E and F, while trustee instead has rated G, H and I.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
Trustor	7	8	9	9	8	6	-	-	-
Trustee	3	4	5	-	-	-	4	5	5

Table 3 - Example of ratings given by a trustor and a trustee on the articles denoted by a letter.

When calculating the Pearson correlation for the shared set of ratings for A, B and C, the coefficient becomes 1.0, which means perfect correlation. Because the trustee actually tends to rate much lower than the trustor, the Pearson correlation appears to have correctly found a linear dependence between the sets.

However, if the same trustee rates 8, 8, 8 for the articles G, H, I, the coefficient would still return 1.0 since the correlation computation is only based on the shared set of articles. But in this situation it would not be fair to say that the trustee generally rates lower - this is only the case for the shared set and is not a general tendency for the trustee. This means Pearson correlation alone would not be enough to determine whether or not the trustee and the trustor actually agree.

Another issue with Pearson correlation is if a user tends to always give the same ratings, such as only rating articles that he or she finds really great. One could imagine a user who only issues ratings of value 9. The problem is that Pearson correlation cannot compute a coefficient if all values of the set are equal. For a user with a rating set of [9, 9, 9, 9, 9] Pearson correlation will never be able to compute a single similarity score with any other user, regardless of their ratings, because the calculation will always result in a division by zero.

Within other data sets where the number of ratings in common is likely to be large i.e. where the number of items to be rated is relatively low, compared to the number of raters and ratings given, the Pearson correlation coefficient is likely to give useful results. Several of the sources of this thesis analyze collaborative filtering based one movie rating systems such as FilmTrust [Guo et al., 2012], [M. Jamali, 2010] and Flixster [Guo et al., 2012].

There is a tendency with movies that when they are released, some are extremely hyped and as a result will be seen by a high number of people within a short time span. These movies are more likely to receive ratings on a rating site in the time after the premiere, creating a cluster of users centered on the movie. By rating one of these movies, the user is likely to interact with a significant number of other users that has also rated the same movie and thus expand the network of trustees. If the same user rates more of these highly popular movies, the chances of interacting multiple times with the same trustee increases over time and thus improves the accuracy of using metrics such as Pearson correlation, which works best with a larger number of commonly rated items.

On Wikipedia the premises are different. Wikipedia does not in the same sense contain extremely popular articles that are hyped in order to attract people. Instead articles are put on Wikipedia silently and clusters may not form around articles as they do with movies. There will still be articles that are extremely popular, but it is not immediately obvious which ones those are. This difference is only natural given that movies and encyclopedia articles are two different mediums. The movies that a person rates as good or bad tells a lot about the preferences of that person, because often, if a person likes one movie of a specific genre he or she is likely to like other moves of the same genre. A person who liked the Lord of the Rings saga might be more likely to also like The Hobbit than someone who hated the former and likewise someone who likes Jackie Chan might also like movies with Bruce Lee. This will tend to attract the same people to a smaller subset of the movies, something that is not trivially true for Wikipedia.

Wikipedia is first and foremost an encyclopedia. We believe that as such the use case of the encyclopedia is as follows: If a user has a question one wants answered about a particular subject, then the user finds the corresponding article about the subject. But because Lord of the Rings fans also tends to like The Hobbit does not mean that the person who wants to find the natural habitat of the great white shark wants to know anything at all about the bull shark, even though these two can be considered somewhat similar. The user was merely trying to answer a question that one of the articles is much more likely to answer and there is nothing that indicates that the user will rate the bull shark article.

An opposing view could be that if e.g. a computer scientist reads up on background material about some subject, this person is likely to read and maybe rate several articles centered on this subject. Other computer scientists could be in a similar situation, and clusters of interest could form around articles which are likely to be read when researching a specific subject. However, we do not believe that this is how the average Wikipedia user is likely to use the encyclopedia.

The difference described in the preceding sections between movies and articles means that articles on Wikipedia might not generate the same cluster of users around similar articles as is the case with movies. This means that there is no intuitive evidence that rating two related articles will have a significantly higher probability of interacting with the same trustee than rating two unrelated articles.

The consequence is that often when the correlation/similarity between two users is being calculated it will be based on a small dataset (possibly 3 or less common ratings) which means that Pearson correlations will not be an appropriate tool to calculate the similarity within WRS.

This is backed by the fact that our primary concern is the cold-start problem which further reduces the likelihood of interacting with a particular user on multiple articles.

Instead we propose similarity based on Squared Euclidean Distance (SED) measure, which gives better results for small datasets:

$$d^2(p,q) = (p_1 - p_1)^2 + (p_2 - p_2)^2 + \cdots + (p_i - p_i)^2 + \cdots + (p_n - p_n)^2$$

In order to normalize the value within a range [0; 1] where higher value means higher similarity we modify the result as follows:

$$si = \frac{1}{1 + d^2(p,q)}$$

The 1 is added in the denominator in order to avoid division by zero. Using the formula and especially the part where each distance is squared we make sure that greater distances between ratings count progressively higher in a negative direction when the similarity index is calculated.

The following table shows a few handpicked sets of possible ratings by a trustor and a trustee along with the calculated Pearson correlation coefficient and the SED similarity index. The table illustrates the point that Pearson correlation does not work well for small set sizes. Of the 9 different sets 4 cannot be calculated, 4 gives 1.0 even though the sets are dissimilar, and only the one with a set size of 4 gives a coefficient which appears to be useful.

On the other hand the SED similarity index can always be calculated, and the relative size of the index grows and shrinks according to the absolute difference between each rating in each set with no surprising exceptions.

Ratings by trustor	Ratings by trustee	Pearson correlation coefficient	SED similarity index
[1]	[1]	NaN	1.0
[1]	[5]	NaN	0.059
[1,2]	[1,3]	1.0	0.5
[1,8]	[4,5]	1.0	0.053
[4,5]	[4,4]	NaN	0.5
[1,1,2]	[1,4,4]	1.0	0.111
[2,2,2]	[6,6,7]	NaN	0.017
[1,2,3]	[6,7,8]	1.0	0.013
[3, 5, 7, 2]	[4, 4, 6, 1]	0.893	0.2

Table 4 - Examples of Pearson correlation coefficient and SED similarity index for various rating sets

This similarity index can be used directly when calculating the predicted rating by using a weighted average of ratings where the weight of each rating is the similarity index between the trustor and the trustee who gave the rating.

However, using only the similarity index as weight presents one challenge: A single interaction between trustor and trustee where the ratings were equal will weigh the same as 10 interactions where all ratings were equal.

Intuitively one would think that the single positive interaction may be sheer luck, while 10 interactions show that trustor and trustee consistently agree on articles and therefore the suggested rating from the trustee with the high number of ratings should carry more weight.

In order to achieve this behavior we reduce the weight of the similarity index if the number of interactions is less than 10. This is done by multiplying with a factor that depends on the number of interactions that the similarity index is based on:

$$weight = \begin{cases} si \cdot \frac{n}{10} & n < 10 \\ si & n \geq 10 \end{cases}$$

Where *n* is the number of interactions. Using this technique, we make sure that the weight which is based on 10 equal ratings will count 10 times as much as the weight which is based on a single agreement. We pick the number 10 because we believe that trustor and trustee is unlikely to have more interactions than this. However, the number is a variable that can be easily changed if, at a later stage, experimental data show that the number can be optimized.

With this measure for the weight we are not strictly required to filter out the contributions from trustees that the trustor does not agree with, because when the similarity is low the weight will be extremely low and therefore is very unlikely to affect the final rating.

The following example shows a prediction being calculated for article X for trustor D based on ratings from trustee A, B and C. In the example A, B and C has rated the same three articles as D but have also rated the article X which they have all given a different rating.

A: [1, 2, 5, 6]
B: [3, 5, 7, 2]
C: [8, 9, 1, 9]
D: [3, 4, 8]

As evident by observing the numbers, B appears to rate quite similar to D. C rates very differently from D and A is somewhere in the middle.

Using our Squared Euclidean Distance based weight we will calculate the predicted rating. First we will calculate the weight of the contribution from A:

$$d^2(A,D) = (1-3)^2 + (2-4)^2 + (5-8)^2 = 25$$

$$w_{A,D} = \frac{1}{1+25} \cdot \frac{3}{10} = \frac{3}{260}$$

Then we calculate the weight of the contribution from B:

$$d^2(B,D) = (3-3)^2 + (5-4)^2 + (7-8)^2 = 2$$

$$w_{B,D} = \frac{1}{1+2} \cdot \frac{3}{10} = \frac{3}{30}$$

We calculate the weight of the contribution from C:

$$d^2(C,D) = (8-3)^2 + (9-4)^2 + (8-1)^2 = 99$$

$$w_{C,D} = \frac{1}{1+99} \cdot \frac{3}{10} = \frac{3}{1000}$$

The result is that the rating of item X from B should count *8.7* times more than the rating from A:

$$\frac{\frac{3}{30}}{\frac{3}{260}} \approx 8.7$$

And the rating from B should count in excess of 33 times more than the rating from C:

$$\frac{\frac{3}{30}}{\frac{3}{1000}} \approx 33.3$$

This also illustrates the point that the contribution from trustees that the trustor is dissimilar to only has insignificant impact on the calculated rating.

The calculated rating then becomes:

$$\text{rating} = \frac{\frac{3}{260} \cdot 6 + \frac{3}{30} \cdot 2 + \frac{3}{1000} \cdot 9}{\frac{3}{260} + \frac{3}{30} + \frac{3}{1000}} \approx 3$$

The calculated rating of 3 makes sense because it is very close to the rating from B which D seems to agree the most with, but it is still affected by the ratings from A and C, and this is why the calculated rating is not exactly the same as the rating given by B.

3.1.3.2 Combining different similarity measures

In the section above we have discussed Pearson correlation and SED as measures for similarity within WRS. We have shown that even though Pearson correlation appears to be the preferred measure within scientific literature concerned with collaborative filtering, it may not work well within WRS where the number of common ratings is likely to be low.

However, one could argue that it would make sense to combine the two measures of similarity i.e. use Pearson correlation for when the number of interactions is sufficiently high. This would provide the following two benefits:

1. Pearson correlation coefficient is very well documented compared to SED
2. Pearson correlation is able to display a correlation in the situation where trustee may be generally more harsh or mild when rating articles, compared to the trustor. For example the Pearson correlation in a situation where trustor A has rated [1, 3, 5, 6] and trustee B has rated [2, 4, 6, 7] will be 1.0 i.e. full linear correlation. This may be seen as a benefit because the two raters appear to agree on the relative quality of the four articles, but A seems to generally rate lower. However, since WRS works on absolute ratings this type of perfect correlation coefficient would require the system to translate the rating from the scale of the trustee to the scale of the trustor and it may not be trivial to determine when a translation is appropriate because the correlation coefficient does not reveal this information.

The problem with such combination is that the coefficient calculated from the two measures cannot be interchanged and therefore cannot be used side by side. For example, if there are two relevant trustees for article X, A and B where with A, the trustor has three interactions, while with B he or she has four interactions. In order to weigh the rating from A and B when calculating the predicted rating of X, the similarity index would play a role, but if one is calculated using SED and the other using Pearson correlation, the results would be skewed.

This tendency is illustrated in the following graph, where we show how the Pearson correlation coefficient develops when compared to the SED similarity index.

In the graph we focus on a common set size of 6, where one is fixed to [1,2,3,1,2,3] and the other starts as the same set but the rating for the last article is increased in steps of 1.

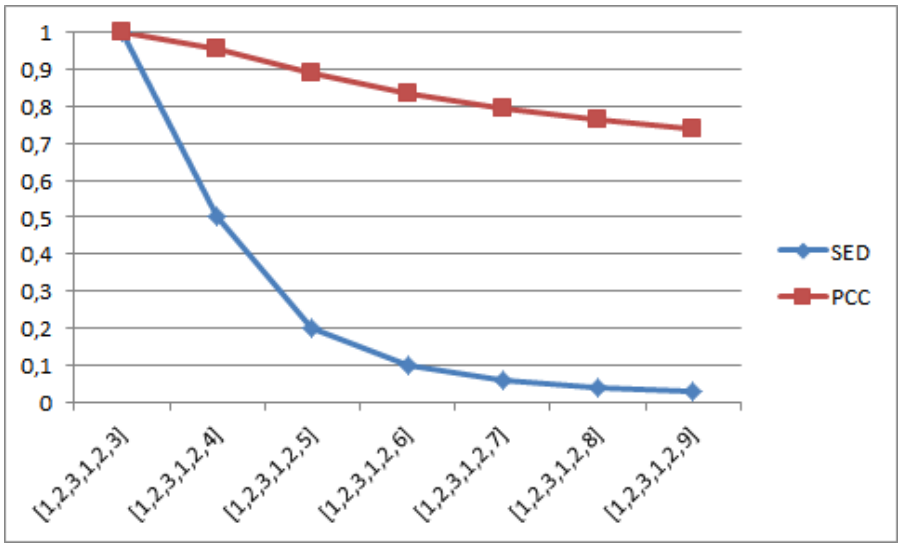

Figure 7 - Development of SED similarity index (SED) and Pearson correlation coefficient (PCC).

It is obvious to see from the illustration that the values for the two measures of similarity is very different when calculated for the same sets, and this is why they cannot be used side by side.

3.1.3.3 Squared Euclidean Distance as an alternative to the WRS trust model

As described previously the WRS trust model consists of the following components:

1. A range between [-1,1] where -1 signifies complete distrust, 1 signifies complete trust while 0 signifies neither trust nor distrust
2. Trust evolution: The trust function is affected by whether or not the trustor appears to be optimistic or cautious
3. Trust dynamics: Older interactions carry less value
4. Categorizations: Interactions may carry more or less value depending on whether or not trustor and trustee agree on the category of the article

In the following section we will analyze how these components of the trust model translate in a similarity index calculated using SED.

3.1.3.4 Similarty range of in SED

The range switches from [-1,1] to [0,1], where values close to 0 means low or very low similarity and values close to 1 means very high similarity. The initial value is 0 signifying no similarity at all.

The WRS trust model only considered trustees with a trust value greater than 0, meaning that trustees with negative trust value were filtered out. When using SED we do not filter out any trustees based on the similarity index. However, gradually as the similarity between trustor and trustee is reduced, the weight of contributions from this trustee becomes lower until a point where the contribution is insignificant.

3.1.3.5 Trust evolution

In the previous versions of WRS the system updates the trust function based on whether or not the WRS user is *satisfied* with the rating that WRS provided. If the user states dissatisfaction with a rating the system should make sure that the user gains trust at a slower rate than he or she otherwise would. This is done based on the answer to the *Yes*/*No* question related to the text: "Was this information useful to you?" as highlighted in the following figure:

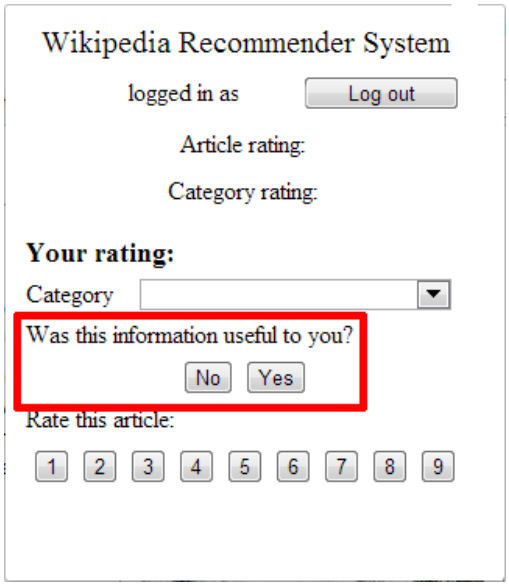

Figure 8 - The main window in the Chrome extension from the previous version of WRS

However, we find that it will be beneficial to remove the trust evolution function and the associated highlighted question for the following reasons:

1. We find that in a system where trust is nothing but similarity between previous ratings, the trust/similarity should only be affected when new ratings are given, thereby possibly changing the level of similarity. In a system where the trust function changes according to whether or not the user states satisfaction with the provided rating, a user may negatively affect the trust function in such a way that it is unlikely that there are any potential trustees that the trustor has a positive trust value in. This means that the system has less data to use to predict ratings, thereby further reducing the likelihood of a correct predicted rating. On the other hand a user may also affect the trust function in a positive direction which means that trustees may be able to gain a trust value of almost 1 with only a couple of interactions. The contributions from a trustee with only 2 interactions may carry almost the same weight (same trust value) as a user with which the trustor has 10 interactions. Both of these consequences of trust evolution may lead to the system giving less predictable ratings, compared to a linear trust or similarity function
2. The highlighted question is mandatory for the current version of WRS. However, it follows from intuition that asking the WRS user as few questions as possible is generally better because this requires less effort from the user and thereby it increases the likelihood that the user will actually provide a rating.

3. The highlighted question may be ambiguous: Does it relate to the rating or the article itself? As developers we know that it relates to the rating, but some WRS users may wrongly assume that it relates to the article and may use it to signify whether or not he or she found the *article* useful, thereby affecting the trust function in an unintended way.

SED does not have embedded support for trust evolution. However, for the reasons stated above, we do not consider it as a drawback that this effect is removed.

3.1.3.6 Trust dynamics

In the previous version of the WRS trust model, older interactions carry less value. The reason behind this way of handling trust dynamics is that if a person A recommends an item X to a person B then there are two dimensions to consider:

1. After some time the item X may have changed significantly since it was recommended, so the recommendation from A does no longer apply to the current X
2. The opinions of user A may have changed so even if A and B agreed on the quality of X some time ago, the recommendations of A are no longer useful to B because the new opinions of A are different from those of B. It could also be that B, the trustor, has changed opinions over time.

By reducing the weight of older interactions, the consequences of changes in these dimensions are reduced.

We believe that the state and quality of articles in Wikipedia is far more volatile than the opinions and sense of quality of the readers of Wikipedia, and therefore in the following analysis we will focus primarily on the first dimension.

Over time a Wikipedia article will evolve and change, and if there are several months between two ratings were given, it may in reality be two completely different articles that were rated, and therefore the trust or similarity should not be affected.

However, the concept of having interactions count gradually less is not ideal when basing similarity on statistical correlation. Instead we will discuss three ways of supporting a variant of trust dynamics within SED i.e. techniques for including only relevant interactions.

1. Age of interaction: Filter out interactions of a certain age
2. Article revision interval: Filter out interactions that have a certain number of revisions between them
3. Text difference: Filter out interactions where the ratings were based on article versions that were significantly different.

3.1.3.7 Age of interaction

We find that time is not an optimal choice for weighing ratings: A Wikipedia article may have been static for the last year and therefore the interaction should intuitively carry full weight even after a year. It may also have completely changed in a single day, thereby meaning that even though two ratings were given to article A only one week apart, the ratings should carry no value at all because in reality it was two different articles that were rated. However, the intuition is that there is usually a correlation between time and changes to the article i.e. it is *more likely* that an article has changed significantly after a year, than that it has after a single day.

3.1.3.8 Article revision interval

As mentioned earlier, Wikipedia has also added support for article ratings. They have chosen another measure than age for differentiating on when ratings should carry less weight namely the revision number.

The idea is that when the article has changed a specific number of times, older ratings should no longer count when calculating the average rating of an article. Wikipedia stops using ratings when they are given to an article that has changed more than 30 revisions since the rating was given[17].

However, this measure is also not perfect, since the changes in a revision may be large or small, and an article may have been completely rewritten in 1 revision, or may have undergone only minor grammatical changes after 30 revisions. But similar to *age of interaction* the intuition is that it is *more likely* that an article has changed significantly after 30 revisions, than it has after a single revision.

Another challenge with this solution is that Wikipedia does not operate with sequential revision numbers for their articles. Instead each revision has a unique ID, and by using the WikiMedia API[18] which is available for Wikipedia, one can query for the number of revisions between two revision IDs, but this information is not inherent in the revision number. This means that it cannot be directly be determined locally if the distance in number of revisions is above a specific threshold.

3.1.3.9 Text difference

We find that the optimal measure would be some kind of textual comparison, such as what can be provided by the UNIX *diff* tool. The MediaWiki API can also provide a *text diff* of two specific article revisions using the `rvdiffto` argument to a query action with prop=revisions[19].

This could be used to compare the article in two revisions where it was rated, and if the article has changed more than a specific threshold, the rating may be ignored. Say for example that the diff operation reveals that there are changes in more than 20% of the sentences of the article, then the

[17] http://www.mediawiki.org/wiki/Article_feedback/FAQ#How_are_the_averages_calculated
[18] http://www.mediawiki.org/wiki/API
[19] See example of changes on the Albert Einstein article using WikiMedia's diff:
http://en.wikipedia.org/wiki/Special:ApiSandbox#action=query&prop=revisions&format=json&rvprop=ids%7Ctimestamp&rvstartid=536912695&rvendid=536911908&rvdiffto=next&titles=Albert_Einstein

system may decide that the article has changed too much and the two ratings should not be regarded as an interaction.

The problem with using this solution within WRS is that determining the level of change is not trivial. It will require some variant of either of the two following solutions:

1. Query the API every time it must be determined if two ratings should be regarded as an interaction.
2. Save the complete article in the WRS database every time a rating is issued, and locally on the WRS server determine the difference level of the articles when required.

However, both of these solutions would require a significant overhead.

Solution 1 would require a tremendous number of queries to the API, and cause a significant slowdown of the system as a whole. Say for example that user A has rated 5 different articles and each of these articles has been rated by 20 other users. Then the system would be required to determine if each of these interactions should actually be considered by querying the API. This would require up to 100 API calls which may each take 100 ms[20] or more. The requests may need to be executed sequentially or only partially in parallel so even for this small example the slowdown could be 10000 ms. Moreover the Wikipedia API may not even allow that many API calls from the same IP.

Solution 2 would require WRS to potentially store every revision of every article of Wikipedia, and this obviously is infeasible too.

It may be possible to design solutions which reduce the overhead but we consider solving this problem beyond the scope of this thesis.

Instead of calculating the basic text difference, it may also be possible to determine how much the text has changed through automated semantic analysis. However, this presents the same challenges as basic text difference and is also out of scope of this thesis.

3.1.3.10 Solution

From the discussion of the three solutions described in the preceding sections it is apparent that the age difference between ratings is the only metric that can be determined locally and in constant time. Therefore we pick this solution, even though we recognize that it is probably not the optimal measure of difference. We pick the number 12 as the highest number of months that may separate two ratings, in order for the ratings to be regarded as an interaction.

Statistical analysis of the rate of change in articles in Wikipedia over time may be used to optimize this number.

[20] This was the average response time when we tested using the API.

3.1.3.11 Categorization

In the previous version of WRS the trust model was based on a primary metric which was the rating, and a secondary metric: The category. As mentioned in *State of the Art,* an interaction where trustor and trustee agree on the rating but disagree on the category would still be regarded as a positive interaction, while if they disagree on both rating and category it would carry even more negative value than if they only disagree on the rating. The choice of weighting within rating and category in the WRS trust model is not based on experimental data.

As mentioned in *Trust dynamics* we want to avoid having to assign different weights to ratings because we base predictions on statistical correlation. Instead we will discuss 3 ways of handling categories in relation to ratings:

1. Ignore categories when calculating similarity
2. Ignore interactions when trustor and trustee assign different categories to the rated article.
3. Ignore categories when calculating similarity if the ratings are similar (trustor and trustee agrees) and ignore the interaction if trustor and trustee disagrees on both rating *and* category.

3.1.3.11.1 Ignore categories

One solution would be to completely ignore categories when calculating similarity i.e. base the similarity calculations purely on the rating. This way, categories would serve only as extra information about the article, which will be displayed to the user. The advantage of this solution is that it will be simple to implement and it will not reduce the coverage because no interactions will be filtered out based on the category. However, this change would reduce the accuracy, and it will essentially remove the purpose of categories within WRS.

3.1.3.11.2 Ignore interactions in different categories

Instead of having the category as a secondary metric we could regard only interactions that happen in the same category. For example if trustor A rates article X 7 in category “Computers” while trustee B rates X 5 in category “Science” this would not count as an interaction.

The idea behind this solution is that two users may actually not disagree on the quality of the article even if they assign different ratings. They may have simply read the article from two different viewpoints and therefore they may judge the articles on different criteria. For example one user may read an article about Albert Einstein as an article about the German and American society in the 20th century and may find it a very good *Society* article. Another user may read the same article and see it as a *Science* article, but may find it that it lacks content and is not very useful because Albert Einstein’s theories are not sufficiently elaborated. These two users do not necessarily disagree on the quality of the article, but they perceive the article differently and this should not affect their similarity in a positive or negative direction.

This solution would reduce coverage because many interactions will be disregarded, thereby leaving an interaction graph less connected.

3.1.3.11.3 A third solution

A third solution would be to ignore the category when trustor and trustee agrees: The idea would be that if trustor and trustee agree that an article is either of high or low quality it really is not essential if they place it in the same category. An article of a low quality may be poor no matter if it is conceived as a *Sport* article or as a *Games* article - and likewise a high quality article may be great, no matter if one focuses on the *Society* or *Science* aspect of the article.

On the other hand, as described in *Ignore interactions in different categories* it is possible that if trustor and trustee disagree on both rating and category, they may in fact not really disagree because they see the same article in a different light. Therefore it could possibly increase accuracy to ignore this type of interactions.

With this hybrid solution we would give the category concept a similar role as with the trust model of the previous version of WRS, even though we somewhat alter the usage.

We believe that this third solution gives the optimal tradeoff between coverage and accuracy, but we cannot know this for sure. In this context we will consider trustor and trustee to agree if the absolute difference in the rating value is 2 or less. It will however slightly increase the average similarity index between any two WRS users because the tendency will be to include similar ratings and disregard dissimilar ratings (If the category also differs).

3.1.3.12 Trust and similarity in WRS

In the preceding section we have discussed a similarity measure which can replace the trust model from the previous versions of WRS and can serve the same purpose of predicting ratings but doing so in a more reliable way because it will base predictions on all ratings ever given through WRS.

A central effect of this shift in foundations is that conceptually we stop operating on a directed trust network and start operating on an undirected similarity network. The following sections will describe how to utilize the new similarity model and the accumulated rating data in new version of WRS.

3.1.3.13 Trust propagation

In *State of the Art* we covered some of the basics of the trust propagation techniques. This section will further explain this concept and how it could apply to WRS.

Trust propagation can be used to calculate user A's trust in user C, through a common interaction with B. The idea is to build a trust network of the community, where the network is modeled as a directed weighted graph as seen below:

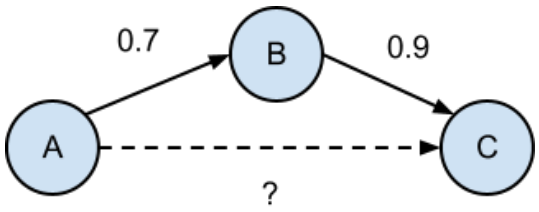

Figure 9 - The trust of A in C is calculated using trust propagation over B

It is worth mentioning that the graph above is directed, since trust is asymmetric at this point. A does not necessarily trust B as much as B trusts A or vice versa, in fact B might not even know A exists at all.

3.1.3.13.1 MoleTrust

As mentioned in *State of the Art*, Andersen elaborated on the MoleTrust and TidalTrust algorithms in order to finally choose MoleTrust.

MoleTrust got its name from an online community website called www.moleskiing.it [Avesani, Massa, Tiella, 2005]. The website revolves around a community of ski mountaineers who explore mountains all over the world. Because the ski mountaineering domain is dangerous with avalanches potentially having fatal consequences, experience means everything but obviously a skier cannot have firsthand experience from every mountain route in the world. This is where moleskiing.it comes into play as a place for the skiers to share their experiences.

The MoleTrust algorithm is a two-step algorithm, which takes four parameters: *trustor*, *network, horizon* and *threshold*. The *trustor* is the user whose Web of Trust we seek to expand, the *network* is the database containing all interactions, the *horizon* is the depth of how far we want to move away from the trustor and finally the *threshold* is the minimum trust value a trustor must have in a trustee, for the trustee to be included in the computation. In the first step all cycles in the network are pruned and the resulting set has the trustees listed by their distances from the trustor. The second step does the actual trust computations based on the cycle-free network. This is done by following all individual paths from the trustor to a trustee who has rated a specific item, and calculating a weighted average, based on the trust along the path.

The computational complexity of MoleTrust was examined by Mohsen Jamali based on the dataset from Epinions website [Jamali, 2010]. The following numbers outline the size of the dataset:

Users	49.290
Items to rate	139.738
Ratings	664.824
Trust statements	487.181

The table below shows the results in the form of the average number of database queries required in order to predict a single rating for increasing horizon values. It also shows the increase in percentage of ratings covered and the absolute number of ratings covered, each time the horizon was increased (actual results were for horizons up to and including 9, but we only show up to and including horizon=4):

Horizon	Average queries	Increase in ratings covered	Increase in coverage
1	95	157.392	23,67%
2	8.169	175.124	26,34%
3	23.149	208.088	31,30%
4	43.042	86.190	12,96%

Table 5 – Complexity for MoleTrust on increasing horizons using the Epinions dataset

These numbers show that retrieving an article rating for a single user requires a large number of database queries for neighboring relations when the horizon increases. When comparing the numbers from Epinion to the numbers from Wikipedia:

Users	~300.000.000[21]
Items to rate	23.900.000[22]

It becomes apparent that with a dataset that is this much larger, the potential number of queries becomes significantly higher, thereby effectively making large horizons infeasible.

3.1.3.13.2 MergeTrust

Following our own research on the MoleTrust algorithm, we encountered a paper on another method for trust propagation, called the Merge method [Guo et al., 2012], and in this thesis referred to as MergeTrust for convenience.

The MergeTrust algorithm works by looking at the trusted neighbors of an active user and merging these ratings in order to treat them as the user's own ratings. The rating for an item is then computed as the weighted average, where the neighbors' ratings are weighed according to how much the active user trusts the neighbor. This gives a merged set of ratings which is said to reflect the preferences of the active user. The merged set is then used to find similarity with other users.

[21] 12,6% of the 2,4 billion internet users according to http://www.internetworldstats.com/stats.htm
[22] # of Wikipedia articles as of November, 2012: http://stats.wikimedia.org/EN/TablesArticlesTotal.htm

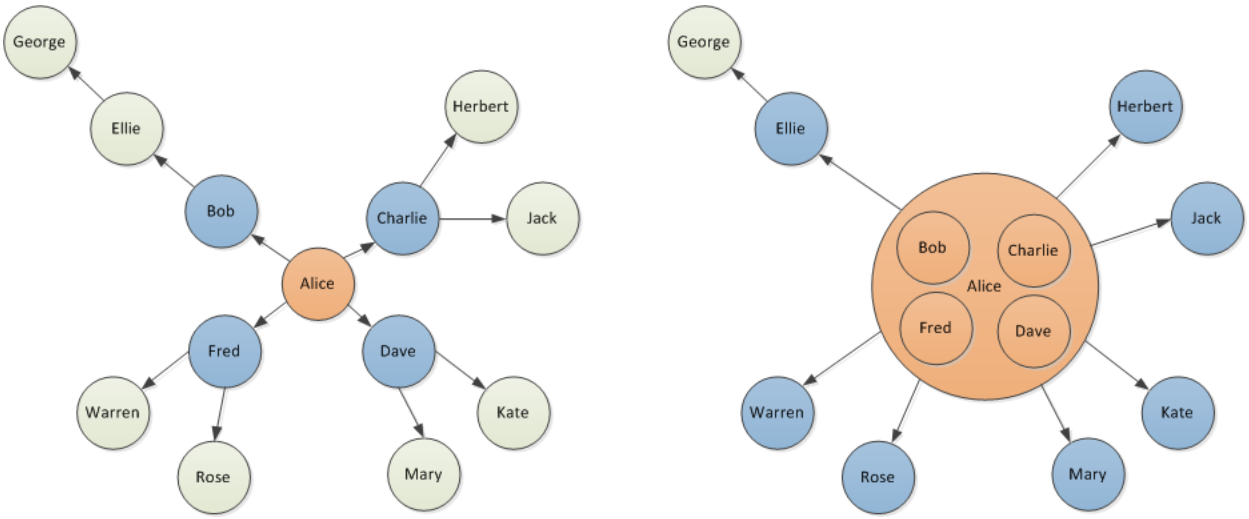

Figure 10 - Comparing coverage of traditional collaborative filtering (left) and MergeTrust (right)

As seen in the left graph above, the active user Alice has had interactions with the users Bob, Charlie, Dave and Fred. With traditional collaborative filtering techniques, these interactions can be used to compute similarity scores which in turn can be used for predicting article ratings of the articles that these four users have rated.

By using MergeTrust, we improve on the coverage in the graph by merging the ratings of Bob, Charlie, Dave and Fred into Alice and treating all ratings given by these users as Alice's own. These are called the trusted neighbors in the MergeTrust Algorithm. If an article has received multiple ratings by users merged into Alice, e.g. Bob and Charlie rated the same article, the final rating will be a weighted average of the two, according to Alice's similarity with Bob and Charlie. After the merge, it is possible to reach all users that the merged users have had interactions with: Ellie, Herbert, Jack, Kate, Mary, Rose and Warren. These people are called the nearest neighbors and article predictions will be available for all articles where either a trusted or nearest neighbor has given a rating.

3.1.3.14 MergeTrust experiments

In the paper by Guo et al. MergeTrust was tested against MoleTrust with different horizons (MoleTrust**x** for **x** = horizon) on three different datasets: FilmTrust, Flixster and Epinions [Guo et al., 2012]. The table below shows the average coverage and the mean absolute error (MAE) of items where a rating could be predicted for cold-start users. A lower MAE is an indication of higher accuracy. Cold start users are defined as having at most 5 ratings. Text in bold is the winner in each respective category.

		Horizon = 1		Horizon = 2	
	Direct collaborative filtering using Pearson correlation	MoleTrust1	MergeTrust	MoleTrust2	MergeTrust2
FilmTrust (Coverage)	39.64%	17.11%	**47.70%**	23.19%	**48.52%**
FilmTrust (MAE - Scale: 0.5-4.0)	0.744	0.853	**0.732**	0.880	**0.753**
Flixster (Coverage)	3.27%	8.11%	**37.07%**	**52.69%**	52.53%
Flixter (MAE - Scale 0.5-4.0)	1.153	1.127	**1.043**	1.005	**1.001**
Epinions (Coverage)	3.22%	6.57%	**15.98%**	22.06%	**22.90%**
Epinions (MAE - Scale 1-5)	1.032	**0.756**	0.943	**0.916**	0.920

Table 6 - Comparison of coverage and accuracy for different trust propagation algorithms

The mean absolute error is a term used to describe how far a predicted value is from the true value. It is an average of the combined absolute differences between all predicted and true values:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i|$$

Where f_i is the predicted value or the forecast and y_i is the actual value. When calculating the MAE, every value y_i is momentarily removed from the dataset such that the remaining data is used to predict f_i . The values are then applied to the equation for MAE.

It is important to consider both the mean absolute error *and* the coverage, because coverage alone is not enough to compare algorithms: One could easily design an algorithm with 100% coverage by having it always return an arbitrary value within the scale of the system for any user. However, this type of algorithm is likely to have a larger mean absolute error than the tested algorithms.

The paper by Guo et al. included experiments for a MergeTrust2 which included another layer of neighbors and is also depicted in the table above, but they found that the increase in coverage is too insignificant to justify the increased computational complexity, but it is included for the sake of comparison. Even compared to MoleTrust2, MergeTrust1 performs quite well and our solution will be based on MergeTrust1, simply referred to as MergeTrust from this point on.

The results show a tendency that MergeTrust generally performs better on MAE and coverage when compared to MoleTrust on the tested datasets on both horizons. They also show the benefits of propagating trust across neighbors, which significantly increases the coverage for cold start users. The reason behind this is called the Friendship Paradox.

3.1.3.15 Friendship Paradox

When a user uses a recommender system for the first time, there will be no data available for rating prediction. Trust propagation seeks to mitigate this by exploiting the concept known as the friendship paradox which is a phenomenon discovered by sociologist Scott L. Feld [Feld, 1991] with data gathered from a study by James Coleman [Coleman, 1961]. The paradox is based on the fact that on average, all your friends seem to have more friends than you do. The following figure shows an exact replica of a subset of the data, which he used to illustrate the paradox:

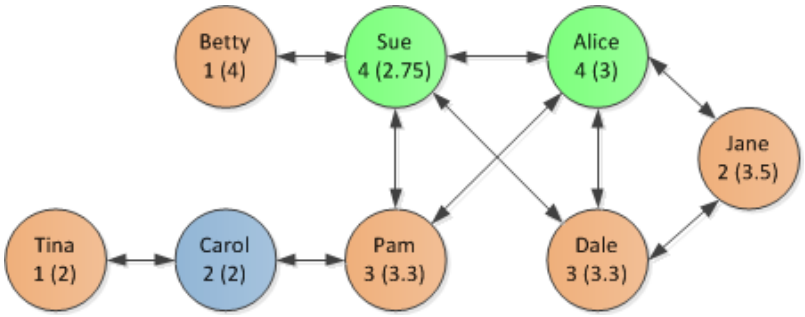

Figure 11 - A small social network showing mutual friendships. The number under each name is the number of friends and the number in parenthesis is the average number of friends of her friends

Looking at the social network above, we see that 5 girls (orange nodes) have fewer friends than the average of their friends, while only 2 have more (green). A single girl has the same as the average of her friends.

The observations by Feld were explained by sampling bias, since the people with a lot of friends are also more likely to be observed than those with only a few.

The above tendency can be transferred to the recommender system domain, where the users that the active user has had interactions with through ratings will on average have given more ratings than the active user. By exploiting this tendency the coverage in a recommender system can be significantly improved.

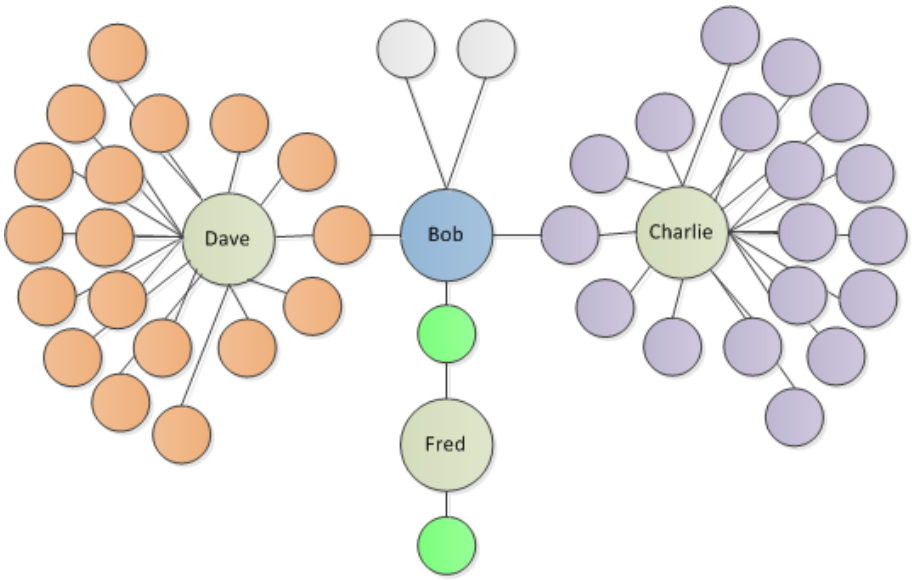

Figure 12 - Unnamed nodes are rated articles by the connected users. Dave and Charlie are observed more than Fred. Bob is the active user who exploits ratings of neighbors through interactions.

Consider the figure above, where the colored nodes (green, orange and purple) are articles that the users Fred, Dave and Charlie rated. We also assume that this little experiment deals with a tiny subset of only 40 articles. Two articles have no other ratings than those by Bob and these will result in no interactions and thus will not boost the coverage through propagation, but the chances of that to happen is only 5%. Clearly it would not help much if a user has had an interaction with Fred either, since he has only rated 2 articles and if we had an interaction with him on one article, trust propagation would only allow rating prediction for a single extra article, being the only other one Fred has rated, alas Fred only boosts the coverage by a single article (2.5%). However, the chance of actually interacting with him through one of the 40 articles is also small (5%).

On the other hand Charlie and Dave are much more visible and the chances of interacting with them is much higher (45%) and upon interactions with either of them the coverage is boosted quite significantly (42.5%). Assuming that the ratings by Charlie and Dave do not overlap, this shows that by rating only 2 articles (one purple and one orange) the coverage is potentially 85%. Obviously this is an extreme scenario, but it once again shows how the chance of interacting with a particular user is closely related to the increase in coverage as a result of said interaction.

3.1.3.16 Trust propagation in WRS

Andersen compares the trust propagation algorithms MoleTrust and TidalTrust and concludes that of these two potential algorithms, MoleTrust is preferable [Andersen, 2011].

We have further analyzed the results from from Guo et al. [Guo et al., 2012] and Jamali [Jamali, 2010] and used them to compare MoleTrust and MergeTrust in order to qualify the choice of algorithm for WRS.

However, it is important to realize that MoleTrust, MergeTrust and other well-documented trust propagation algorithms are not really applicable to WRS. This is because these algorithms work by doing exactly what the name states: *propagating trust*. In these instances trust comes from explicit trust statements of some type. The general idea of any trust propagation algorithm is to let the trust propagate through a number of trustees and then, in the end of a chain of trust propagations, determine similarity. But as mentioned earlier, WRS does not have explicit trust statements, only similarity, which means that the premises of both algorithms are not met.

If we do use a trust propagation algorithm in our setting we will essentially use it for *similarity propagation* instead. So will this be possible? Yes maybe. Even though trust and similarity is two significantly different concepts, scientific studies show that there actually is a strong correlation between trust and similarity.

Cai-Nicolas Zieglera and Jennifer Golbeck [Ziegler, Golbeck, 2005] investigated data from FilmTrust and concluded the following:

> *"We see that as the trust between users increases, the difference in the ratings they assign to movies decreases. This is a significant change, and the correlation between trust and similarity is strong"*

We cannot know for sure if these results will also hold when applied to Wikipedia, so it is important to realize that what we are going to do is an experiment, and we do not know if it is actually going to work i.e. give accurate predictions of ratings. However, we do find it likely.

3.1.3.17 MergeTrust or MoleTrust?

We have described two trust propagation algorithms, namely MoleTrust and MergeTrust. In the following we will list the relevant differences between the two algorithms:

1. The basic MergeTrust will merge only the closest level of trustees, thereby achieving interactions with trustees that are one level away from the trustor. MoleTrust, on the other hand, may go an arbitrary number of levels away from the trustor, thereby possibly increasing coverage even more.
2. As MergeTrust works by merging the trustor together with his or her trustees, the merged entity is likely to have a very high number of ratings - higher than that of most actual WRS users and higher than the number of ratings of the individual trustees which MoleTrust1 will base similarity on.
3. MergeTrust generally appears to have a higher coverage than MoleTrust1 as seen in the results from Guo et al. while the mean absolute error is similar [Guo et al., 2012]. This is a consequence of (2) as the merged entity is likely to have a higher number of common ratings with a level 2 trustee who has rated the relevant item. This means that if there is a minimum number of common ratings for the similarity to be calculated, then this number is more likely to be achieved by the merged entity.

MoleTrust2 has higher coverage for some data sets according to Guo et al., but we do not find this relevant, as it will be too computationally intensive to examine ratings of trustees that are two levels away from the trustor [Guo et al., 2012].

We base this on the observations that follow from the following example: if we assume that the average number of ratings for a Wikipedia article is 100 and each rater has rated an average of 10 articles, then without trust propagation we need to find similarity with a potential of 100 * 10 = 1000 different trustees (Assuming no two encounters with the same user). With one level of trust propagation we will have to calculate similarity with each of these trustees with potentially 1000 other trustees requiring up to 1000^2 similarity calculations given the assumptions. MoleTrust2 will raise the exponent to 3 making the similarity calculations simply infeasible. This is also consistent with the results of the experiments by Jamali illustrated earlier [Jamali, 2010].

We cannot know for sure whether MergeTrust or MoleTrust1 will perform the best within WRS. This is due to the fact that the domains in which the results from Guo et al. apply are different from Wikipedia and we do not know how the performance of the algorithms will be affected when they are used for similarity propagation, instead of trust propagation [Guo et al., 2012]. However, as the results that we do have point towards MergeTrust performing better, this is the algorithm that we will implement in WRS.

3.1.3.18 MergeTrust in WRS

As described, MergeTrust works by merging the trustor with all users that the trustor trusts. MergeTrust will then continue to operate on a special entity which contains all ratings of the trustor and the trustees and this entity will be used to find similarities with other users.

In order to be able to use MergeTrust in our setting we need to consider the following differences between our setting, and the setting for which MergeTrust was designed:

1. MergeTrust uses explicit trust when merging trustor with trustees. After this merge all ratings from trustees and trustor count equally. This means when we use similarity a suitable threshold for the similarity must be applied, in order to only merge with trustees that the trustor has a certain degree of similarity with.
2. MergeTrust does not have the concept of categories, therefore we need to consider the consequences of merging ratings from categories in which trustor and trustee has no interactions.
3. MergeTrust uses Pearson correlation for similarity, while we suggest SED instead because it works better on small data sets.

The following figure shows WRS users as circles, articles as rectangles and the edges between them indicate ratings.

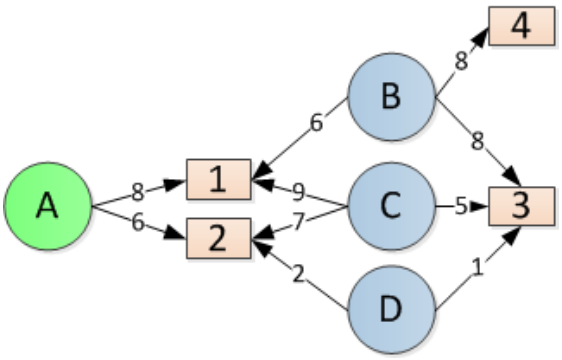

Figure 13 - Interactions between 4 users A, B, C and D

For this explanation of the problem we will consider A as trustor and B, C and D as trustees.

Since we do not operate with explicit trust we need to define some rules for when trustees are included in the merge i.e. they are sufficiently similar to be considered trusted. These rules will be a tradeoff between coverage (The ability to provide ratings at all) and accuracy (The ability to provide useful ratings).

We have no statistical or real world data to base these rules upon, so instead we pick values that appear to be sensible and at a later stage one can experiment with the values, thereby tweaking the system to increase accuracy and/or coverage:

1. There must be at least 1 shared rating between trustor and trustee in any category.
2. The average distance between ratings from trustor and trustee may not exceed 2.

Using these two rules A would not be merged with D because the difference in the given ratings for article 3 violate rule 2:

$$|6 - 2| \geq 2$$

Using rules 1 and 2 one can determine that A will be merged with B and C.

We recognize that these rules - Requiring only 1 shared rating in any category where the rating may be as much as 2 different from the ratings of the trustor - may result in too many inaccurate ratings to be used when calculating similarity. However, since the main purpose of this thesis is to integrate solutions to the cold start problem, we have weighted coverage over accuracy and in later revisions of the program one may choose to change the rules.

In order to mitigate the loss of accuracy which is likely to follow from the rules above we include two mechanisms to limit the effect:

1. We add WikiTrust as a user with which any trustor always have a similarity index of 1.
2. We only use MergeTrust when there are no similar users with which the user has had direct connections and who have rated the article for which a rating is required.

By regarding WikiTrust as a user with which the trustor is completely similar to, we achieve that the weight of the rating from this WikiTrust entity will have a significant impact on the final calculated rating, in the case where the trustor has only a few interactions and SED with weight correction is used as described in *Analysis*.

This can be illustrated by calculating the predicted rating of *article 3* of figure X, by first finding the weight of the ratings by B and C:

$$d^2(A, B) = |8 - 6|^2 = 4$$

$$w_B = \frac{1}{1+4} \cdot \frac{1}{10} = \frac{1}{50}$$

$$d^2(A, C) = |8 - 9|^2 + |6 - 7|^2 = 2$$

$$w_C = \frac{1}{1+2} \cdot \frac{2}{10} = \frac{1}{15}$$

Assuming WikiTrust has calculated a rating of 3 for the article, the predicted rating will then become:

$$\frac{\frac{1}{50} \cdot 8 + \frac{1}{15} \cdot 5 + 1 \cdot 3}{\frac{1}{50} + \frac{1}{15} + 1} \approx 3.2$$

Since WRS only operates on integers and will round the value up or down, the predicted rating is equal to the one from WikiTrust, but it is apparent from the calculation that as a trustor achieves a higher number of similar interactions, the contribution from WikiTrust will become less significant.

A potential problem with MergeTrust based on the rules above is that some accuracy is lost when merging with trustees with varying degrees of similarity. Figure 14 below shows an instance where the problem is evident:

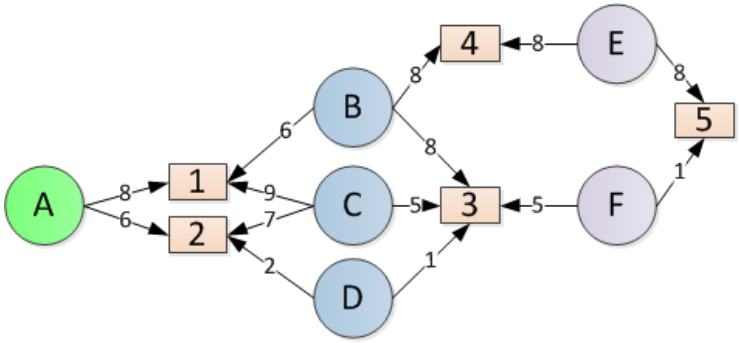

Figure 14 - Interactions between 6 users A, B, C, D, E and F

When the trustor A merges the ratings of the trustees B, C and D, the merged rating for article 3 becomes the weighted prediction that was calculated above, but without the contribution from WikiTrust:

$$\frac{\frac{1}{50}\cdot 8+\frac{1}{15}\cdot 5}{\frac{1}{50}+\frac{1}{15}}\approx 5.7$$

This new rating is rounded up to 6, but still remains closer to the rating of C than that of B, because A had a higher similarity with C. The new merged rating will be the rating used to find the similarity with user F. The problem arises when calculating the final rating prediction for an article such as article 5. For this article, the similarities with E and F are calculated:

$$d^2(ABCD,E) = |8-8|^2 = 0$$

$$w_E = \frac{1}{1}\cdot\frac{1}{10} = \frac{1}{10}$$

$$d^2(ABCD,F) = |6-5|^2 = 1$$

$$w_F = \frac{1}{1+1}\cdot\frac{1}{10} = \frac{1}{20}$$

The above calculations show that since B is the only trustee with an interaction with E and they both gave the article the same rating, they have a similarity of 1.0. The weight of E's rating on article 5 weighs twice as much as that of F, even though the similarity with F is perhaps more trustworthy given that it is

based on a merged rating of several trustees. In fact the rating by the less trusted B is the single reason why the predicted rating on article 3 was pushed upwards from 5 to 6 in the first place, otherwise also giving a similarity with F of 1.0. This demonstrates a scenario where one sacrifices some accuracy in order to achieve better coverage.

3.1.3.19 Categories and MergeTrust

The problem of including categories into similarity propagation using MergeTrust does not have an obvious solution. We believe that in order to achieve optimal accuracy one would have to work on different graphs for each category, which means merging only with trustees that the trustor have agreed with within the specific category.

Otherwise we can have a situation where trustor A has agreed with trustee B in a category such as "Science" and then gets recommendations from B and B's trustees within the category "Kids and Teens" and these may be very different from the ratings that A would have given within that category.

In order to avoid this, one could design a solution that works as follows:

1. Trustor A visits article X
2. Trustee B and trustor A have previously agreed on articles in category K
3. Trustee C and trustee B have previously agreed on article Y in category K. Trustor A have had no interactions with C.
4. Trustee C has rated article X in category K

Using the interaction graph centered on trustor A in category K, trustor A could possibly get an accurate rating for article X through C.

However, such a solution would significantly increase the cold start problem because in order for a trustor to get a rating for an article in some category based on his or her previous interactions, the trustor would have to have rated articles within that specific category. For such a solution the same number of ratings would need to be distributed among 15 different interaction graphs (because of the 15 different categories in the Open Directory Project categorization scheme) instead of just one, which would make it less likely that a rating could be predicted.

The purpose of this thesis is to integrate solutions to the cold start problem, and we find that in the current situation, an increase in accuracy which could be achieved from the solution described above would come at a too high price in terms of a loss of coverage. Therefore, we will use only one interaction graph, and this means that the trustor can get predicted ratings within categories that he or she has not yet rated articles in.

3.1.4 Wikipedia's own rating system

In 2010 Wikipedia launched its own rating system called *Article Feedback Tool* (AFT). This tool has been somewhat controversial, and this section will seek to get an overview of the results of the evaluation of the system and analyze the controversies, in order to relate them to WRS.

The tool presents the box depicted below on Wikipedia articles and allows any visitor to rate the article.

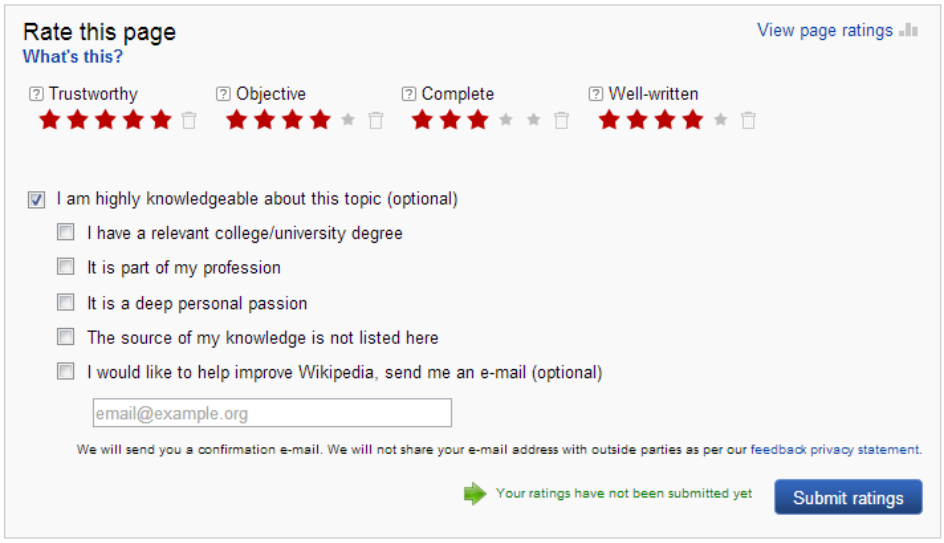

Figure 15 - Wikipedia's Article Feedback Tool

One purpose of the feedback tool is obviously to allow a user to rate the article on the above mentioned criteria, in order to allow future users to quickly reflect on the ratings received by an article. If an article seemed well-written at first, but actually contains many errors that are not easily spotted for someone who is not highly knowledgeable about the topic, the ratings could warn the user that it may not be such a good article after all.

This, however, is not the only purpose of the tool. It is also designed to encourage users to contribute to the content on Wikipedia. It does so after a user has rated by inviting the user to edit the page and thus hopefully increase the quality of the article, so as to make *contributors*, or so called *editors* out of (un)registered users.

AFT is currently in version 4 with version 5 scheduled for release in March 2013. During the first 4 versions a variety of statistics have been collected from the users, these show tendencies that are also encouraging news with regards to WRS.

Up until now AFT has collected over 800.000 ratings with the latest numbers at around 2.000 ratings / day[23]. Since Wikipedia has 300 million unique daily visitors, we see that the potential user base of WRS is extremely large.

As already mentioned there has been quite some debate surrounding AFT and whether it will at all be beneficial to Wikipedia. On the official Wikipedia discussion page of the tool, the user "*Crakkerjakk*" questions the effect of a rating system altogether, with the following argument:

> *"I've seen well written, well-sourced articles rated very low because 10-year-old girls*

[23] Article Feedback v5 Dashboard statistics: http://toolserver.org/~dartar/aft5/

> *don't like a character an actor plays on TV, and I've seen mere stub articles consistently given high ratings because people love and remember their work from 50 years ago."*[24]
>
> *- Crakkerjakk, Wikipedia*

This claims that some people are rating based on what they think about the topic, rather than the quality of the article. This is also backed by the article "Why ratings will ruin Wikipedia" on Gizmodo[25], where the author states that:

> *"Ratings would work great if people weren't, well, people. That is to say, stupid, biased, and lazy."*
>
> *- Mat Honan, Gizmodo*

The article also mentions three reasons why the rating system will fail, although this is strictly according to the author:

1. **Anyone can rate:** No matter how knowledgeable they are on the topic, the vote by a 10-year-old will count exactly as much as a person with a PhD on the topic.

2. **People are biased:** Controversial topics will likely be the target of users' subjectivity. Articles such as "*God"* are likely to contain information that will not please every reader. These articles are supposed to be objective, but nothing prevents the user from placing a bad rating, because he does not like the content, even though it may be a high quality article.

3. **It is too easy to rate:** Wikipedia relies on contributions by the community in order to prevent vandalism. This works by allowing editors to revert changes and in the case of continuous vandalism, by locking an article to various degrees. The difference between edits and ratings is that once a rating is given, it will affect the average article rating for as long as the rating computation considers it, with no way of checking the premises on which the rating was given.

While browsing through the feedback on AFT, the comments appear to be mostly negative, but while this is indeed a bad sign for AFT, it is actually not that discouraging for a system like WRS, since there is a key difference between the two.

The difference is that the ratings of Wikipedia's Article Feedback Tool are based on *global ratings* where the ratings that are displayed to the users are based on a global average of everyone who rated the article and where every vote counts the same. A user therefore has to completely trust every other user and the presented rating will be the same no matter who visits an article.

This is fundamentally different from the idea behind WRS, being a collaborative filtering system. Within WRS a rating is given based primarily on the past ratings of the user, which means that two WRS users who visit the same article may be presented with completely different ratings, since the predicted rating will be a weighted average of the ratings given only by users that the active user has had previous positive interactions with (and WikiTrust). It also means that much of the negative feedback provided by the users of Wikipedia does not apply to WRS.

[24] http://www.mediawiki.org/wiki/Talk:Article_feedback#Please_stop_4324
[25] http://gizmodo.com/5823523/ratings-will-ruin-wikipedia

3.2 Current WRS implementations

The goal of this thesis is to create a working version of WRS which integrates the two previous versions by Andersen [Andersen, 2011] and Mihăilă [Mihăilă, 2011]. The resulting system should be a working version which combines the best ideas of each system along with our own improvements in order to produce an efficient solution to the problem of collaborative filtering and rating prediction in Wikipedia. We have the following concrete goals in terms of the resulting version of WRS:

1. Avoid the usage of the Java based client application along with the Scone proxy. Instead a browser extension should be used.
2. Include the WikiTrust rating in the resulting rating from WRS in the cold start case.
3. Use similarity propagation to faster expand the network from which ratings can be achieved.

In the following sections we will analyze the current solutions of WRS in order to determine how to successfully integrate them in order to achieve the goals mentioned above, and in order to produce a working version of WRS.

3.2.1 Common starting point

The theses by Andersen [Andersen, 2011] and Mihăilă [Mihăilă, 2011] are both based on the same version of WRS by Pilkauskas [Pilkauskas, 2010]. In this version rating data was stored at user pages on Wikipedia. However, in the time between Povilas Pilkauskas delivered his solution and the time where Andersen and Mihăilă started their work on WRS, the Wikipedia API had changed. This required both projects to adapt to the new API in order to actually test the core functionality.

During testing, accounts ended up being blocked as a result of the behavior of WRS. Having the data stored on Wikipedia's user pages, resulted in continuous edits as a consequence, which was incompatible with the general policies set forth by the Wikipedia administrators [Andersen, 2011]. Both Andersen and Mihăilă realized that a fundamental change to WRS was required, and they chose a similar architecture, as described in *State of the Art*. This architecture stored ratings on a central server while a client application on the user's machine retrieved data about the rating through a web service exposed by the central server. However, on the technical level they chose significantly different paths. In the following sections we will look into each level of the architecture in order to pick the best solution for our implementation:

1. Database and Application server
2. Client application
3. Backend design
4. Web service technology and communication protocol

3.2.2 Database and Application server

Andersen [Andersen, 2011] and Mihăilă [Mihăilă, 2011] diverge in the choice of database management system (DBMS). In the following section we will list the reasons which the authors mention as basis for their choices.

The master thesis from Andersen focuses on the Virtuoso Server which is picked as the backend for [Andersen, 2011]. Virtuoso Server is a commercial product which exists in a limited feature open source version.

The commercial Virtuoso server has a hybrid architecture which enables it to offer traditionally distinct server functionality within a single product that covers the following areas:[26]

1. Relational Data Management
2. RDF Data Management
3. XML Data Management
4. Free Text Content Management & Full Text Indexing
5. Document Web Server
6. Linked Data Server
7. Web Application Server
8. Web Services Deployment (SOAP or REST)

However, most of the technologies which are mentioned by Andersen as unique key features of Virtuoso, such as support for SPARQL, SPARUL, RDF, RDFS, OWL and FOAF are left unused [Andersen, 2011]. As part of the thesis some RDF views for the relational data in the database are generated but these views are irrelevant for the WRS system and only serve the purpose of exposing the data through a specific interface which could be used by some potential unknown third-party.

However, two central parts of Virtuoso Server are used:

1. Virtuoso relational database
2. Virtuoso's application server and user management

On the other hand, Mihăilă utilizes the well-known open source RDBMS system MySQL along with the open source application server GlassFish [Mihăilă, 2011].

Virtuoso Server is an extensive server system, and in the general case it will be difficult to directly compare it to the combination of MySQL and GlassFish because of the different feature set of the systems.

However, because only the most basic parts of Virtuoso Server are used, we are able to do some comparison between the two.

[26] http://virtuoso.openlinksw.com/

The advantage of a unified system such as Virtuoso Server is that it may be easier to set up because the same installation will contain all parts needed to host the service - There are no tasks of integrating the servers with drivers or similar like with Glassfish and MySQL.

Another advantage for Virtuoso Server in specific is that it contains support for user profiles per default which means that Andersen could skip the task of creating tables for storage of WRS user data and could instead utilize the existing storage within Virtuoso.

However, there are also significant drawbacks associated with Virtuoso Server:

1. Virtuoso Server is a niche product compared to Glassfish – which is the application server that is bundled with the official Java IDE Netbeans from Oracle – and MySQL which may be the most commonly used DBMS within web hosting in the world. We believe that a very widely used product which thousands of companies depend on is less likely to suddenly become unavailable or unsupported than a niche product and this makes the Glassfish/MySQL combination more stable.
2. Virtuoso Server is a commercial product which has a limited feature open source version while GlassFish and MySQL are true open source products. This means that at some point Virtuoso Server may have its license conditions changed in a way so it becomes unusable for WRS, thereby also making this product an unstable platform.
3. It is beyond the scope of this thesis to discuss or analyze performance of the platforms within the service of WRS. But the fact that MySQL and Glassfish are loosely coupled means that either of the servers can be replaced independently at a later stage, while this is not the case for the unified product Virtuoso Server.
4. The fact that Virtuoso Server stores user profiles in some specific built-in tables makes the WRS data less portable. The thesis by Andersen was delivered with SQL statements to create WRS tables and populate them, but this SQL could not be executed on a clean Virtuoso Server installation because the SQL had foreign key constraints to user data in the built-in tables [Andersen, 2011]. The thesis also had a 100 MB+ database file attached, but this was also unusable because this was compatible only with the Virtuoso Server version of 2011 and could not be used with the current version. On the other hand the version by Mihăilă came with SQL scripts that could be directly executed on a clean MySQL Server installation because WRS was self-contained within the user defined tables [Mihăilă, 2011].

Based on these considerations we decide to continue using the MySQL/GlassFish combination for our implementation of WRS.

3.2.3 Client application

The previous versions of WRS treated Wikipedia as a legacy system where WRS was unable to manipulate the HTML. In order to include the rating information, two different techniques have been utilized:

1. A Java application containing Scone proxy is used to intercept the response from http://wikipedia.org in order to manipulate the content to include a rating box.

2. A browser extension which detects that the browser is showing a Wikipedia page and in this case creating an overlay containing a rating box.

Using the Scone proxy has some clear disadvantages compared to the browser plugin:

1. Even for relatively low numbers of ratings the load time is increased with a factor 400 according to Pilkauskas [Pilkauskas, 2010][27].
2. The proxy is no longer supported or updated. The current version is from February 2009[28] and is based on IBM's even older Web Intermediaries technology.
3. It makes the installation of WRS quite complicated for the average user [Mihăilă, 2011].

There are other disadvantages, but these should be sufficient reasons to find an alternative.

Using a browser extension on the other hand seems to be the optimal solution: Apart from not having the mentioned disadvantages of the Scone proxy it also has other advantages:

1. In 2011 Mozilla revealed that 85% of all Firefox users had addons installed[29]. This means that the users of Firefox are likely to install add-ons/extensions to their browser, if they encounter something that they like. This gives WRS an advantage over a regular application when it needs to be distributed.
2. A browser extension can work asynchronously to create an overlay on the viewed page without delaying the page load. This means that the page can be browsed while the extension works on preparing the rating and still maintaining responsiveness that is similar to browsing without the extension.

The disadvantages of basing WRS on a browser extension are:

1. Currently browser extensions only work on the browser for which they are developed. This means that parallel versions of the WRS client must be maintained for each supported browser.
2. Some browsers may not support extensions at all.
3. Most browsers support only simple scripting languages such as JavaScript for plugins which means that heavy calculations must be avoided.

As stated above it will be nearly impossible to be able to support all browsers by basing WRS on a browser extension. However, by supporting the four most common browsers, Safari, Chrome, Internet Explorer and Firefox, 96.86% of all internet users can be supported[30] which should be an acceptable coverage.

By relying on a central WRS server to do the calculations, the server can expose web services that an extension can take advantage of and simply act as a thin client. For this purpose a scripting language should be quite sufficient.

[27] Reults for a large page, 0.211 sec without WRS, 86 sec with 90 recommendations with WRS (pp. 54)
[28] http://www.scone.de/download.html
[29] http://blog.mozilla.org/addons/2011/06/21/firefox-4-add-on-users/
[30] According to statistics from GlobalStats, January 2013: http://gs.statcounter.com/#browser-ww-monthly-201301-201301-bar

After having established that a browser extension is capable of meeting the requirements of WRS, we will describe the Google Chrome extension developed by Mihăilă [Mihăilă, 2011] and improve on this solution, rather than basing our solution on the Scone proxy which is hereby obsolete as a direct consequence of this decision. Our attention is on the version of the plugin that is currently available in the Google Web Store. When the extension is installed it remains inactive until visiting a page on wikipedia.org, more specifically the English Wikipedia site, upon which the extension triggers and shows a small blue "WRS" icon in the right part of the browser's address bar as shown below:

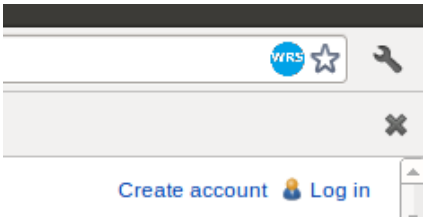

Figure 16 - WRS Chrome extension: Icon showing that WRS is active on the current page

If the icon is shown in the address bar, then it is an indication that WRS is active on the current page. If the icon is pressed, a popup is shown providing the calculated rating for that particular article as well as the article's category, if applicable.

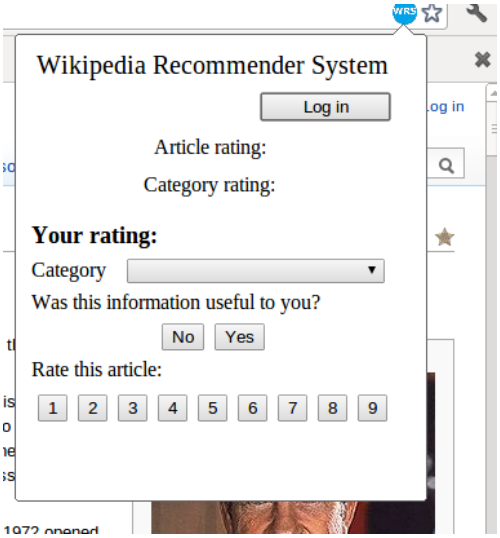

Figure 17 - The popup dialog when the WRS extension icon is pressed

The popup also reveals if the user is logged in and if not, provides a login button bringing up the options page. The popup allows the user to rate the article, assign a category and finally give feedback on whether the information was useful or not. In order to rate an article, the plugin requires the user to be logged in.

3.2.4 Backend design

The different design choices for the client application have great impact on the backend design for the two current versions of WRS.

In the version by Andersen the unmodified WRS client application is preserved, which means the backend is only required to extract the rating data from the backing database and deliver them through a specific interface [Andersen, 2011].

On the other hand, in the version by Mihăilă the client application is replaced with a browser extension which acts as a thin client [Mihăilă, 2011]. In the previous versions WRS consisted only of the WRS client application which is programmed in Java. As Chrome extensions cannot be programmed in Java all the application logic of WRS trust values needed to be removed from the client's computer and moved onto the server.

This change has a very important consequence for WRS as a system: The Web of Trust is no longer stored on the client's machine but on the WRS server.

In the version by Mihăilă the behavioral logic of WRS remains largely unaffected by this change [Mihăilă, 2011]. Instead of storing the Web of Trust as a file on the user's computer it is now stored as a blob of data in the database of the server, and this blob of data is only utilized by WRS.

There are two significant implications of the change:

1. Privacy issue: Now the WRS user must trust the WRS server with his or her private trust data in the form of the Web of Trust
2. WRS can now use trust data from all WRS trustors when predicting ratings for a specific trustor. This opens up for the possibility of more accurate ratings within WRS.

Mihăilă does not reflect about these implications, but we believe that they represent a game changer for WRS and this is part of the reason why we recommend the new theoretical foundation as described in *Theoretical foundation* [Mihăilă, 2011].

3.2.5 Web service technology and Communication protocol

In the thesis by Andersen, the web service is based on SOAP. This choice makes sense because the client is a Java application which can use java classes generated by the Axis framework as abstractions for the XML elements of the communication protocol [Andersen, 2011].

For the Chrome extension by Mihăilă which is based on JavaScript a RESTful web service based on JSON (JavaScript Object Notation) is a far superior choice [Mihăilă, 2011]. XML needs careful manual parsing in JavaScript while JSON can be parsed directly, for instance through the built-in `eval` function of JavaScript.

3.2.6 Implementing a new version of WRS

As described in the preceding sections, we find that overall the version of WRS provided by Mihăilă [Mihăilă, 2011] is superior on all relevant levels when compared to the version by Andersen [Andersen, 2011].

Therefore this version will be used as the foundation for the new version of WRS which we are going to implement, and we will bring over only part of the ideas of propagation from Andersen to the final system [Andersen, 2011].

3.3 Summary

The theoretical foundation of WRS has been thoroughly analyzed, and as a result the existing trust model has been replaced with a model that is based on statistical correlation calculated using Squared Euclidean Distance.

Furthermore it has been analyzed how to implement similarity propagation within WRS by using the existing algorithm MergeTrust. Wikipedia's Article Feedback Tool has been reviewed and it has been found that the concerns that have been raised about the tool relates to problems that are not inherent within WRS.

The two implementations of WRS have been analyzed and it has been determined that we will achieve the optimal technical solution by basing it on the MySQL/Glassfish Application Server combination which provides a RESTful API that can be accessed through a browser extension.

4 Design

This chapter seeks to clarify the overall approach to the system design, which will be the basis of our implementation. The chapter will consist of a schematic representation of the system as well as a description of the various technologies that our implementation relies on.

It will also analyze the problem of implementing similarity propagation within WRS using the MergeTrust algorithm. This analysis will be focused on running time and space usage of the data structures that are required in order to achieve an efficient implementation.

4.1 Overall design

In order for WRS to behave as intended, a number of entities are involved, which provide functionality of different kinds. A schematic overview is as follows:

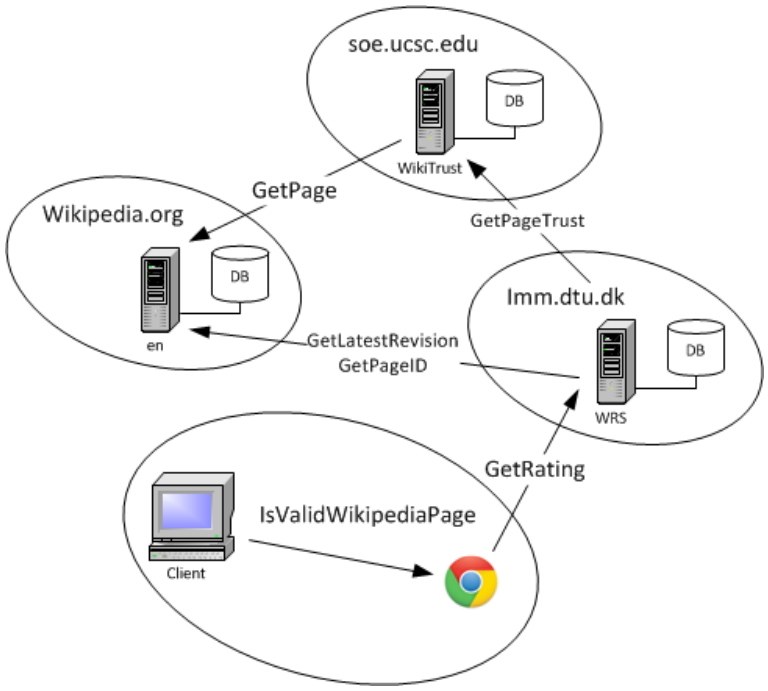

Figure 18 - Overview of the system architecture showing the different entities

The main entities are wikipedia.org, the WikiTrust server, the WRS server and the client application in the form of a Chrome Extension. The client application communicates with a web service provided by the WRS server, and the WRS server communicates with Wikipedia and WikiTrust using a number of different API calls.

4.1.1 Chrome Extension

The Chrome Extension was originally created by Mihăilă for the previous version of WRS and is written in Javascript. The extension provides a non-intrusive interface for rating articles as well as presenting a rating prediction upon opening an article on the English version of Wikipedia. It also makes sure to trigger only on valid Wikipedia articles and otherwise stay hidden and inactive. On valid pages, the extension may utilize the functions provided by the WRS web service.

We have a number of requirements to the extension in order for it to function optimally in correlation with the new version of WRS:

- Update Chrome extension manifest version from 1 to 2, in order to make the extension behave properly for new versions of Chrome. This also entails partially re-structuring the extension in order to adhere to the new specifications of the Chrome API.
- Remove the question “Was this information useful to you? ” in the extension
- Add support for notifying the user that only 1 rating can be placed per user and article
- Update the extension so that it shows meaningful information when neither WikiTrust nor WRS is able to provide a rating prediction or category for a specific article
- Change the extension so that it only requests the rating for an article once. In the current version, the rating is requested once for the notification and then again when a popup is shown.
- Perform basic input validation where relevant e.g. do not allow the creation of a new user without specifying a password.

4.1.2 WRS RESTful web service

The WRS web service is implemented as a RESTful web service, which allows for using the standard HTTP methods (GET, PUT, POST and DELETE) in order to communicate with the server and access shared resources. It is based on the standard Java implementation of RESTful, located in the library package `javax.ws.rs`.

The resources are manipulated according to the HTTP request method used and while REST is not a protocol like for example SOAP but merely an architectural style, there are still best practices which the implementation for this thesis seeks to follow.

For WRS, we build the web service around three resources:

- Users
- Categories
- Ratings

Common to all resources and their associated request is that they should adhere to the following conditions:

- **PUT, DELETE** are idempotent, meaning that multiple identical requests should result in exactly the same state in the system as a single request would have, eg. putting the same element twice will result in the first getting overwritten by the second and not in two identical elements.
- **POST** is not necessarily idempotent, since it is up to the system to decide how to cope with subsequent identical requests and the effects of the request.
- **GET** is the only request of the ones mentioned that should produce no side-effects from being used. It should be used to retrieve data for the caller, but in no way alter the state of the system.

In order to meet these conditions, the GET method should only be used for retrieving data from the database such as for login, getting the list of categories or predicting a rating. In contrast PUT, POST and DELETE are to be used when the required functionality produces side-effects by altering the database, such as creating a new user or rating.

4.1.2.1 Users resource

For a web service running on localhost, the resource for users could be located at localhost:8080/wrs-webapp/users/. The purpose of an interaction with a resource strictly depends on whether it is issued on a collection or on an element of a collection.

For an anonymous user, the resource can be used to create a user in the system, which is necessary for taking advantage of rating prediction. Since creating a user can only be done once with the same username and password, it suffices to implement this using a PUT request:

Resource	PUT
/wrs-webapp/users/	**Create** a new user in the collection. If the username already exists, the state is unaltered.

Table 7 - RESTful web service: Create new user

Once the user is created, it is available as a resource that can be used for authentication, such that only the authenticated user can set and get ratings on his behalf. The user is available at the following resource for the user *username1*:

Resource	GET
/wrs-webapp/users/username1	**Retrieve** the user from the collection.

Table 8 - RESTful web service: Retrieve existing user

4.1.2.2 Ratings resource

In order to set and get ratings for an article, the ratings resource is defined. It allows for getting a rating (the user's own or a prediction) and adding one, given a username and article URL pair. Creating a rating for an article is only possible once, after which that rating is permanent. A subsequent rating by a user on the same article produces no side-effects and thus it also suffices to implement it as a PUT request rather than POST:

Resource	GET	PUT
/wrs-webapp/ratings/username1/articleUrl1	**Retrieve** the article rating for the user	**Create** a rating for the given user and article pair

Table 9 - RESTful web service: Retrieve article rating for an article and username

4.1.2.3 Categories resource

The concept of categories is decoupled from that of users and ratings on articles. The categories do not depend on any other data and have their own resource:

Resource	GET
/wrs-webapp/categories/	**List** the available categories that are predefined in the database

Table 10 - RESTful web service: Retrieve all existing categories

For all requests, the communication between the extension and the web service should conform to a mutual standard of which the chosen communication protocol is JSON.

4.1.3 JSON

JavaScript Object Notation or JSON for short is a text-format used primarily for interchanging data between a client and server over the HTTP protocol. JSON is an alternative to XML, but while XML requires careful parsing of the data if used in JavaScript, JSON can be parsed directly using the built-in JavaScript function `eval()`. JSON also has a smaller memory footprint than XML. JSON formatted data for the user John Doe who has rated two articles could look like the following:

```
{
	"username": "John Doe" ,
	"password": "DoesPassword",
	"ratings": [
		{	"ratingID": 86,
			"articleUrl": "articleUrl2",
			"categoryID": 3,
			"rating": 8,
			"ratingTime": "2013-02-14 14:37:26"
		},
		{	"ratingID": 69,
			"articleUrl": "articleUrl1",
			"categoryID": 3,
			"rating": 6,
			"ratingTime": "2013-02-14 14:33:43"
		}
	]
}
```

Code listing 1 - Example of JSON code

The code snippet above is an example of JSON and does not reflect the exact format used in WRS.

4.1.4 Utilizing MySQL through Java

The database schema of WRS will remain largely unchanged in the new version of WRS. It will still be relatively simple, and the number of tables is going to be reduced from four to three, as the *weboftrust* table from the previous version is obsolete in the new version of WRS.

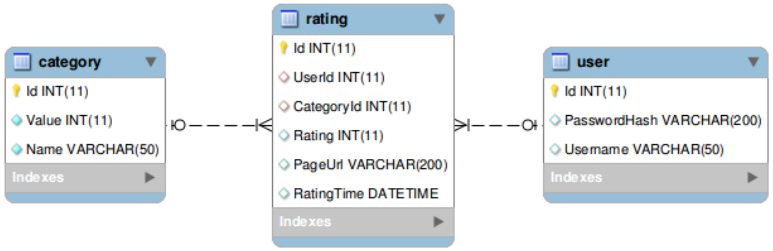

Figure 19 - Database schema for the updated version of WRS

Interactions with the database are done using SQL queries wrapped in a database abstraction layer called JPA (Java Persistence API). JPA allows for object-relational mapping of tuples in a database, such that tuples are directly mapped to Java objects that reflect the state of the database.

4.2 Introducing in-memory databases

It is apparent that the significant number of database queries involved in similarity propagation could very well become the bottleneck in the performance of WRS. The problem stems from the fact that rating prediction involves a chain of queries that will grow polynomially in the size of ratings and users of the system. In order to predict a rating, we need to retrieve all articles that a trustor has rated and for all these articles to find all trustees that have rated the same. The result is the set of interactions that the trustor has had with the trustees. From this set we need to calculate the similarity between the trustor and trustees in order finally compute a weighted average and thus predict a rating. In the breakdown of MoleTrust, we saw how computing a single rating for the Epinions dataset required more than 23.000 database queries, which puts a lot of strain on the database.

We seek to mitigate this by storing the data in memory instead of relying on a traditional RDBMS running on magnetic hard drives, while using an RDBMS as a fallback mechanism for failsafe operation. The use of an in-memory database will greatly improve the performance and furthermore allow for expected constant-time lookup of data that would otherwise involve chains of queries.

Below is a comparison of a number of key features of RAM compared to HDDs and SSDs:

	Random Access Memory	Hard Disk Drive	Solid State Drive
Price[31]	$5.400/TB	$54/TB	$970/TB
Transfer speed[32]	16.000 MB/s	150 MB/s	500 MB/s
Latency[33]	12 ns	7.000.000 ns	40.000 ns

Table 11 - Comparison of prices for various types of data storage

While we see that the price of memory is a factor 100 times more expensive than traditional magnetic drives and a factor ~5.4 more expensive than SSDs, the raw transfer speed and latency of RAM dwarfs that of the others. The result is the number of users that can be served at any given time is vastly superior by storing data in memory. The question that remains is if it is practically possible to store the information required by WRS entirely in RAM. Answering yes to this question is motivated by the fact that Facebook manages to pull it off, being one of the most data intensive websites in existence and still managing to store 75% of the total size of data in memory (excluding images) [Ousterhout et al., 2009]. Worth noting is also that we are not storing the actual contents of articles with pictures, videos and sound files taking up large amounts of space, but simply storing article ratings and users, which further

[31] Exchange rates from xe.com/ucc and prices from edbpriser.dk shows harddrives available at $54/TB, solid-state drives at $970/TB and RAM at $5400/TB on Feb 06, 2013

[32] Performance as per benchmarks carried out during overclocking a PC on hobby basis

[33] Figures from Triple-Channel DDR3-1600 RAM running 9-9-9 timings, Western Digital VelociRaptor 600GB spinning at 10.000 RPM and Kingston HyperX 3K 240GB SSD

motivates that the idea is plausible. We will carry out an analysis of the expected space complexities of this design.

4.2.1 Designing an in-memory database for WRS

In order to increase the performance of WRS, the MySQL database will be supplemented by data structures which allow for the relevant data to be stored in memory instead of the disk. Since memory is volatile and thus cleared when powered off, the database will remain as a failsafe backup of data and kept in sync during operation. This section seeks to uncover the different data structures required for efficient lookup of data.

We assume that the following two entities are stored in the database:

1. Rating
2. User

A rating will have a foreign key to a user. This simple data structure allows for a database growth which is linear in the number of users and ratings. In order to calculate similarity between any two users (trustor and trustee) this database schema would require the following steps:

1. Retrieve all ratings by the trustor - denote these ratings A
2. For each rating in A, use the article identifier B of the rated article to:
3. Retrieve all trustees C who have also rated B (Given certain restrictions such as category and age as described in earlier sections)
4. Retrieve all ratings D given by C
5. For each D calculate similarity between the intersections of ratings A and D

However, as we do not necessarily want to calculate the similarity between a trustor and all potential trustees, we will need to further restrict C to include only the trustees who have rated the article X, which the trustor requires a rating prediction for. Needless to say, extracting the intersections that is used to calculate the similarity will require either a significant number of SQL queries, or excessively complicated queries with several sub queries with restrictions in the form of SQL IN statements or similar.

In order to improve performance, we will design in-memory data structures which will allow for constant or near-constant time lookup instead of chains of database queries potentially in the thousands of queries in order to predict a single rating. The dictionaries provide the following functionality:

Dictionary	Input	Output	Description
A	Username	Interactions	Returns a list of the names of the users that the provided user has had interactions with. Alongside each user is also the list of articles which that particular user has rated in common with Username
B	Article URL	Usernames	Returns a list of all the users who has rated the article
C	Username & Article URL (key)	Rating	Returns the actual article rating given by the user on a specific article
D	Username	Articles	Returns a list of articles that the user has rated

Table 12 - Dictionaries in WRS

The dictionaries will be based on `java.util.HashMap` which provides constant time for the basic operations `put` and `get`[34].

The data structures will be used in the following manner:

Whenever a trustor visits an article X, WRS checks in constant time if the trustor has already rated X by using C and in this case presents the existing rating.

Otherwise the system looks up the trustor in dictionary A resulting in a list of all the trustor's interactions, mapped by trustee to articles which both trustor and the specific trustee has rated.

The system will then filter out the trustees who have rated X by looping through B for X and the list from A. This operation will use linear running time in the size of either the number of trustees that the trustor has interacted with, or the number of raters for X (whichever is largest).

After this step, the *relevant trustees* have been isolated and by continuously using the data structure D the similarity between the trustor and each relevant trustee can be found.

While the general outline of this approach closely resembles that of a design based on database queries, the actual time complexity is significantly improved by using dictionaries with expected constant-time lookups. The most significant performance gain is in getting the interactions between the trustor and all the trustees, which is returned in a single constant-time operation.

If the same was to be done with a database query, the result would be a nested query of several sub queries that each on their own demanded careful optimization. Fortunately most RDBMSs such as MySQL will do this automatically, but experiments should be carried out to ensure that optimization is indeed complete regardless.

4.2.1.1 Running time analysis

We will calculate the running time for the two cases:

1. Predict rating (Without similarity propagation)
2. Predict rating (With similarity propagation using MergeTrust)

[34] http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

As described earlier, MergeTrust will only be utilized if the rating cannot be calculated based only on data from trustees i.e. no trustees have rated the article for which the trustor requires a predicted rating.

Predicting a rating without MergeTrust

Say we want to predict a rating for trustor A on article X. We assume that the number *n* of ratings for article X is greater than the total number of ratings *o* that A has given. The number of *relevant trustees* i.e. trustees who have rated X will be called *m*.

1. Find all A's interactions: O(1)
2. Isolate the trustees who have rated X: O(*n*)
3. Calculate similarity between A and each *m*. Calculating similarity using SED has a running time which is linear to the number of ratings that the similarity calculation is based on. This number cannot be greater than *o*. This step then has a worst case running time of O(*n * o*).
4. Calculate the predicted rating as a weighted average of ratings where the weight is the similarities calculated in 3: O(*m*)

This would give a running time of O(*n * o + m*)

However, as described in *Other techniques of finding similarity* the likelihood of having a high number of interactions between two random WRS users is vanishingly small. Therefore we will assume that calculating the similarity can always be done in constant time. Also, we know that *m* cannot be greater than *n* (The number of relevant trustees cannot be greater than the number of users who have rated X), which means that the running time becomes O(*n*) where *n* is the number of ratings on the article for which a rating is predicted.

We will now analyze the case where MergeTrust is used. We define the following variables:

- The number of ratings on the article for which a rating is predicted: *n*
- The total number of ratings given by A: *r*
- The number of trustees with which A has had interactions: *p*
- The total number of ratings given by all trustees in *p*: *s*
- The number of trustees of A's trustees: *m*

1. Isolate the trustees of A that A is highly similar with: O(*p*) (still assuming that similarity can be calculated in constant time).
2. Merge A together with these trustees. After this step a new entity will exist which appears to have issued every rating from either A or a trustee of A. When more than one of the merged users have rated the same article, a weighted average rating is calculated where the weight is the similarity between the trustor and A: O(*s+r*)
3. Find the relevant trustees of this entity O(*m+n*) - We do not know if *n* or *m* is larger.
4. Calculate the similarity between the entity and each relevant trustee: O(*n*)
5. Calculate the predicted rating as a weighted average of ratings where the weight is the similarities calculated in 4: O(*n*)

The full running time then becomes O(*p*+s+r+*n*+*m*). This can be further simplified as we know that *p* is less than or equal to *s* i.e the number of trustees of A cannot be greater than the number of ratings given by trustees of A since every trustee must have given at least one rating to become a trustee. We will also assume that *r* is less than *s* i.e. the sum of ratings given by all trustees of A is greater than the number of ratings given by A.

The running time then becomes: O(*s*+*n*+*m*) where *s* is the number of ratings given by the trustees of A, *n* is the number of ratings of article X and *m* is the number of trustees of A's trustees. We do not know the relationship between these numbers, so the running time cannot be simplified any further.

4.2.1.2 Space usage of the data structures

In order to achieve the running times with constant time lookup using dictionaries, parts of the data must be stored several times in memory.

The core data structure A is a key/value dictionary. The key is a unique identifier of a user U and there is one key for each user who has rated something. The value is another dictionary which maps from another user R, with which U have had interactions to the list of articles on which U and R have had an interaction.

This way A actually describes the graph of interactions, where each user is a node and each rating connects the user who places the rating with all other users who have rated the same article and can be compared to an adjacency matrix representation of the graph.

However, when comparing the space usage of the data structure A and the data which is stored in the database, there is one key difference. The data structure does not scale linearly in the number of users and ratings, but in the number of nodes and edges of the interaction graph. The number of nodes will not exceed the number of WRS users, but one rating may add as many edges to the interaction graph as the current number of ratings for the specific article and then one.

As described in *Evolution of the theoretical foundation* we do not necessarily regard any two ratings of the same article as an interaction in the interaction graph. Only if the ratings have less than 12 months between them *and* they either agree on the rating *or* disagree only on the rating and not the category.

This way we avoid irrelevant data in the graph representation, thereby improving performance and reducing the memory footprint of the data structure.

The following table shows the worst case space usage of each of the data structures used by the implementation in Big O notation. We denote the number of active users (users who have issued ratings) *n*. The number of ratings given is *m*.

Dictionary	Space complexity	Description
A	O(*n***m*)	One rating can at most connect every active user. This means that *m* ratings can at most add *n***m* edges in the interaction graph.
B	O(*m*)	The dictionary will hold exactly one value for each given rating.
C	O(*m*)	The dictionary will hold exactly one value for each given rating.
D	O(*m*)	The dictionary will hold exactly one value for each given rating.

Table 13 - Theoretical space usage from WRS in-memory data structures

4.2.1.2.1 Memory footprint calculations

In the following we are going to estimate the actual size of these data structures when held in the memory of a server. This is done in order to get a rough idea about whether it is actually feasible to use the proposed data structures and when it may be necessary to use a cluster of servers or other techniques as mentioned in *Introducing in-memory databases*.

The following table shows the space usage of the actual data which is the main content of the dictionaries:

Data content	Space usage	Description
Username	30 byte	Average of 15 symbols in encoded in 16 bit Unicode
Article identifier	80 byte	Average of 40 symbols in encoded in 16 bit Unicode
Rating	4 byte	32 bit integer
Timestamp of rating	8 byte	64 bit long

Table 14 - Actual space usage for content of WRS in-memory data structures

The next table shows the actual calculation of the space usage of each of the data structures. It is important to emphasize that these calculations form very rough estimates and the purpose is solely to give a rough idea of the memory consumption as ratings are added to WRS.

We operate with a fixed 200% overhead each time we add one entry to a dictionary which accounts for the overhead of storing the data in the various Java objects. This number is a rough estimate as we cannot calculate the exact overhead because this depends on the Java version, the Java Virtual Machine on which WRS runs and other unknown parameters.

Dictionary	Space usage of 1 rating	Description/assumptions
A	20 Article identifiers: 1600 byte 20 Username: 600 byte Sum: 2200 byte Overhead: 4400 byte **Result: 6600 byte**	We assume each article is rated by an average of 10 users, meaning that when one rating is given, 10 users are connected. The user who just rated two must be linked to each of the 9 other users who have rated the article. Likewise, each of the 9 users need to be linked back to the rater.
B	Username: 30 byte Article identifier: 80 byte Sum: 110 byte	

	Overhead: 220 byte **Result: 330 byte**
C	Username: 30 byte Article identifier: 80 byte Rating: 4 byte Date: 8 byte Sum: 122 byte Overhead: 244 byte **Result: 366 byte**
D	Username: 30 Article identifier: 80 Sum: 110 byte Overhead: 220 byte **Result: 330 byte**

Table 15 - Estimated space usage for content of WRS in-memory data structures

It is obvious from the calculations above that data structure A will be the first to become a problem in terms of consuming too much memory, as it takes up more than 6 KB for one rating under the given assumptions. The actual memory consumption of adding 1 rating to A varies according to the number of ratings that has already been given to the article in question, while the memory consumption of the other data structures is constant.

To give a rough idea of whether this memory consumption is acceptable for a prototype of WRS, we will try to calculate how much the data structures grow every day given the following assumptions:

1. The data structures grow 8KB with each added rating[35]
2. WRS is deployed to a HP blade server such as *HP ProLiant BL460c Gen8* with 64GB of RAM
3. Of these 64GB of RAM, 60GB can be used by the central WRS data structures
4. WRS gets 1350 ratings per day[36]

60 GB of RAM equals 60 * 1024* 1024 = 62.914.560 KB

This system would be able to run 62.914.560 / (1350 * 8) = 5825 days or close to 16 years.

After this time the system would contain 1350 * 5825 = 7.863.750 ratings which amounts to less than 2 ratings per article if we assume that the number of English Wikipedia articles stays at around 4.1 million. This is less than 10 which was the assumed average number of ratings per article used in the previous calculations meaning that the memory footprint of a rating may indeed be smaller, thereby letting the system run for even longer before lack of memory becomes an issue.

[35] This number comes from summing up the space usage from each of the 4 data structures given the assumptions above.

[36] In the fall of 2011 when after Wikipedia's article feedback tool was released to almost all English articles, it got up to 45.000 ratings per day and about 3% of them came from registered users - The rest was from anonymous users. We assume that we get the same number of ratings from registered user: 45.000 * 0.03 = 1350 ratings/day. Wikipedia rating data from: http://en.wikipedia.org/wiki/Wikipedia:Article_Feedback_Tool

The calculations above show that if our assumptions hold, a single powerful server should be able to hold the proposed data structures in memory, even if it WRS grows rather popular. Should some of the assumptions not hold then the data structures may grow faster, and extra RAM must somehow be made available to the server.

4.2.1.2.2 Maintaining the data structures

The data structures should reside in an object based on the singleton design pattern. When the WRS server application is started up, it builds up the data structures, based on ratings and users in the persistent storage database.

Once this is done, the application will ensure that both the database and the relevant data structures are kept up-to-date. The data structures change only when new ratings are added by a WRS user and the following table shows the running time of maintaining each data structure by adding a new rating. We call the current number of raters of the article which is rated for *n*.

Dictionary	Running time	Description
A	O(*n*)	A new connection must be added for each of the users who have already rated the user.
B	O(1)	Extend a HashMap through one `put` operation which runs in constant time.
C	O(1)	Extend a HashMap through one `put` operation which runs in constant time.
D	O(1)	Extend a HashMap through one `put` operation which runs in constant time.

Table 16 - Theoretical running time of maintaining WRS in-memory data structures

4.2.1.2.3 Race conditions for the data structure

Since the data structures in this singleton will form the core of the system we anticipate heavy concurrent usage of the class and its data structures. The concurrent usage can be divided into two types:

1. Read-only operations: This happens when the data structures are utilized to predict ratings.
2. Read/write operations: Adding new ratings.

We need to somehow handle concurrent usage of the data structures and make sure the system will not fail due to race conditions. One way to avoid race conditions is to use a *Monitor* to achieve mutual exclusion of access to methods of the singleton - This can achieved by using `synchronize` in Java. However, we anticipate that limiting the number of threads which can access the singleton to only 1 will result in a significant bottleneck in the system.

Therefore we suggest another solution which allows concurrent usage and which tolerates small degrees of inconsistency within the data structures. We base the dictionaries on `java.util.concurrent.ConcurrentHashMap` instead of the ordinary `java.util.HashMap`. This way we have a data structure which allows for concurrent modifications but at the price of not guaranteeing

a consistent snapshot at all times: A `get` method may run after a `put` operation and return `null`, if the `put` operation was not complete when the `get` method was executed[37].

In order to work safely under these conditions, the consequences must be anticipated. There are two groups of concurrency issues which may occur:

1. Limitations of `java.util.concurrent.ConcurrentHashMap` (race conditions within each dictionary)
2. Inconsistent data across the four dictionaries: The dictionaries will be updated sequentially so during the process of adding a new rating, the four dictionaries may be mutually inconsistent.

The potential consequences of these groups are:

1. Two users simultaneously rate the same article A. This may not lead to an interaction because when each thread runs through the list of ratings for A, the rating of the other user may not show up.
2. Two users simultaneously rate the same article A: This may lead to two interactions between two users for the same article, instead of only one as intended.
3. The same user simultaneously rates the same article (possibly by sharing his or her credentials with someone else). This may lead to the same user having rated the same article twice.

By further analyzing ways of implementing the addition of ratings, it becomes clear that it is actually possible to implement the functionality in a way so that some of the consequences can be avoided.

Say we update the dictionaries in the order: B, C, D and then A. Then there is no way two threads can be in the state of simultaneously updating A, without the dictionaries B, C and D already containing the ratings, and it is the data from these data structures that is used when updating A. Thereby consequence 1 can be avoided.

Also, if we make sure the ratings are added to the database first and secondly to the data structure, we can make sure that constraints within the business logic or the database stops the operation, if a user attempts to add the same rating twice, and this avoids 3.

The following diagram illustrates the behavior:

[37] http://docs.oracle.com/javase/7/docs/api/

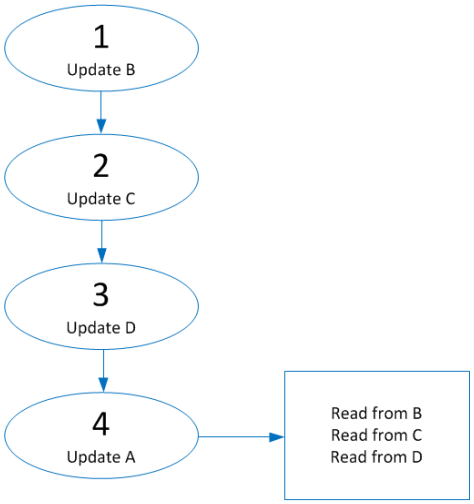

Figure 20 - Illustration of the behavior of updating the dictionaries

The code will be designed to not fail in phase 4 if it encounters the situation where one dictionary appears to include a specific rating, while another dictionary does not. Instead the system will skip this seemingly inconsistent rating and continue.

Say two threads T1 and T2 are adding a rating from two different users U1 and U2 for the same article X. If T1 is in phase 4 and T2 is in phase 3: T1 will find that the states of B, C and D are inconsistent for U2 and will skip adding an interaction for this user. When T2 arrives at phase 4, this thread will add the interaction for both users.

However, if both T1 and T2 are simultaneously in phase 4 they will both see that another user has rated article X and they will both add interactions, resulting in two interactions for the same article. This means that consequence 2 can actually happen.

However, seeing as it is quite unlikely, and that the alternative is adding locks to the operations we find that this potential inconsistency is acceptable, as the only way it will affect the behavior of the system, is by slightly skewing the similarity of the users who have double interactions.

4.3 Security aspects of WRS

A user of WRS entrusts WRS with two pieces of data:

1. User credentials
2. Rating data

The handling of this data in WRS must be considered. For WRS running in a production environment we will assume that WRS is running on a server which has a valid and signed SSL/TLS certificate, and where the RESTful API can only be accessed using the secure HTTPS protocol.

However, for the prototype of the system, this will not be the case. This is due to the following reasons:

- The address of the server where the system has to run is embedded in the certificate. This address is not known at the moment.
- A certificate does not come for free, but must be purchased through a known certificate authority.
- Instead of a certificate signed by a known certificate authority, one can create a self-signed certificate. This will provide the same level of security against *man-in-the-middle* attacks as a signed certificate, but every time a user interacts with a server that uses such a certificate, a dialog is displayed to the user. In this dialog, the user must decide whether or not he or she trusts the self-signed certificate.

When the secure HTTPS protocol is used to transmit data between the Chrome extension client and the WRS server, data cannot be extracted from the requests through a man-in-the-middle attack. The problem with the self-signed certificate is that the browser will very obtrusively warn the user that the identity of the website that issued the certificate cannot be verified and will show the following warning page:

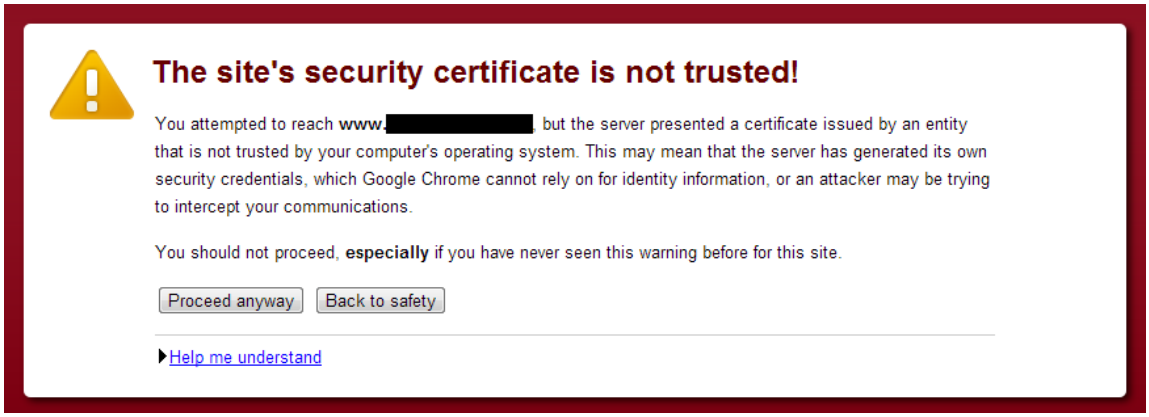

Figure 21 - Warning upon entering a website with a self-signed certificate

The problem with this message is that it may lead the user to believe that the security on the website is insecure, which is far from the truth. In fact the security is functionally the same as a signed certificate, since it is not vulnerable to man-in-the-middle attacks like the HTTP protocol which is vulnerable to this type of attack, but does not show the warning from the figure above. This will perhaps give the user a false indication of an untrustworthy website.

Figure 22 - Self-signed certificate (left) and certificate verified by internet authority (right)

Password handling in WRS

WRS must pay special attention to the handling of passwords as this is the most sensitive bit of information that WRS is entrusted with.

As described earlier, the client in the form of a Chrome extension communicates with the server through a RESTful API. This means that in order to apply to the best practices of RESTful web services, resources must be retrieved through GET requests.

Some of the resources that the client may request requires authentication, and as WRS does not operate with sessions, the credentials are sent as part of each request that requires authentication. The problem is that it is considered very bad practice to include clear text passwords in URLs, and GET requests may not contain a request body.

In order to conform to both of these two design recommendations we have considered two possible options:

1. Use Basic Auth[38] where credentials are contained in clear text in HTTP headers. This form of authentication relies on the security of the protocol.
2. Transmit a hashed version of the password in the request using GET parameters.

WRS uses option 2, where the password is hashed in the client using Stanford Javascript Crypto Library[39] and the hashed password is then transmitted as a request parameter in the GET request.

This option relies partially on the security of this library, and partially on the security of the protocol. Without the security of the protocol, an attacker may intercept the request and may use the hashed password to interact with WRS while posing as the actual user. However, the attacker cannot easily make out what was actually the user's password and can only interact with WRS using the hashed version of the password.

This solution appears to be a reasonable compromise for the initial situation where the system is in a test phase where it may be running as a prototype without using a secure protocol.

In order to not rely solely on the security of the Stanford Library, the hashed password is hashed again in the backend using SHA-256 from the package `java.security`. This way we make sure that even if WRS should be attacked and someone got access to the full WRS database including user credentials, the cleartext cannot be extracted - Even if the hashing function of the Stanford library should end up being insufficiently secure.

[38] http://tools.ietf.org/html/rfc2617
[39] http://crypto.stanford.edu/sjcl/

4.4 Summary

In this chapter we have described the overall structure of the new version of WRS along with the technologies that have been utilized in order to achieve the core functionality. The RESTful API which is used by the Chrome extension has also been defined.

Furthermore we have described which data structures are required in order to support an efficient implementation of MergeTrust within WRS. The running time of core operations such as adding ratings and retrieving rating predictions have been analyzed both with and without similarity propagation. Lastly the space usage and the consequences of concurrent usage of the data structures have been analyzed and we have looked into the security aspects of WRS.

5 Implementation

WRS consists of two separate systems which interact through a common interface:

1. A Chrome Extension acts as the client application which communicates with the server backend through the HTTP protocol.
2. The Java backend implemented as a RESTful web service which exposes methods for the client.

This chapter will describe the overall architecture of the two systems and will also examine central parts of the functionality of the WRS backend.

5.1 Chrome Extension

The overall structure of the Chrome extension is preserved from the previous version of WRS, although some modifications were necessary.

The extension consists of the following files:

- JSON
  - `Manifest.json`
- HTML
  - `popuppage.html`
  - `options.html`
- JavaScript
  - `eventpage.js`
  - `options.js`
  - `popuppage.js`
  - `global.js`

The file `Manifest.json` is the core file of the extension and defines the extension in terms of metadata and is read by Chrome. The structure of this file is defined by the manifest version, and the update of the version number and the changes implied by the new version was one of the central changes to the extension in this version of WRS.

The two HTML files define the two pages of the plugin. `Popuppage.html` describes the popup that opens when the WRS icon in the address bar is clicked, while `options.html` describes the page with options shown when the "Options" button on the popup is clicked.

The JavaScript file `eventpage.js` is the central file for functionality of the plugin and is referred to from `Manifest.json`. This file defines the behavior of the extension in terms of displaying the WRS icon in the address bar of the browser when English Wikipedia articles are visited. It is also responsible for showing the rating notification by using the notification API of Chrome.

The files `options.js` and `popuppage.js` contains the behavioral logic of the two HTML pages of the extension, and these script files also add all event handlers to the elements of the HTML pages. These files were added as part of the new version of WRS.

The last JavaScript file `global.js` contains various functions that are utilized by the other three script files. For instance it contains functions to perform the AJAX requests through which the extension communicates with the RESTful web services of the WRS server.

5.2 RESTful web service

The client communicates with the backend implemented as a RESTful web service, with functionality implemented using the standard Java library package `javax.ws.rs`. This package includes interfaces and annotations for creating RESTful resources. The backend exposes three RESTful resources defined in their own Java classes:

1. CategoriesResource.java
2. RatingsResource.java
3. UsersResource.java

These classes all use functionality from the `javax.ws.rs` package in order for the application server to treat them as web services.

The layout of the entire backend is visualized in the following diagram, which shows the system split into a number of meaningful packages:

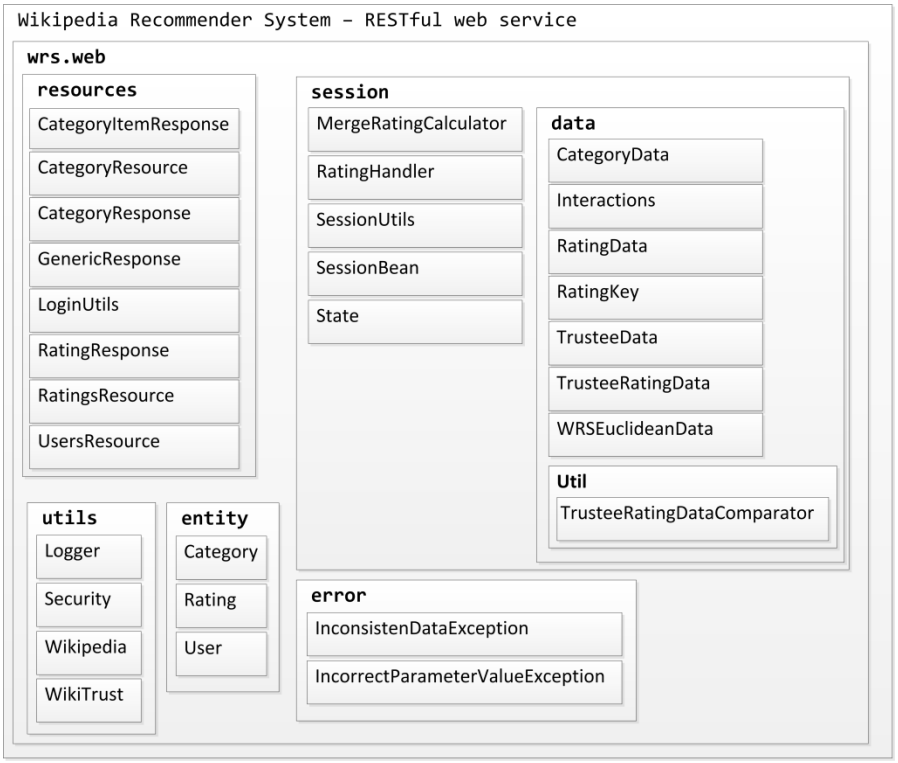

Figure 23 - Package overview of the WRS backend

Common to all the RESTful resources is the communication protocol being used to return data to the client. The chosen format is JSON. All resources return an instance of the base class `GenericResponse.java` or a class that extends from this. This generic response contains a result property of either “success” or “fail” and in the case of a failure also an optional error message. Each of the resources will be described in greater detail, alongside a graphical representation of the different modules of the system that the resources depend on.

5.2.1 CategoriesResource

The Categories resource is the simplest of the three and consists only of a single GET request, namely the getCategories method.

getCategories

The getCategories method is implemented as a GET request with no parameters. It is used only to populate the dropdown menu in the extension with the 15 predefined article categories as defined by Open Directory Project - Dmoz. It is called using the following URL:

```
/wrs-webapp/categories
```

Code listing 2 - Calling the getCategories resource

It calls the database layer through `SessionBean.getAllCategories()`, which has a `namedQuery` annotation used as follows:

```
List<Category> values =
    entityManager.createNamedQuery("Category.findAll").getResultList();
```

Code listing 3 - Interacting with the categories table in the database

The categories are returned to the extension using the class `CategoryResponse` which contains a list of `CategoryItemReponse` objects. Each item contains the ID of the category as well as its name.

The `getCategories` method depends on only a small number of the simple classes of WRS as highlighted with green in the following diagram:

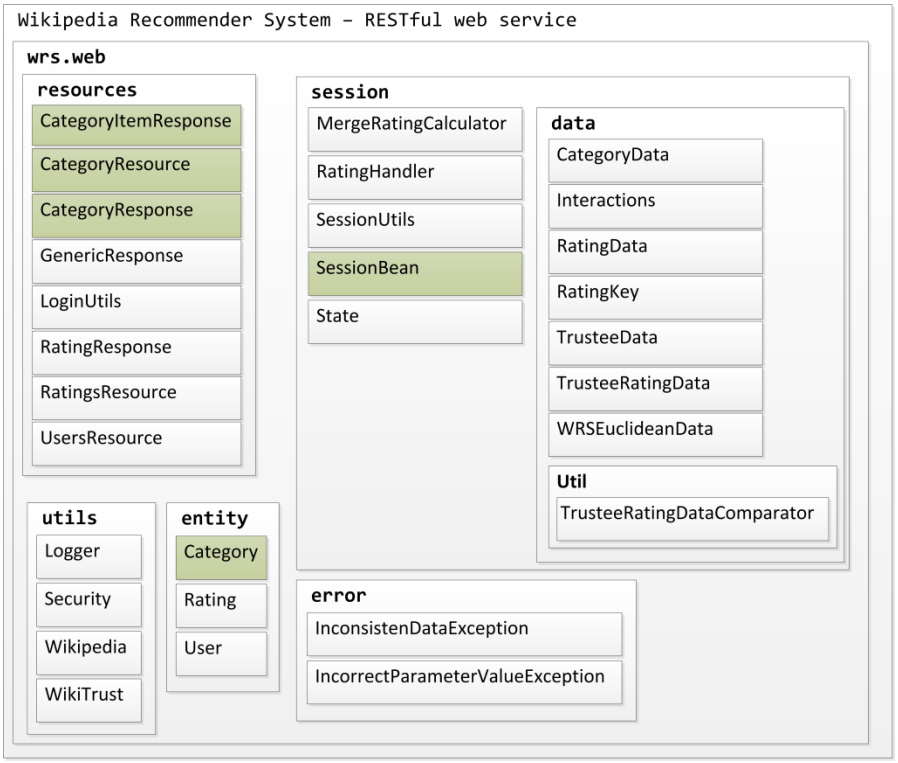

Figure 24 - Classes referenced by the getCategories method

5.2.2 UsersResource

This resource provides means of creating users and verifying credentials in order to authenticate a user. Thus it consists of the following methods:

1. createUser
2. verifyCredentials

5.2.2.1 createUser

The task required by a potential WRS user in order to start using the extension is to create a user in the system. The method is implemented with the username as a path parameter and the password as a query parameter and is called using a HTTP PUT request. Upon filling out the desired username and password, the extension calls the web service through the URL:

```
/wrs-webapp/users/USERNAME?password=PASSWORD
```

Code listing 4 - Calling the createUser resource

The USERNAME and PASSWORD is submitted by the user through the extension and the web service will now try to create the user in the system. The PASSWORD parameter contains a hashed version of the password. It calls the static `createUser` method on the `SessionBean` class and will continue as explained in the example of the *Database Abstraction Layer* section in *Design*.

The submitted password is hashed again with SHA-256 using the `security` package and stored in the database in its hashed form for security reasons.

If the user was successfully created, the web service will simply return `true` wrapped in a `GenericResponse`. An exception is thrown if a user already exists with the submitted username, informing the user of this, once again through the `GenericResponse`.

The `createUser` method depends on the classes that are highlighted with orange in the following diagram:

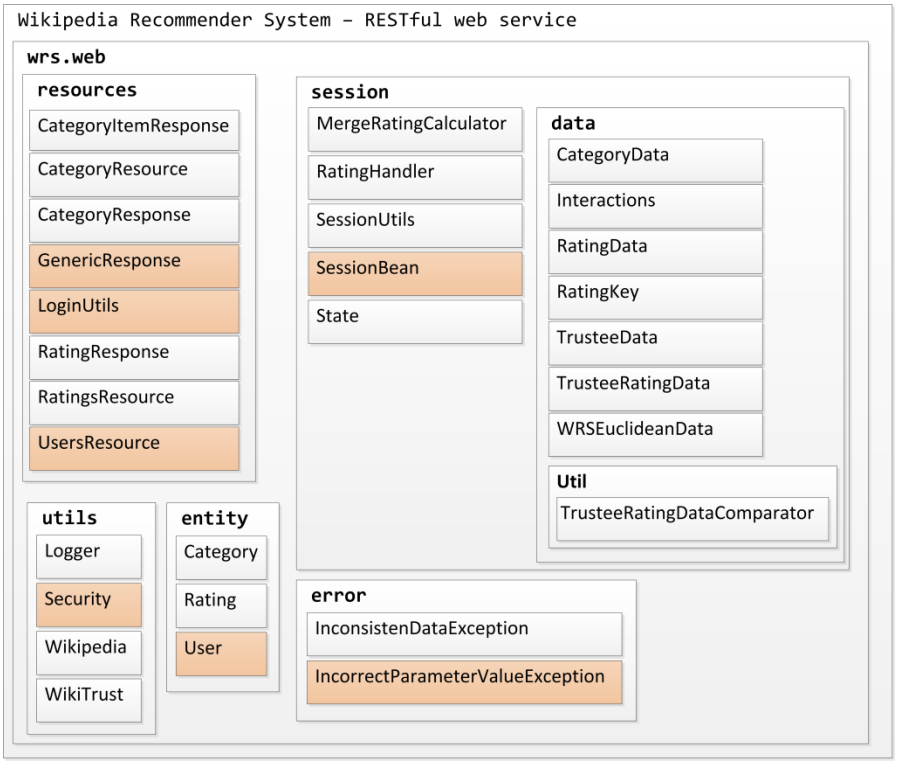

Figure 25 - Classes referenced by the createUser method

5.2.2.2 verifyCredentials

The `verifyCredentials` method takes the same parameters as `createUser` but ambiguity is avoided since it is called as a GET request even though the actual URL is identical:

```
/wrs-webapp/users/USERNAME?password=PASSWORD
```

Code listing 5 - Calling the verifyCredentials resource

The method tries to find the user in the database in the same way as `createUser`. The submitted password is hashed and compared to the hashed password stored in the user table. In the case where something goes wrong the method throws an exception in the form of

`IncorrectParameterValueException` informing the user of either a wrong username or password. If nothing goes wrong, the extension treats it as a successful verification.

The `verifyCredentials` method depends upon the same packages as `createUser` only with the addition of the `IncorrectParameterValueException` package.

5.2.3 RatingsResource

The ratings resource provides means of adding ratings on behalf of a user and returning a predicted article rating when possible. It consists of the following methods:

1. `setRating`
2. `getRating`

5.2.3.1 setRating

The `setRating` method takes a group of parameters as seen in the following URL that will call the service:

```
/wrs-webapp/ratings/USERNAME?pageUrl=PAGEURL
    &password=PASSWORD&category=CATEGORYID&rating=RATING
```

Code listing 6 - Calling the setRating resource

USERNAME and PASSWORD are included for authentication purposes, such that the user currently logged in is checked in the database as described in the `verifyCredentials` method above. If no error is thrown the method will proceed.

USERNAME is implemented as a path parameter while PAGEURL, PASSWORD, CATEGORYID and RATING are all query parameters.

The method creates an instance of `RatingHandler` and calls the `setRating` method on it, which takes four parameters: USERNAME, PAGEURL, CATEGORYID and RATING.

The `setRating` method gets an instance of the current `State` implemented as a singleton and calls `addRating` on the state, providing it with the same parameters. `AddRating` adds the rating to the database through the `SessionBean` class using JPA in a similar manner to the `verifyCredentials` procedure, which will not be further explained. It then carries on updating the in-memory representation of data to be in sync with the database in order to reflect the newly added rating using the following four methods:

- extendUsernameToRatingsMap
- extendArticleUrlToRatersMap
- extendRatingKeyToRatingDataMap
- extendUsernameToInteractions

5.2.3.1.1 extendUsernameToRatingsMap

Gets the list of pageURLs that the provided USERNAME has rated, from the usernameToRatings dictionary and adds PAGEURL to the list. If the username does not currently exist as a key in the dictionary, add it with PAGEURL as the single element of the list

5.2.3.1.2 extendArticleUrlToRatersMap

This dictionary is the reverse of the above and simply adds USERNAME to the list of usernames that have rated PAGEURL.

5.2.3.1.3 extendRatingKeyToRatingDataMap

`RatingKey` is a simple data structure containing a username and a pageUrl which together forms a pair that works as the key to the dictionary. For the key [USERNAME, PAGEURL], an instance of `RatingData` is inserted containing the tuple:

- [RATING, CATEGORYID, date, USERNAME, PAGEURL]

5.2.3.1.4 extendUsernameToInteractions

Extending the `usernameToInteractions` dictionary means adding the raters of PAGEURL to USERNAME's interactions and for all these raters also adding USERNAME as an interaction on PAGEURL.

First the list of raters who has rated the article at PAGEURL is retrieved with:

```
List<String> currentRaters = this.articleUrlToRaters.get(pageUrl);
```

Code listing 7 - Get the list of raters of *pageUrl*

And the interactions of USERNAME are found:

```
Interactions interactions = this.usernameToInteractions.get(username);
```

Code listing 8 – Getting the matrix of trustees of trustor and for each trustee the mutual articles

For each of the raters, check if the given rating is to be counted as an interaction, if the time between the two ratings is no more than 12 months and one of the following conditions is true:

1. The difference between ratings is at most 2
2. The category given for the two ratings is the same

The dictionary with USERNAME's interactions takes the rater as a key and returns a list of pageURLs where there is an interaction between USERNAME and that particular rater. If the interaction is to be counted as per the above conditions, then add PAGEURL to the list of interactions of the user:

```
interactions.getMap().get(rater).add(pageUrl);
```

Code listing 9 - Adding *pageUrl* to *trustor's* interactions with *rater*

Now loop over the raters of PAGEURL one more time, but this time in order to add USERNAME to the rater's interactions. This works the same way as above, but the interactions are as seen from the rater's perspective:

```
Interactions interactions = this.usernameToInteractions.get(rater);
interactions.getMap().get(username).add(pageUrl);
```

Code listing 10 - Adding *pageUrl* to *rater's* interactions with *trustor*

If the `get` operation returns `null` given a rater or username as the key, then the key is added to the dictionary prior to adding PAGEURL to the list of interactions.

Finally a `GenericResponse` is returned to the Chrome extension with `result = true`.

The `setRating` method depends on the following classes:

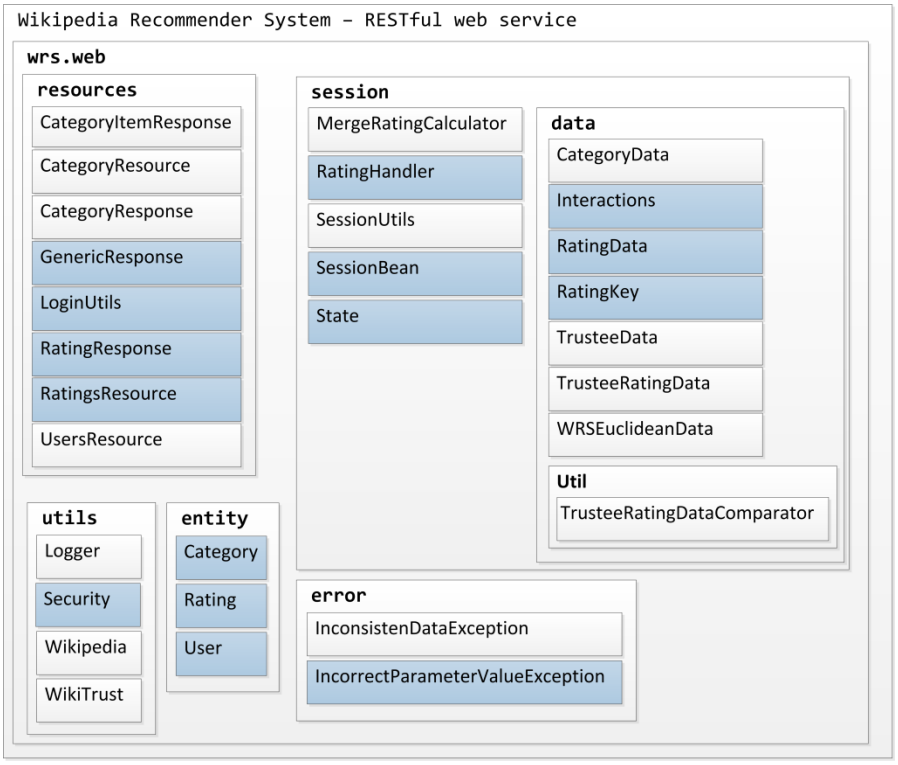

Figure 26 - Classes referenced by the setRating method

5.2.3.2 getRating

The final method exposed by the web service and by far the most complicated one, is the one used for retrieving a rating already given to an article or for predicting a rating for new articles, based on previous interactions with other users.

The `getRating` method takes 3 parameters as seen in the following URL that will call the service:

```
/wrs-webapp/ratings/USERNAME?pageUrl=PAGEURL&password=PASSWORD
```

Code listing 11 - Calling the getRating resource

It authenticates the user in the same way as `setRating` and proceeds if successful, by creating an instance of `RatingHandler` upon which it calls the `getRating` method. It looks in the `ratingKeyToRatingData` dictionary for a rating given by the active user and if found will be returned to the client and presented to the user. If no such rating exists, it will continue and attempt to predict a rating using the `MergeRatingCalculator` class.

`MergeRatingCalculator` carries out two or three steps.

The first step is to find all trustees whom the trustor has had direct interactions with, through a mutually rated article and who have rated the article for which the trustor requires a predicted rating. This list is saved in a `relevantTrustees` variable.

If `relevantTrustees` is not empty, the system calculates similarity coefficients between each pair of trustor and trustee and uses it to calculate the rating as has already been described in detail earlier in this report.

If `relevantTrustees` is empty, an extra step is carried out in which the combined list of articles rated by the trustor and its trustees is computed. Using this, the next level of neighbors is found, which consists of every user the trustees have had interactions with and with the added condition that an interaction for this step only counts if the average difference between the ratings of trustor and trustee is at most 2. The predicted rating is now also based on the neighbors of trustees as per the detailed description of the MergeTrust algorithm.

The final step is to incorporate WikiTrust into the predicted rating. The rating from WikiTrust is weighted as a trustee with perfect correlation. This is only achieved with regular users if the trustor and trustee share at least 10 ratings and that each of them are pairwise identical.

Finally the rating is returned to the Chrome extension in a `RatingResponse` object together with the category. The category is computed as the category that the majority has rated it as, along with the percentage of this majority.

The only thing remaining for the `getRating` method is an overview of the classes that this method depends on, and this diagram clearly shows that this is the most central of the five methods exposed by the web service:

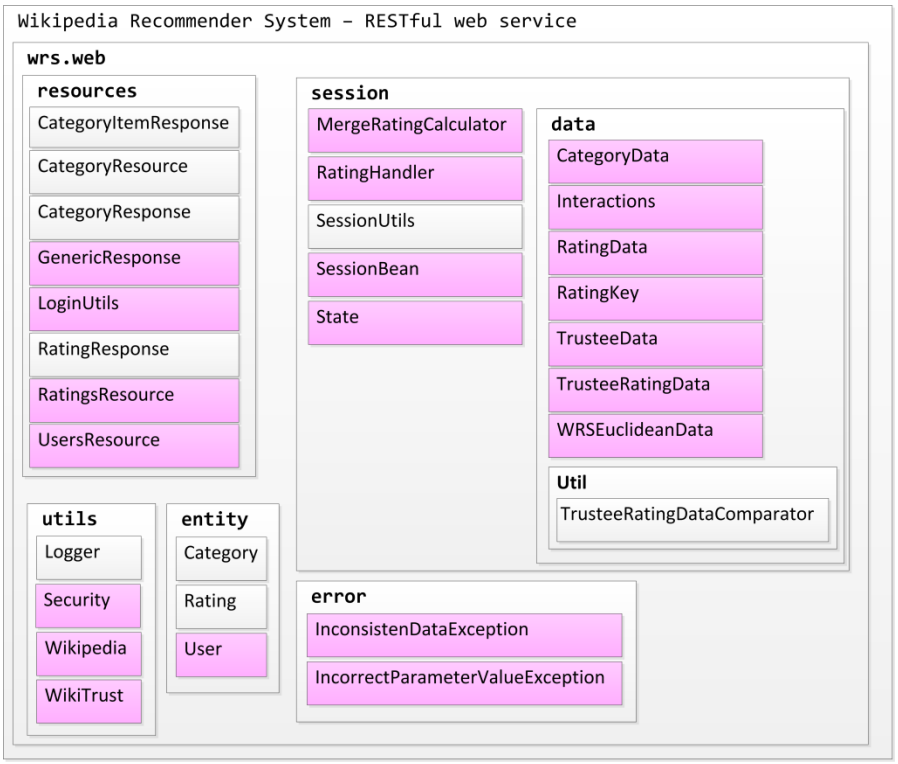

Figure 27 - Classes referenced by the getRating method

5.2.4 Undescribed classes

Two classes did not appear as being used by any of the five methods, namely `Logger` and `TrusteeRatingDataComparator`. The reason is that `Logger` was only used for debugging purposes and as such, it has been used quite extensively throughout the entire backend, but it could be removed altogether at this point. The comparator is used only for our `JUnit` test suite covered in *Evaluation*.

5.3 JPA in WRS

JPA is an abstraction layer which makes it possible to work on database tables through Java objects instead of through SQL statements. An example of the use of JPA is seen below. The example is taken from the `User` class, and is annotated using a variety of annotations from the `java.persistence` package:

```
@Entity
@Table(name = "user")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "User.findAll", query = "SELECT u FROM User u"),
    @NamedQuery(name = "User.findByUsername", query = "SELECT u FROM User u
        WHERE u.username = ?1")})
public class User implements Serializable {
    … class logic excluded in snippet ..
}
```

Code listing 12 - Example of JPA annotations

The `NamedQueries` are defined in the same class as the entity that the queries reflect, and can then be called elsewhere in the program. The `createUser` method is an example of this, which checks if the submitted username already exists by calling the `User.findByUsername NamedQuery` in the following way:

```
Query q = entityManager.createNamedQuery("User.findByUsername");
q.setParameter(1, username);
return (User) getSingleObjectOrNull(q.getResultList());
```

Code listing 13 - Using JPA to get a user by *username*

By using `EntityManager` as shown above, the query will directly return an instance of the `User` class if the username was found and `null` otherwise. If the username did not already exist in the database, `entityManager` is used in a similar way as above for writing back to the database in order to finally create the user in the system.

5.4 WikiTrust

The implementation of WikiTrust into WRS is limited to one main Java class, `Wikitrust.java` and its helper class `Wikipedia.java`.

`Wikitrust.java` contains the `getPageRating` method which is a static method that takes an article URL as the only parameter and returns the computed rating as an integer.

The rating relies on an external web API located at `http://en.collaborativetrust.com/`, which exposes a set of methods, of which WRS uses only one, wikimarkup. The method is called as follows:

```
/WikiTrust/RemoteAPI?method=wikimarkup&pageid=PAGEID&revid=REVID
```

Code listing 14 - Calling the WikiTrust API for WikiTrust markup of *PAGEID*

The method above takes a PAGEID and REVID, corresponding to the ID of the article and the ID of the revision of that article, both being IDs retrieved from Wikipedia.org.

These parameters can also be used to retrieve the article through the Wikipedia API instead of the WikiTrust API using the following call:

```
http://en.wikipedia.org/w/api.php?format=xml&action=query&revids=REVID
    &prop=revisions&rvprop=content
```

Code listing 15 - Calling Wikipedia API for wikimarkup of revision with *REVID*

For REVID = 411787463, which is a revision of the “Muslim Brotherhood” article, a subtext could be as follows:

```
===Main activity-plan===
The main goals on mid-term as approved by the Executive office and the Shura
Council are formulated in a 5-year action plan derived from transcripts.
```

Code listing 16 - Wikimarkup as received from calling Wikipedia API

When the same article is retrieved through the WikiTrust API, the subtext becomes:

```
==={{#t:10,217630808,BoogaLouie}}Main activity-plan===
{{#t:10,206853346,BertVorenk}}The main goals on mid-term as approved by the
Executive office and the Shura Council are formulated in a
{{#t:9,206853346,BertVorenk}}5-year
{{#t:9,217630808,BoogaLouie}}action {{#t:8,217630808,BoogaLouie}}plan
{{#t:7,206853346,BertVorenk}}derived {{#t:4,206853346,BertVorenk}}from
{{#t:3,206853346,BertVorenk}}transcripts.
```

Code listing 17 - WikiTrust markup as received from calling WikiTrust API

The added markup follows the format `{{#t:RATING,REVID,USERNAME}}` and is used to rate subtexts of an article according to different measures. The markup `{{#t:10,217630808,BoogaLouie}}` means that the text following this markup and continuing until the next similar markup, was written by the user BoogaLouie in revision 217630808 and that the text has received a rating of 10.

Under the hood, the workings of WikiTrust resembles that of a reputation system in that the rating of an article should reflect the trustworthiness of the users that has contributed to the different revisions of the article.

In order to become a trusted user, the system does not rely on issued trust statements as it is seen on MoleSkiing.it or similarity scores as in WRS. Instead a user gains trust through article contributions that survive revisions by other users. The trust gain is proportional to the trust in the user doing the subsequent revisions. Similarly changes that are reverted will result in a loss of trust, again proportional to the trust in the user that does the reverting. The result is that the trust of the user will primarily be based on other users and their trust, effectively limiting the impact of vandalism on both an article's rating, but also on a user's trust. It also limits the effectiveness of a vandal creating a large number of users so as to boost the trust in him or the vandal's revisions. This is due to the fact that if a user has his changes reverted by a user with low trust, then the impact will not be as great as if the user doing the reverting had had a high trust.

WikiTrust is available as a plugin for Mozilla Firefox, which works on a selected number of languages on wikipedia.org or it can be installed as a plugin on the server, providing the functionality out of the box for that wiki, without running the plugin. The plugins provide a visual indication of the trust in a subtext of an article as seen in the following graphic:

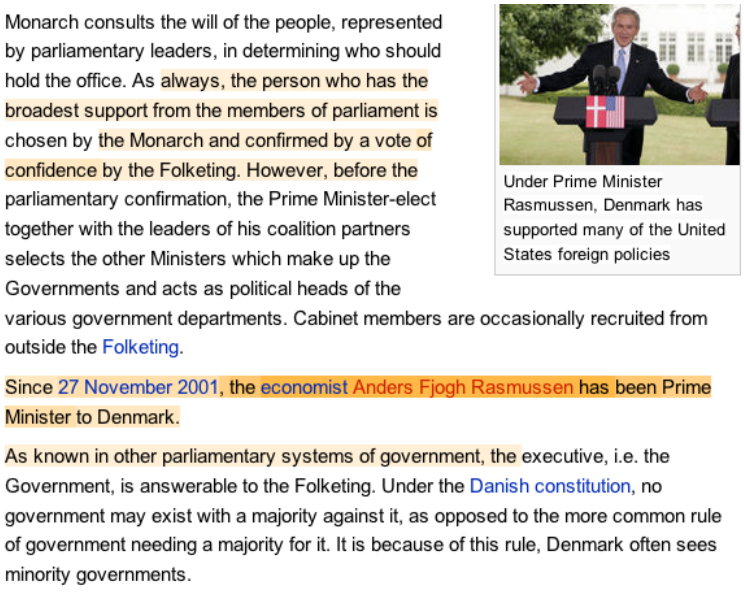

Figure 28 - Example of color coding in WikiTrust. Indications that Anders Fogh Rasmussen is a fool is easily spotted

In WRS, the different ratings of subtexts are used to calculate a weighted average of the entire article, so as to present the user with a single rating as an indication of the quality of the article.

The final rating is calculated by summing up the ratings, `trustArray`, multiplied by the length of the subtexts, `trustWeightArray`, and keeping track of the combined length:

```
for (int i = 0; i < trustArray.size(); i++) {
    rating += trustArray.get(i) * trustWeightArray.get(i);
    totalWeight += trustWeightArray.get(i);
}

rating = rating / totalWeight;
```

Code listing 18 - Rating calculation

This rating is given on a scale from 0 to 10 and is converted to a 1-9 scale in order to fit in with the rest of WRS.

5.5 Summary

In this chapter we have briefly described the architecture of the Chrome extension in terms of what files the extension contains, and what purpose each file serves. We have also described the architecture of the WRS system and which resources the RESTful web service exposes. For each resource we illustrate which classes of the system the methods of the resource depend on.

Finally we have given an example of the usage of JPA within WRS and we have described how the APIs of WikiTrust and Wikipedia are used to retrieve ratings for Wikipedia articles.

6 Evaluation

In this chapter we will reflect over the process of working on WRS and taking over a project that has evolved over several master theses, and which challenges this process presented us with.

We will also look into the process of testing both the WRS backend through unit testing, and the client application using black-box testing.

Finally we will briefly reflect over the result of our work with WRS in terms of which goals we achieved, and what was left out.

6.1 Taking over WRS

When extending and documenting a system based on work done in a sequence of previous theses, the early phases of the project were spent reading these theses and looking at their implementations, in order to get an overview of the current state of the project and getting an idea of the direction our own thesis was going to take. The obvious place to look for ideas was in the *Future Work* sections of the previous theses, of which we have implemented some points while others were deemed out of the scope of our thesis.

Prior to the process of scrutinizing the work of the previous theses, we created an early blueprint of the tasks which required solving, and this plan ended up being close to the tasks that we actually solved. Some pieces of the system, such as the Chrome Extension turned out to require less work than we anticipated, due to the fact that it was constructed in a relatively simple manner and that it was based on easily comprehensive JavaScript.

The backend on the other hand required almost completely rebuilding, beginning with the resources exposed by the RESTful web service. These were not in line with the best practices of REST, in the sense that every method used the same GET request and used a query parameter to actually determine what the desired functionality was. On top of that, creating users and adding ratings obviously had side-effects, thus violating the best-practice of GET requests to not change the state of the application. We also realized that the way JPA was utilized was suboptimal, and we wished to re-construct the way that trust/similarity was handled.

The implementation of trust propagation also required quite a bit more work than we anticipated, as we established that the concept had only been introduced but never actually found its way into WRS. We spent a great deal of time researching whether the reason for not implementing it was actually that there were implications we had not foreseen, but this turned out not to be the case.

We feel that we have succeeded in integrating the two most recent theses in such a way that the system lives up to the initial requirements. We have refactored the code so as to remove obsolete code and sought to increase the general code quality of the system as a whole. Finally we have introduced some improvements of our own such as keeping the state of WRS in memory and also implemented a more efficient trust propagation algorithm than what was originally proposed in a previous thesis.

6.2 Stability of WikiTrust

When we tested the integration with WikiTrust, we occasionally received glitches which at first appeared to be bugged code on our part. It quickly became apparent that the cause of the glitches was out of our hands.

During the initial part of our development phase, we encountered only occasional problems, but these problems became more frequent up to a point where it stopped working completely. At this point we contacted the developers of WikiTrust. Unfortunately we learned that the developers were no longer actively developing WikiTrust, and that this was a decision that had been made quite a while before we actually started work on this thesis. However, the developers kept the WikiTrust server running and stated that they had been maintaining WikiTrust and will continue to do so for as long as it only requires a minimum of effort.

The reason for the period in which it stopped working completely, turned out to be that the server had crashed and that they had been unaware of this. For quite some time after turning the server back on, WikiTrust was unavailable as a result of having to catch up with all revisions in Wikipedia from the period where the system was offline. This however did not go as expected either, since Wikipedia had changed the API, resulting in corrupt data. When we contacted the developers, we were told that the server was now correctly processing the articles but that only 10% of the English articles had been processed and that we were looking at another few weeks before the system would catch up with Wikipedia.

For the remaining bit of development of our version of WRS, we were forced to carefully select the articles that we could use for testing in order to find the ones that WikiTrust had already processed. We also had to modify WRS to be more fault-tolerant to the situation where the WikiTrust API provides invalid responses, since this happened frequently and could not be allowed to further break the functionality of WRS.

For the continued development of WRS in the future, one should consider whether relying on WikiTrust in its current form is a good idea, since it seems possible that the service is suddenly going to be shut down in the middle of the process, potentially removing core functionality from WRS.

One potential solution is to host WikiTrust along with WRS. The source code of WikiTrust is open source and freely available and could be deployed to a local server straight away, instead of relying on maintenance by the original developers, since it is only natural that they will move on eventually.

Deploying WikiTrust locally could even open up for a whole new branch of functionality that would not be practical otherwise, due to the overhead introduced by calling an API across the globe.

6.3 Unit testing

As a means of easily testing the code of WRS in an automated fashion, the Java unit testing framework JUnit has been used extensively throughout the implementation phase. Unit testing not only allows for functional testing of a recently implemented part of the program in a structured manner, but it also makes it possible to test if any particular function is still intact later in the implementation phase. In this way it is possible to ensure that as the program evolves, the functional requirements from the early unit tests are still met, simply by running the automated tests and thus making sure nothing is broken in the process.

The test suite is structured into four test classes:

1. CreateUserTest
2. AddRatingTest
3. StateTest
4. MergeRatingFunctionalityTest

6.3.1 CreateUserTest

This class tests the `SessionBean` class and thus the database interaction, by creating a new user and getting it again through the same `SessionBean` class. The test does assertions on the username and also tests whether the hashing in the `Security` class is able to match the hashed password in the database. Finally the test deletes the user and makes sure that it no longer exists in the database, which successfully makes the test pass.

6.3.2 AddRatingTest

In order to test the “add rating” functionality, the functionality of `CreateUserTest` is reused for the `setUp` and `tearDown` parts of the unit test, such that a test user is available. The rating is added with some predefined variables and then retrieved and compared to these variables. The test passes if all variables match and that the rating is successfully deleted in the end.

6.3.3 StateTest

Up until now, the tests have been centered on the interactions with the database. This test case on the other hand focuses on the functionality of the in-memory database in the form of the singleton `State`.

First three users are added, *username1*, *username2* and *username3*, then ratings are added such that the first two users have rated the same three articles and *username3* has only rated two of them. The final goal is to predict a rating for *username3* on the last article. The scenario is depicted below, where the numbers on edges are ratings given by the user for that article:

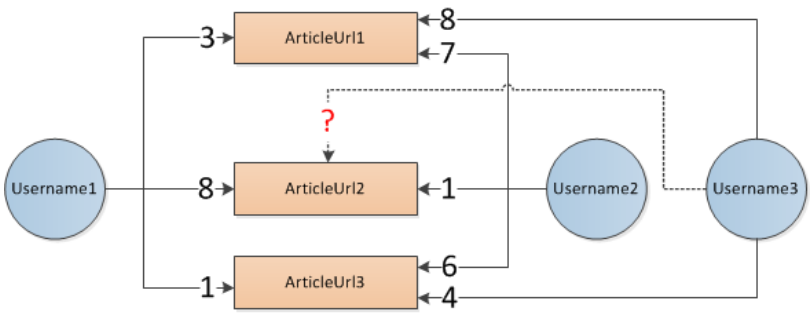

Figure 29 - A small network of ratings, only username3 has not rated articleUrl2, but he can use username1 and username2's ratings to predict one

The `StateTest` test class' core purpose is to test whether the in-memory database in the form of `State` is correctly updated whenever users and ratings are added, so as to reflect the given ratings as well as potential interactions. All the maps that were described earlier are tested thoroughly in order to check that everything is in place according to Figure 29 and also that nothing extra is added along the way.

First up is the `usernameToRatings` map, which is tested to ensure that given *username1*, three ratings are returned and that they are the correct values and URLs, just like *username2* and *username3* are tested similarly.

Next is the `ratingKeyToRatingData` map, to check if it is possible to create a new instance of `RatingKey` on the basis of a username and article URL pair, and given this newly created key, that the map returns the correct rating.

Third is the `articleUrlToRaters` map, which given a URL should return the list of raters that has rated it.

The last of the four maps is `usernameToInteractions`, which given *username1* will return a new map. The keys of this new dictionary are usernames of users with which *username1* has had interactions. The values are lists of articles where the interactions have taken place. Getting the list for *username2* from this second map will correctly return the three articles that they have mutually rated. All other combinations are also tested successfully.

The final step of `StateTest` is to get the ratings of all users on all articles, but this time not directly through the state but rather through `RatingHandler`'s `getRating` method used for calculating a rating, in order to ensure that for all users on all articles, it simply returns their own rating. Calling the `getRating` method on `ratingHandler` in the same way as before, but for *username3* on *article2*, which has not yet been rated by this user, will correctly compute a rating based on previous interactions with *username1* and *username2*.

The example from above will result in the following computation:

The formula for our similarity metric, SED is:

$$si = \frac{1}{1 + d^2(p, q)}$$

The distance between *username1* and *username3*'s ratings for *articleUrl1* and *articleUrl3* are 5 and 3 respectively, whereas the distance between *username2* and *username3* are 1 and 2 for the same articles. This gives:

$$s_1 = \frac{1}{1+5^2+3^2} * 0.2 = 0.005714, s_2 = \frac{1}{1+1^2+2^2} * 0.2 = 0.033333$$

Where the multiplication of 0.2 is due to having two ratings in common with both users. These similarities are now used for predicting the rating of *articleUrl2*, using *username1* and *username2*'s ratings for this article (8 and 1 respectively):

$$r_{articleUrl2} = \frac{s_1 * 8 \ + \ s_2 * 1}{s_1 \ + \ s_2} \approx 2$$

The test correctly returns this value.

6.3.4 MergeRatingFunctionalityTest

This test is by far the most extensive and also the one that required the most effort. It tests a large number of methods involved in the process of predicting a rating using trustees and trust propagation. The test is structured in a way such that the helper functions are tested from the innermost level and out, meaning methods that depend on the functionality of others are tested *after* their dependencies. The `getRating` method of `MergeRatingCalculator` has a method tree that looks as follows:

Figure 30 - The numbers denote the order in which the methods return

Most of these methods are defined as private methods in order to conform to the Java standards of *encapsulation*, and this means they cannot be tested directly from an external test class. Instead the

unit testing relies on reflection to allow for external calls of the private methods. Only a few of these methods will be explained in detail with source code.

An example of calling `getMergedTrustedNeighbours`, which is the first method tested, is as shown in the following snippet:

```
Class c = Class.forName("wrs.web.session.MergeRatingCalculator");
Interactions interactions = state.usernameToInteractions.get(username1);

Class[] inputClasses = new Class[] { Interactions.class, String.class };
Method method = MergeRatingCalculator.class
    .getDeclaredMethod("getMergedTrustedNeighbours", inputClasses);
method.setAccessible(true);

Object[] inputObjs = new Object[]{ interactions, username1 };
Map<String, Double> trustedNeighbours =
    (Map<String, Double>) method.invoke(c.newInstance(), inputObjs);
```

Code listing 19 - getMergedTrustedNeighbors example

Using this approach it is no longer possible to utilize strongly typed methods, but instead the test relies on `String` literals for defining the methods that are to be called.

The result of `getMergedTrustedNeighbours` is a map of neighbors with direct interactions with the active user, where the key is the username of the neighbor and the value is the computed similarity score.

The next method is `getRatingsOfNeighbour`, which returns a map from an article URL to `TrusteeRatingData`, which contains the username of the rater and the actual rating. The snippet below shows another example of the use of reflection and how much of the code that is reusable:

```
inputClasses = new Class[] { Map.class };
method = MergeRatingCalculator.class.getDeclaredMethod("getRatingsOfNeighbour",
    inputClasses);
method.setAccessible(true);

inputObjs = new Object[]{trustedNeighbours};
Map<String, List<TrusteeRatingData>> articleToRaters = (Map<String,
    List<TrusteeRatingData>>) method.invoke(c.newInstance(), inputObjs);
```

Code listing 20 - getRatingsOfNeighbour example

In order to thoroughly test the contents of this map, we have to check the ratings for every user on every article, but the order of the users in a list of `TrusteeRatingData` is not something of concern with regards to normal operation of the system. Instead we implemented the `TrusteeRatingDataComparator` which is used to sort the list according to username and when we know the order, we can simply check for rating equality as shown below:

```
List<TrusteeRatingData> trusteeRatingData = articleToRaters.get(articleUrl1);
Collections.sort(trusteeRatingData, new TrusteeRatingDataComparator());
assertEquals(6, trusteeRatingData.get(0).getRating());
assertEquals(4, trusteeRatingData.get(1).getRating());
```

Code listing 21 - TrusteeRatingDataComparator example

The next couple of methods being tested are left out, because the approach follows the same general concept as shown already. The final step is to predict a rating for *user 1* on *article 9* as per the example used in this test case:

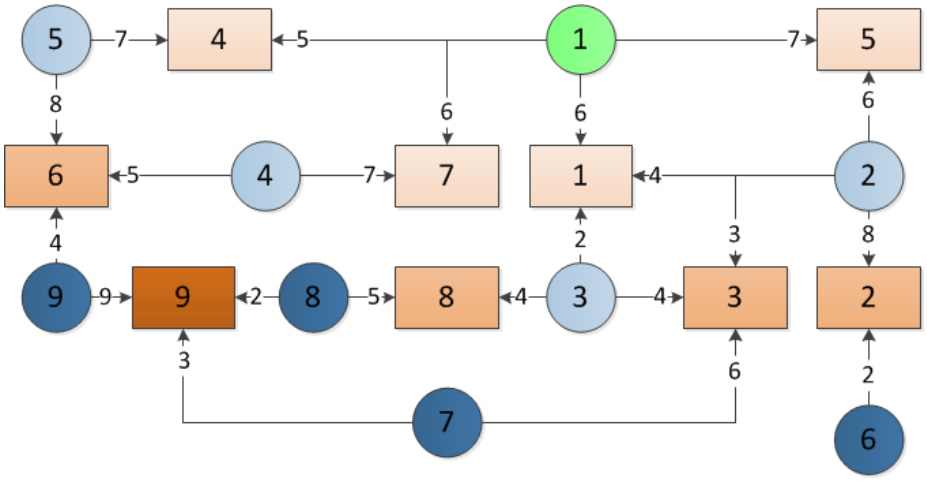

Figure 31 - Circles, rectangles and edges are users, articles and ratings, respectively.

For the example above, we will also show the computation of the predicted rating for *article 9* and compare it to the result returned by the method. The trustor is shown as the green circle, while the light blue circles are trustees reachable through articles in common with the trustor, depicted as the rectangles with the lightest shade of orange. The dark blue circles are the nearest neighbors, which are reachable through articles in common with the trustees, this time depicted in a slightly darker shade of orange. Finally the darkest shade of orange shows the article that requires similarity propagation through the trusted neighbors and the nearest neighbors in order to predict a rating.

First we calculate the merged rating for *article 6* on basis of the two users *user 4* and *user 5* who both rated the article. The similarity scores for these two are:

$$s_4 = \frac{1}{1+1^2} * 0.1 = 0.05,\ s_5 = \frac{1}{1+2^2} * 0.1 = 0.02$$

The rating of *article 6* then becomes:

$$r_{article6} = \frac{s_4 * 5 \ + \ s_5 * 8}{s_4 \ + \ s_5} \approx 6$$

The similarity with *user 9* can now be calculated from the merged rating $r_{article6}$ and the rating for *article 6* given by *user 9*:

$$s_9 = \frac{1}{1+2^2} * 0.1 = 0.02$$

Thus one path to *article 9* is covered. The second path goes through *article 1* and *user 3*, but we observe that the average difference between ratings given by this user and the active user on their only mutual article is greater than 2 and thus it is disregarded.

The last path goes through *article 1* and *article 5* over *user 2*. The similarity with *user 2* is:

$$s_2 = \frac{1}{1+1^2+2^2} * 0.2 = 0.0333$$

Since *article 3* has been rated only by *user 2* and *user 3*, where *user 3* was disregarded, we simply keep *user 2*'s rating as the merged rating. We now calculate similarity with *user 7*:

$$s_7 = \frac{1}{1+3^2} * 0.1 = 0.01$$

Finally the rating for *article 9* can be calculated from s_7 and s_9:

$$r_{article9} = \frac{s_7 * 3 \ + \ s_9 * 9}{s_7 \ + \ s_9} \approx 7$$

Once again, this is also the result returned by the `getRating` method.

6.4 Black-box testing the client application

In order to assert that the Chrome extension and WRS work as expected, we perform a number of regular operations through the extensions and verify that we get the expected result.

In the test environment we have the Chrome extension loaded in the browser, and we have an empty (no ratings or users) WRS database - WRS is running on localhost.

The tests are done sequentially, so each test assumes that all previous tests are successfully completed.

6.4.1 Initial tests

The following black-box tests run through creating a new WRS user and session management. It also tests the basic functionality for retrieving ratings from WikiTrust

Test number	Test description	Expected result
1	Browse to any page *not* under http://en.wikipedia.org.	The WRS icon does not appear in the address bar.
2	Browse to any article under http://en.wikipedia.org.	1: WRS icon in the address bar 2: A dialog pops up notifying the user that he or she should log in.

3	Click the WRS icon in the address bar.	A popup with a login button appears.
4	Click the button labeled “Log in”.	Page change to a page with input fields for username and password and a button labeled: “Don’t have an account?”.
5	Click the button “Don’t have an account?”.	Two new input fields for username and password for a new user appear along with a “Sign up” button.
6	Click the button “Sign up”.	A dialog notifies that username and password must be specified.
7	Enter username “John”, password “123456” and click “Sign up”.	A dialog notifies that the user was successfully created.
8	Perform test 7 again.	A dialog notifies that the username is taken and that another should be used.
9	Enter “John” and “123” in the username and password login fields and click “Log in”.	A dialog notifies that the password is invalid.
10	Perform test 9 again with username “Bob” and password “123456”.	A dialog notifies that the username is invalid.
11	Perform test 9 again with username “John” and password “123456”.	Page change: A new page shows that the user “John” is logged in.
12	Click the button labeled “Close window”.	The WRS window closes.
13	Browse to any Wikipedia article where WikiTrust provides trust data.	After a few seconds a dialog shows up displaying a rating for the article.
14	Shut down the browser.	-
15	Open the browser and navigate to any Wikipedia article where WikiTrust provides trust data.	1: The user is automatically logged in using the credentials from the previous session. 2: After a few seconds a dialog shows up displaying a rating for the article.
16	Click the WRS icon in the address bar.	A popup shows up. The popup shows the same rating as the dialog.
17	Click the button labeled “Options”.	Page change to a page that contains a checkbox labeled “Don’t display rating notification”.
18	Check the checkbox and click the button labeled “Close window”.	The window closes.
19	Browse to any Wikipedia article where WikiTrust provides trust data.	A notification about the rating of the article *does not* show up.
20	Execute test number 16 and 17 and uncheck the checkbox “Don’t display rating notification”. Click the button labeled “Log out”.	Page change to the page which was also the result of test number 4.
21	Restart the browser and browse to any Wikipedia article where WikiTrust provides trust data.	The result should be the same as the result of test number 2.

Table 17 - Stage 1 of the black-box test of WRS

6.4.2 Testing rating functionality

The following tests go through the functionality of rating articles through WRS.

These tests will not focus on rating predictions with or without similarity propagation, as this part of the functionality was tested using unit tests. It will focus purely on the behavior of the Chrome extension when it interacts with WRS.

22	Log in to WRS using the credentials of the user “John”. Navigate to any Wikipedia article where WikiTrust provides trust data.	Same as for test 13.
23	Open the WRS popup by clicking the icon in the address bar.	The popup opens, and the article rating is displayed. The category is “Unassigned category”.
24	Click any number button to assign that rating to the article.	A text notifying the user that a category must be selected before that rating can be submitted.
25	Click the select box labeled “Category”.	A list of categories appears.
26	Select any category and then click any number button to rate the article.	A text notifying the user that the rating was successfully submitted.
27	Close the popup by clicking anywhere on the page outside of the popup. Refresh the page that was just rated.	A dialog shows up and displays the rating and category that was just assigned to the article.
28	Open the popup for the same article.	The same rating and category that was just assigned to the article appears in the popup.
29	Select a new category and submit a new rating for the same article.	A text notifying the user that the article has already been rated.

Stage 1 of the black-box test of WRS

6.4.3 Black-box testing results

All 29 steps of the two groups of tests presented above have been successfully completed, and for each test the actual result was in accordance with the expected result.

A test or series of tests cannot prove the absence of errors in a program. However, since the test steps listed above activate all main parts of the Chrome Extension and the RESTful API, we feel confident that the system does not contain serious errors that hinder the basic functionality of WRS.

6.5 Determining the efficiency of the new solution

In order to determine how effective the new solution to the Cold Start problem of WRS is compared to the previous solutions, we run an experiment. The experiment will show the increase in the number of ratings that can be used as a basis for calculating a predicted rating.

The experiment is executed in the following way:

1. We define *n* users and *m* articles
2. For each user we distribute between 0 and 10 ratings among the articles (The number of ratings follows a uniform distribution, and the distribution of ratings on articles is also uniform)
3. We create a new user *n+1* and measure how the number of available ratings increase as the new user rates (random) articles

The following table shows an instance of an experiment before user *n+1* is added where *n = 10* and *m = 30*. The data has been created using a Java program where the randomization has been achieved using java.util.Random. Users are denoted by a capital U followed by a number and articles are denoted by a capital A followed by a number. The intersection between user and article contains 1 if the user has rated this specific article and 0 otherwise. We are only interested in the interaction itself and not the similarity, meaning that the actual rating value is irrelevant.

	U1	U2	U3	U4	U5	U6	U7	U8	U9	U10
A1	0	0	0	0	0	0	0	0	0	0
A2	0	0	0	0	0	0	0	0	0	1
A3	1	0	0	0	0	0	0	0	0	1
A4	0	0	0	0	0	1	0	0	0	0
A5	0	0	0	0	0	0	0	0	0	0
A6	0	0	0	0	0	0	1	0	0	0
A7	0	0	0	1	0	0	0	0	0	0
A8	0	0	0	1	0	0	0	0	0	0
A9	0	0	1	0	0	0	0	0	0	1
A10	0	0	0	0	0	0	0	0	0	1
A11	0	0	0	0	0	0	0	1	0	0
A12	0	0	0	1	0	0	0	0	0	0
A13	1	0	0	0	1	0	0	0	0	0
A14	1	0	0	0	0	1	0	0	0	0
A15	0	0	0	0	0	0	0	0	0	0
A16	0	0	0	1	0	0	0	0	0	0
A17	0	0	0	0	0	0	0	0	0	0
A18	0	0	0	1	0	0	0	0	0	0
A19	0	0	0	0	0	1	1	0	0	0
A20	0	0	0	0	0	1	0	1	0	1
A21	0	0	1	1	0	0	0	0	0	0
A22	0	0	0	0	1	0	0	0	0	0
A23	0	0	0	0	0	0	0	0	0	0
A24	0	0	0	0	1	1	1	0	0	0
A25	1	0	1	0	0	0	0	1	0	0
A26	0	0	1	0	1	0	0	0	0	0
A27	0	0	1	1	0	0	0	0	0	0
A28	0	0	0	0	0	0	1	1	0	1
A29	0	0	1	1	0	1	0	0	0	0
A30	0	0	1	0	1	0	0	0	1	0

Table 18 - User and article matrix

The table above forms the basis of an instance of the experiment.

By inspecting the table one can find the following numbers, where a total of 45 ratings have been given and are distributed as follows:

- U1 has given 4 ratings {A3, A13, A14, A25}
- U2 has given 0 ratings
- U3 has given 7 ratings {A9, A21, A25, A26, A27, A29, A30}
- U4 has given 8 ratings {A7, A8, A12, A16, A18, A21, A27, A29}
- U5 has given 5 ratings {A13, A22, A24, A26, A30}
- U6 has given 6 ratings {A4, A14, A19, A20, A24, A29}
- U7 has given 4 ratings {A6, A19, A24, A28}
- A8 has given 4 rating {A11, A20, A25, A28}
- A9 has given 1 rating {A30}
- A10 has given 6 ratings {A2, A3, A9, A10, A20, A28}

We now introduce a new user A11 who rates 6 random articles and investigate how the number of available ratings increases for each of the four WRS techniques. The rated articles are A8, A4, A25, A2, A10 and A24.

	1 rating	2 ratings	3 ratings	4 ratings	5 ratings	6 ratings
[Pilkauskas, 2010]	8	14	29	35	35	44
[Mihăilă, 2011]	38	44	59	65	65	74
[Andersen, 2011]	21	44	45	45	45	45
New version	51	74	75	75	75	75

Table 19 - Experiment 1 result: Efficiency of different WRS implementations

These numbers are plotted into the following graph, which shows that the number of ratings available when using the new version is clearly higher when compared to the previous solutions:

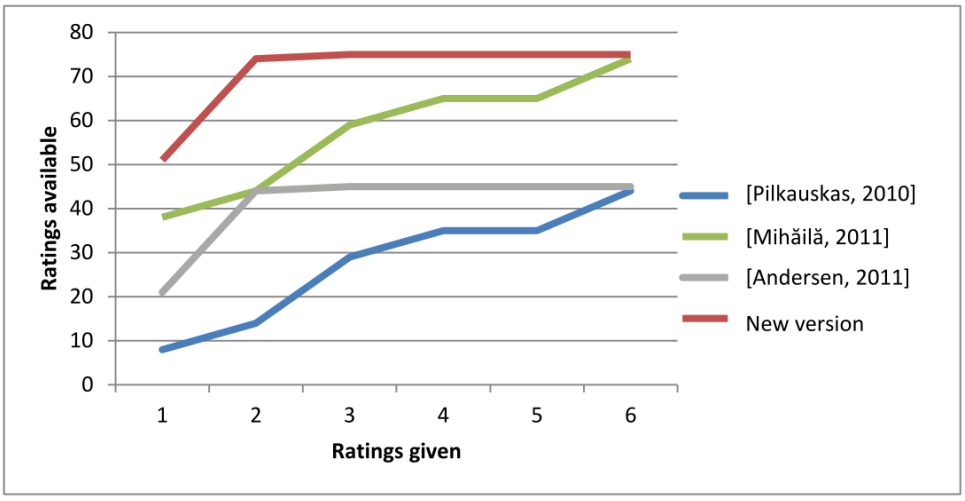

Figure 32 - Growth of the different solutions

The numbers from the table has been calculated as follows:

For the solution by Pilkauskas we start by inspecting the first rating of A8. We see that A8 has been rated by one other user, namely U4 [Pilkauskas, 2010]. As U4 have rated 8 articles, the number of available ratings is now 8. For rating 2 we see that A4 was rated by one other user namely U6 who has rated 6 articles. This means that the number of available ratings is now 8 + 6 = 14. This calculation continues for all six ratings giving a total number of 44 ratings available after 6 ratings have been given.

For the solution by Mihăilă we reuse the numbers from Pilkauskas but we add one rating for each of the 30 articles based on the contribution from WikiTrust [Mihăilă, 2011]. Therefore all the numbers are 30 ratings higher.

The numbers from the solution by Andersen are calculated as follows for rating 1: A8 has been rated by one other user, namely U4. U4 has rated the articles A7, A8, A12, A16, A18, A21, A27 and A29 [Andersen, 2011].

- Articles A7, A8, A12, A16 and A21 have been rated by U4 only and therefore do not contribute with extra ratings
- A21 has been rated by U3 who has rated 7 articles
- A27 has been rated by U3 but we have already seen the ratings from U3
- A29 has been rated by U3 and U6 who has rated 6 articles

This means that after the first rating, a total of 8 + 7 + 6 = 21 ratings are available.

For the new version, the numbers from [Andersen, 2011] are reused and the 30 ratings from WikiTrust are again included.

It is clear from the result that more ratings are available when using the new technique. For this example, almost full coverage is achieved after 2 ratings were given, while the solution that does not use similarity propagation requires 6 ratings to achieve roughly the same coverage.

It is important to emphasize that this experiment shows how the number of available ratings are affected when different techniques are used. However, when the ratings are actually calculated the number of available ratings is likely to be lower, because there will probably not be full similarity between all users in the system. However, the purpose of this experiment is not to determine the actual number of similar users, since this would require knowledge of the rating distributions that we do not have at hand at the current time. The purpose is only to determine the coverage in terms of ratings that become available for similarity and rating calculations as ratings are given.

6.6 A larger example

The example above illustrates the principle of the experiment. However, it is too small to actually show how the development will be when WRS is used for Wikipedia. For this reason we have performed the same experiment where the number of users n = 5.000 and the number of articles m = 15.000. We run the same experiment 100 times and calculate the average number of ratings available for rating prediction:

	1 rating	2 ratings	3 ratings	4 ratings	5 ratings	6 ratings
[Pilkauskas, 2010]	11	21	33	45	56	67
[Mihăilă, 2011]	15011	15021	15033	15045	15056	15067
[Andersen, 2011]	118	231	361	485	597	726
New version	15118	15231	15361	15485	15597	15726

Table 20 - Experiment 2 result: Efficiency on larger sets

The table above shows that the four techniques operate in two different intervals and as such they are illustrated in two separate figures:

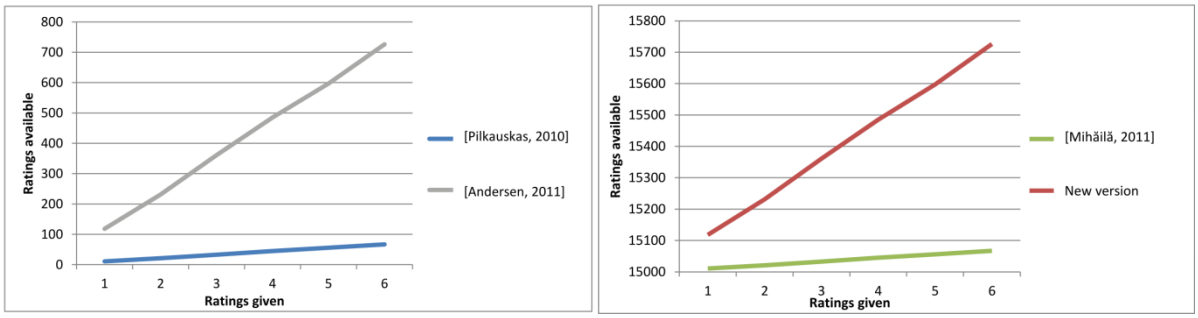

Figure 33 - Still linear, but grows faster with propagation, with (right) and without (left) WikiTrust

From the figures above, there is a clear indication that all solutions result in linear growth in the number of ratings that become available for rating prediction, but the gradient of our solution and that of Andersen is much steeper than the remaining two. At a first glance this appears not to be in line with the figure we presented in the *Introduction* chapter, where we proclaimed that propagation would result in polynomial growth, with the exponent of the equation equal to the number of levels of neighbors that are included in the propagation which in this case is 2. There is however a distinct difference between the equations that are illustrated. The difference is that this experiment focuses on actual observable data, while the introduction relied on assumptions and simplifications.

The major difference is that polynomial growth was seen as a function of the number of interactions rather than the number of given ratings. This simplification was made because there is no safe way to establish how many interactions a given rating results in and as such, the experiments above includes an extra computational step before coming up with the available ratings.

We now perform a similar experiment where we do not focus on the cold start situation. Instead we look at the development when the number of users n = 10.000 and the number of articles m = 5.000. We also increase the average number of ratings per user from 5 to 10, meaning that we have on average 100.000/5.000 = 20 ratings per article, which results in a denser dataset:

	1 rating	2 ratings	3 ratings	4 ratings	5 ratings	6 ratings
[Pilkauskas, 2010]	258	541	821	1088	1360	1630
[Mihăilă, 2011]	5258	5541	5821	6088	6360	6630
[Andersen, 2011]	46296	70912	82936	89096	92692	94859
New version	51296	75912	87936	94096	97692	99859

Table 21 - Experiment 3 result: Efficiency on dense datasets

As expected, the number of available ratings is vastly greater and if the numbers are plotted, we see the following figure:

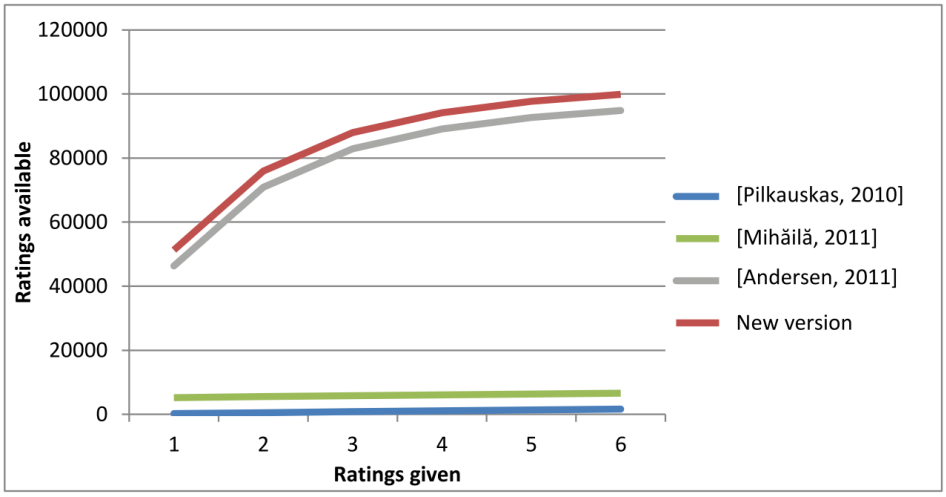

Figure 34 - Growth stagnates as coverage approaches 100%

The figure shows how the number of available ratings seems to stagnate with an increasing number of ratings given, for both our solution and the solution by Andersen. This is due to the coverage approaching 100% of the articles. The figure also shows that the coverage of our solution is marginally better than that of Andersen and significantly more so than the remaining two.

6.7 Results

The result of this thesis is a version of WRS which is both well tested and properly documented. When compared to the previous versions of WRS, the most significant contributions of this thesis can be summed up to the following five additions:

1. Added support for similarity propagation combined with contributions from WikiTrust.
2. Implemented efficient in-memory data structures in order to improve the performance of calculating predicted ratings.
3. Thoroughly analyzed the trust model from the previous version of WRS and replaced it with a new well-founded similarity measure.
4. Re-modeled the WRS backend and the RESTful API to conform to the best-practices of this type of systems, and generally refactored the Java code base to include only necessary code.
5. Added a full JUnit based test suite which greatly improves the workflow of implementing future functionality to WRS without breaking existing functionality.

As we will discuss in *Future work and research* there is still plenty of work to be done on WRS. Some of the work relates to features of the client application which have not been implemented yet because WRS is still in the prototype stage. Other work relates to testing and optimizing variables which may not currently have optimal values because we cannot determine what is optimal until WRS contains proper rating data.

6.8 Summary

In this chapter we discussed the challenges of relying on WikiTrust within WRS, because WikiTrust is no longer actively maintained, and the developers state that they will only keep it running for as long as it can be done with minimal work effort.

We continue to present full JUnit test suites for the WRS server including tests of database interaction, interactions with the in-memory data structures for maintaining the state of WRS, and predictions of ratings with and without similarity propagation.

A description of a black-box test of the WRS client is also provided, and the chapter is concluded by summing up the main contributions to WRS from this thesis.

7 Future work and research

While we feel that we have contributed with important features and improvements to WRS, the system cannot be considered complete or ready for release to the general public. This chapter will discuss some of the features and tasks that we did not complete, and could serve as a foundation for future work on WRS.

7.1 Security

The current implementation sends the username and password directly in the URL with every HTTP request going from the Chrome extension to the WRS server. Since the password is hashed before it is sent, an attacker cannot use WRS to steal the clear text password of users. However, using a man-in-the-middle attack the hashed password can still be intercepted and used to impersonate the user in relation to WRS. In order to mitigate this vulnerability, the communication with the server should be done using the SSL protocol. In order to use SSL, WRS should either purchase a certificate signed by a globally trusted entity or rely on a self-signed certificate.

Limiting login attempts

In order to limit the effectiveness of trying to “guess” a password using brute-force, there should be a delay after a failed login attempt until a new attempt can be made. The delay can initially be small enough for a human to never notice it, but the delay between each attempt can be implemented such that it increases exponentially in the number of failed attempts. Exponential growth will significantly reduce the effectiveness of brute-force attacks. In order to protect against DDoS attacks, great care should be taken as to how the delay is enforced, such that the delayed threads are not taking up unnecessary resources from the system.

7.2 Update ratings

In the present version of WRS, it is only possible to rate an article once in the entire lifetime of the article. We acknowledge the fact that an article changes over time and the basis on which a rating was given could very well change with a change in content. An article in its infancy might be only a stub and receive a lot of bad ratings as a result, but as the article matures up to a point where it would otherwise be considered an article of high quality, the initial bad ratings are still in effect, negatively influencing the rating. Even though they do stop counting towards the predicted rating for new raters after 12 months, they will still prevent the user from rating the article again in order to reflect its current content.

Not allowing the user to re-rate an article was a conscious design choice on our part, because the implications of such a feature are rather complicated. We rely on the in-memory database to provide data for the extension with as little delay as possible. The in-memory database contains dictionaries that quickly finds the relevant trustees, interactions etc., and allowing users to re-rate articles would require some of these dictionaries to be rebuilt every time, for every user that has rated that article. While the change itself is not overly complicated and could be implemented with relative ease, the testing required by this change is quite a bit trickier and since we have sought to thoroughly test the solution we are delivering, this feature was not implemented.

Prompting user for a new rating

Closely related to allowing a user to re-rate an article, the Chrome extension could be extended so as to inform the user if the user has already rated the article, but that the rating was given for an older revision. It would be subject to experiments to determine when a rating was to be considered old. The server could compare the newest revision with the revision on which the rating was given using either the rvdiff function in the MediaWiki API or using some third party tool in order to establish exactly how much the article has actually changed. With such a solution the API calls could be done from the extension instead of from the server and this would improve the efficiency of the solution because the feature would be almost solely handled client side.

7.3 Cross browser compatibility

By actively selecting to support only a single browser through a browser specific extension, we effectively limit the number of potential users. This was a choice made by Mihăilă [Mihăilă, 2011] and since implementing the same functionality in an extension for another browser would demonstrate near to nothing in terms of academic qualifications; it is a choice that we saw no reason to question. Additionally this thesis is primarily a research project and thus the system is only considered a prototype. If the current goal was widespread use of WRS, it would only be natural to create extensions for other browsers.

7.4 Features of the client

The Chrome extension and WRS system provide only a minimum number of features. In order for the extension to be a bit more user-friendly, it could support features such as:

- Recovery of a forgotten password
- Deletion of the user account
- Some form of help information embedded in the application

7.5 Supporting Wikipedia in other languages

Currently WRS only works on the English Wikipedia and never loads on any other page. This is due to the implementation of the extension which uses a simple string match with the address http://en.wikipedia.org/wiki/, which naturally limits the use to the “en” subdomain. There was nothing preventing us from extending this to allow all languages using wildcards, but we chose not to because of the simple fact that WikiTrust only works on a very limited number of languages anyway. We felt that there was no reason to allow WRS on all languages, only to have the majority of them running with limited functionality, due to the absence of WikiTrust.

Localized versions of WRS

Related to supporting Wikipedia in other languages than English, it could also be desirable to allow localized versions of the extension itself, translated to various widely used languages.

7.6 Variable values

A few different places in our implementation, we base functionality on absolute values that we cannot properly justify. For these values, real data gathered from a running version of WRS could lead to optimized values that differ from the current values. These are as follows:

- Correcting for fewer than 10 articles in common
- Counting interactions only for ratings within 12 months

7.6.1 Correcting for fewer than 10 articles in common

When we calculate the similarity scores between users, we use Squared Euclidian Distance to generate a similarity score in the range [0;1]. This value is then corrected according to the number of interactions between the users, by multiplying SED with a factor 0.x, where x is the number of rated articles in common. If there are 10 or more interactions, the score is not corrected. This correction was introduced to penalize similarities computed based on fewer interactions, but the number of interactions required before no penalty is incurred was chosen because it kept the principle simple.

7.6.2 Counting interactions only for ratings within 12 months

The original trust model that we have described in *State of the Art* progressively penalized the weight of a rating through an interaction, the further apart in time two ratings were, according to the following table:

Time difference	Weight
< 1 month	100%
1-6 months	50%
6-12 months	25%
> 12 months	0%

Table 22 - Overview of weights of ratings of increasing age

The introduction of similarity scores, does not directly allow for weighting of individual ratings when computing the similarity with a user, thus the table above is no longer in effect.

Instead we chose to implement a penalty where ratings are disregarded if the dates of the two ratings are more than 12 months apart. We acknowledge that this may possibly be a worse solution than previously, but it was not practically possible to incorporate it into the similarity score computation.

We justify this change by emphasizing the fact that the values of the table above are themselves not based on experiments or a theoretical analysis. In *Trust Dynamics* of the *Analysis* chapter we explained how time is a sub-optimal measure of the relevance of an article, since some articles could very well be changed on a day-to-day basis while others remain almost unchanged for years.

We believe that a better measure would be a percentage of how much the content of the article actually changed. One way to measure this is to use the rvdiff function of the MediaWiki API.

7.6.3 Difference in rating and category

With our hybrid solution of counting ratings by two users as an interaction only if they agree on the category or if ratings are within a distance of 2, regardless of category, is another feature that we figured would be the optimal trade-off between coverage and accuracy. However, we have nothing to base this on other than intuition. It may not work very well at all, or perhaps the absolute distance should have another value.

7.6.4 Merging requires a single article with a distance of at most two

In order to establish which trustees to treat as trusted neighbors for the MergeTrust algorithm, the trustor and trustee are only required to have a single article in common where the average difference in ratings is no more than 2. These rules were introduced because the original MergeTrust relies on directly issued trust statements, while our implementation relies on similarity scores that given an interaction will always result in a positive similarity and thus be counted as having some degree of trust. We chose to limit the expanding nature of the algorithm by choosing the value 2 as the maximum distance to consider.

7.6.5 Variable values summary

The variable values and related measures taken throughout our implementation and listed in the sections above all have a flaw in common in the sense that they are not based on experimental research but rather on careful considerations and intuition. In order to justify whether these values are appropriate or to find values that are better suited for the system, experiments should be carried out by having a number of users rate articles. The system should gather data on the difference in predicted ratings vs. actual given ratings in order to establish the *mean absolute error* (MAE). The MAE could then be used to tweak the values. The requirement for such tests to be valid is that a significant amount of real data is present in order to establish statistical significance rather than raising doubt over whether the results are simply cause by coincidence.

7.7 Similarity measure

In *Analysis* we discussed different similarity measures such as Pearson correlation, Euclidian Distance and Squared Euclidian Distance (SED). We ended up selecting SED because we believe that in WRS the similarity will often be calculated based on only a few commonly rated articles and Pearson correlation does not work well when the number of common ratings is less than 4. However, we cannot know for sure how the ratings are going to be distributed among the articles of Wikipedia, and thus it may actually be the case that there are often a high number of commonly rated articles between WRS users. If this is the case SED may end up not being the optimal choice for similarity measure. However, in order to better qualify the choice of similarity measure, actual rating data from Wikipedia is required.

7.8 Caching similarity scores

At the moment all similarity scores are computed on every article view. In order to improve the performance of the system, these scores could possibly be computed when required and then be cached in the system for easy retrieval.

Cached similarity scores would then be invalidated when the active user rates an article and thus potentially changes the similarity with the current trustees and/or add new trustees, but the scores would also be invalidated if any other user, trustee or not, rates an article that the trustor has already rated.

Experiments could be carried out to determine when the similarity scores should be computed and cached for optimal performance. There are a few things worth considering in regards to this:

- If similarity scores are only cached when the trustor rates an article, the system potentially loses out on ratings that other users have set in the time between caching and viewing an article.
- If they are calculated and cached for all users that are affected by one user rating an article, they might be calculated unnecessarily often, potentially leading to worse performance than when compared to not caching at all.

The optimal solution is most likely somewhere in between these two.

8 Conclusion

The purpose of this thesis was to integrate solutions to the cold start problem of Wikipedia Recommender System (WRS) and to make the two previous versions of the system converge.

This was done by thoroughly analyzing the theoretical foundation and the implementations of the previous systems, in order to determine what parts of each system should be included into the resulting version of WRS.

In the early versions of WRS the system was based on a decentralized solution that stored rating and trust data at the client. It later changed to store rating data on a central server, but the theory behind the trust model remained the same with limited changes. We found that it would be beneficial to switch the theoretical foundation from a trust model to a measure for statistical correlation, now that the rating data for all users was available to WRS.

We analyzed a number of different techniques for calculating similarity based on statistical correlation and we gave a detailed description of Pearson product-moment correlation coefficient because this technique is frequently mentioned in scientific literature concerning the topic of collaborative filtering. However, we found that it would not be optimal within the realm of Wikipedia, where we believe that it is unlikely that users have rated a high number of articles in common and because Pearson correlation only provides a useful coefficient when the dataset is of size 3 at the very least.

Following our analysis of Pearson correlation, we found that a modified version of Squared Euclidean Distance would provide better results for WRS, given that it is able to compute meaningful similarity scores even for sparse data sets.

Through our analysis of the existing versions of WRS we found that the version of WRS by Mihai Mihăilă [Mihăilă, 2011] served as a better foundation for our continued work on WRS, as the technologies used within this system were superior to other versions of WRS on both the client and the server side.

With the goal of implementation of similarity propagation in WRS in mind, we analyzed various algorithms for trust propagation and the applicability of these algorithms within WRS. We found that MergeTrust could be modified to work in our setting, and that scientific literature also indicated that this algorithm would perform better than the other candidates.

In order to achieve an efficient implementation of similarity propagation using MergeTrust, we designed a number of data structures that could store the rating data in memory in such a way that predicted ratings could be calculated efficiently and with minimal overhead. The data structures were analyzed in terms of both time and space complexity using Big O notation.

The WRS backend was rebuilt in order to support similarity propagation alongside potential contributions from WikiTrust and to be robust in terms of the number of user interactions and the availability of the WikiTrust API. We also made sure the system provided the WRS functionality through a RESTful web service that followed the best practices for this type of system.

The system as a whole has been thoroughly tested, and a number of automated unit tests of the core functionality of the system have been implemented. These tests serve two purposes:

1. Asserting that the current version of WRS works as expected
2. Eases the implementation of future improvements, as the tests reduce the likelihood of breaking existing core functionality

8.1 Results

The result of this thesis is a version of WRS which includes both WikiTrust and similarity propagation to mitigate the cold start problem. We sum up our main contributions to WRS as follows:

1. Mitigated the cold start problem by combining similarity propagation and data from WikiTrust.
2. Implemented in-memory data structures to efficiently calculate the predicted ratings, in order to meet the new demands in terms of increased number of database queries. This was necessary because a high number of ratings may need to be considered when similarity propagation is used.
3. Switched to using statistical correlation instead of the trust model to calculate similarity between users.
4. Re-factored the backend in order to remove obsolete functionality and properly implement a RESTful API.
5. Added a full suite of unit tests for the backend functionality.

The resulting version should be regarded as a prototype because it has been implemented with a minimum of features, and because the system can only be used in the Chrome browser. However, the prototype is ready to be deployed to an external test server and as such the system is ready to be used in a test and evaluation phase.

8.2 Future work

We believe that this thesis provides a solid foundation for the continued work on WRS because we have thoroughly discussed and described our design process, and we have highlighted the values and choices that have not been based on experimental data, such that this could perhaps be the basis of a new and improved version of WRS.

It is our opinion that in order to further improve on WRS it would be beneficial to let the system run in an external test phase for a while to gather usage data. This data will indicate if there are specific parts of the system which need re-modeling in order to be able to predict more accurate ratings. Actual data will make it possible to question the assumptions and values we have chosen throughout the implementation and by using a proper dataset, it will be possible to carry out extensive tweaking of these values.

Based on a real dataset it could also be beneficial to test the system with different similarity metrics or propagation algorithms and possibly even allow users to choose for themselves on a per user basis.

The extension could be improved with common functionality such as allowing the user to change password, retrieve a lost password or simply resetting it to some randomly generated temporary password. It could also allow a user to rerate an article that has undergone significant changes, such that outdated ratings can be replaced with a more current one.

Feedback from users, possibly in the form of usability testing could also help in improving the system, by addressing new desired functionality or rework of the current extension.

9 Bibliography

[Korsgaard, 2007] Thomas R. Korsgaard. *Improving Trust in the Wikipedia*. MSc Thesis: Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2007.

[Lefevre, 2009] Thomas Lefevre. *Extending the Wikipedia Recommender System*. MSc Thesis: Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2009.

[Pilkauskas, 2010] Povilas Pilkauskas. *Expertise classification of recommenders in the Wikipedia Recommender System*. MSc Thesis: Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2010.

[Mihăilă, 2011] Mihai Mihăilă. *Addressing the Cold Start Problem in the Wikipedia Recommender System through Content-Based Filtering.* MSc Thesis: Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2011.

[Andersen, 2011] Natasa P. Andersen. *Reducing Cold Start problem in the Wikipedia Recommender System*. MSc Thesis: Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2011.

[Marsh, 1994] Stephen P. Marsh. *Formalizing Trust as a Computation Concept*. PhD dissertation: Department of Computing Science and Mathematics University of Stirling, 1994.

[Jonker, Treur, 1999] Catholijn M. Jonker and Jan Treur. *Formal analysis of models for the dynamics of trust based on experiences.* In *MAAMAW '99: proceedings of Modelling Autonomous Agents in a Multi-Agent World*, 1999.

[Avesani. Massa, Tiella, 2005] Paolo Avesani, Paolo Massa and Roberto Tiella. A trust-enhanced recommender system application: Moleskiing. In *SAC '05: proceedings of the 2005 ACM symposium on Applied computing*, 2005.

[Avesani, Massa, 2007] Paolo Avesani and Paolo Massa. *Trust metrics on controversial users: balancing between tyranny of the majority and echo chambers.* In *IJSWIS '07: International Journal on Semantic Web and Information Systems*, 2007.

[Victor, 2010] Patricia Victor. *Trust Networks for Recommender Systems*. PhD Dissertation: Faculty of Sciences, Ghent University, 2010.

[Jamali, 2010] Mohsen Jamali. *A Distributed Method for Trust-Aware Recommendation in Social Networks.* In *CoRR abs/1011.2245*, 2010.

[Breese et al., 1998] John S. Breese, David Heckerman and Carl Kadie. *Empirical Analysis of Predictive Algorithms for Collaborative Filtering*. In *UAI '98: Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, 1998.

[Singla, Richardson, 2008] Parag Singla and Matthew Richardson. *Yes, There is a Correlation - From Social Networks to Personal Behavior on the Web*. In *WWW '08 Proceedings of the 17th international conference on World Wide Web*, 2008.

[Wan, Chen, 2011] Yu-Hao Wan and Chien Chin Chen. *An effective cold start recommendation method using web of trust*. In PACIS '11: Proceedings of the Pacific Asia Conference on Information Systems, 2011.

[Feld, 1991] Scott S. Feld. *Why your friends have more friends than you do.* In *American Journal of Sociology Vol. 96, No. 6*, 1991.

[Coleman, 1961] James S. Coleman. *The Adolescent Society. Greenwood Press*, 1961.

[Tosun, Sheppard, 2011] Hasari Tosun and John W. Sheppard. *Incorporating Evidence into Trust Propagation Models Using Markov Random Fields*. In PerCom '11: Proceedings of the ninth annual IEEE International Conference on Pervasive Computing and Communications, 2011.

[Victor et al., 2008] Patricia Victor, Chris Cornelis, Ankur M. Teredesai and Martine de Cock. *Whom Should I Trust? The Impact of Key Figures on Cold Start Recommendations*. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, 2008.

[Adler et al., 2008] B. T. Adler, K. Chatterjee, L. de Alfaro, M. Faella, I. Pye and V. Raman. *Assigning Trust to Wikipedia Content*. In *WikiSym '08: Proceedings of the 4th International Symposium on Wikis*, 2008.

[Booth, 2007] Michael Booth. *Grading Wikipedia*. On http://www.denverpost.com/search/ci_5786064, 2007. (Accessed Feb 27, 2013)

[Guha et al., 2004] R. Guha, R. Kumar, P. Raghavan, and A. Tomkins. *Propagation of trust and distrust*. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, 2004

[Alani et al., 2004] H. Alani, Y. Kalfoglou and N. Shadbolt. *Trust strategies for the semantic web*. In *ISWC '04: Proceedings of the Trust, Security and Reputation Workshop at the ISWC04*, 2004

[Huang, Fox, 2006] J. Huang and M. S. Fox. An o*ntology of trust: formal semantics and transitivity*. In *ICEC '06: Proceedings of the 8th International Conference on Electronic Commerce*, 2006

[Ousterhout et al., 2009] John Ousterhout et al. *The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. In newsletter '09: ACM SIGOPS Operating Systems Review*, 2009.

[Sarwar et al., 2001] Badrul Sarwar, George Karypis, Joseph Konstan and John Riedl. *Item-Based Collaborative Filtering Recommendation Algorithms*. In *WWW '01 Proceedings of the 10th international conference on World Wide Web*, 2001.

[Montaner, López, 2002] M. Montaner, B. López and J. L. de la Rosa. *Opinion-based filtering through trust*. In *CIA '02: Proceedings of the 6th International Workshop on Cooperative Information Agents*, 2002

[Montaner, López, 2003] M. Montaner, B. López and J. L. de la Rosa. *A taxonomy of recommender agents on the internet*. In *Artificial Intelligence Review 19, 4*, 2003

[Guo et al., 2012] Guibing Guo, Jie Zhang and Daniel Thalmann. *A Simple but Effective Method to Incorporate Trusted Neighbors in Recommender Systems and Error Correction for "A Simple but Effective Method to Incorporate Trusted Neighbors in Recommender Systems*". In *UMAP '12: Proceedings of 20th International Conference on User Modeling, Adaptation and Personalization*, 2012.

[Richardson et al., 2003] M. Richardson, R. Agrawal and P. Domingos. *Trust management for the semantic web*. In *ISWC '03: Proceedings of the Second International Semantic Web Conference*, 2003

[Castelfranchi, Falcone, 2001] C. Castelfranchi and R. Falcone. *Social trust: a cognitive approach*. In *Trust and deception in virtual societies*, Kluwer Academic Publishers, 2001

[Ziegler, Golbeck, 2005] Cai-Nicolas Ziegler and Jennifer Golbeck. In*vestigating correlations of trust and interest similarity - Do birds of a feather really flock together?* In *Decision Support Systems*, 2005.

[Abdul-Rahman, Hailes, 2000] A. Abdul-Rahman and S. Hailes. *Supporting trust in virtual communities*. In *HICSS '00: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, 2000

[Emigh, Herring, 2005] W Emigh and S. Herring. Collaborative authoring on the Web. In HSCC '05: Proceedings of the conference on Hybrid Systems: Computation and Control, 2005

10 Appendix

10.1 Pearson correlation and Squared Euclidian distance coefficient for small data sets

Pearson correlation and Squared Euclidian distance coefficient were calculated for all different permutations of sets of size 2 to 6 where the sets contain integer values in the range [1,9]. The actual results are found on the attached CD, where the .txt files are named after the size of the permutation sets:

- `2.txt`
- `3.txt`
- `4.txt`
- `5.txt`
- `6.txt`

10.2 WRS Source code

All WRS source code is found in the attached CD-ROM.

The source code for WRS server application is found in the folder “WRS server source”. A full Netbeans project containing the source files and compiled files are available in the zip-file “Netbeans WRS project.zip”.

The source of the Chrome extension is found in the folder “WRS-extension”.

10.3 Deploying the WRS backend

In the following guide we assume a clean installation of the Debian based Linux distribution Ubuntu (possibly in a virtual machine). Other distributions may require other steps.

1. Make sure the server software is up-to-date by executing the following commands:
    a. `sudo apt-get update`
    b. `sudo apt-get upgrade`
2. Install JDK:
    a. `sudo apt-get install openjdk-7-jdk`
3. Install mysql (Be sure to pick the same user password as the one which is defined in the WRS project or create a specific WRS user after the installation is done):
    a. `sudo apt-get install mysql-server`
4. Log in to the mysql server by executing (substituting `root` and `123456` for the proper credentials):
    a. `mysql --user=root --password=123456`
5. Run WRS SQL scripts on the server by executing the following, where `/absolute/path/to/sql/script` is replaced with the path to each SQL script.
    a. `source /absolute/path/to/sql/script`
6. Download Glassfish application server from: http://glassfish.java.net/public/downloadsindex.html - Download the file next to the text “Not sure which version to use? Try the simple Zip archive”. This will give a ZIP file.
7. Unzip the downloaded file from the home directory:
    a. `unzip Downloads/glassfish-3.1.2.2.zip`
8. Download the MySQL connector from the following URL: http://dev.mysql.com/downloads/connector/j/
9. Unzip the connecter and copy the file mysql-connector-java-bin.jar to the folder `~/glassfish3/glassfish/lib`
10. Start the Glassfish server using a default domain by going to the Glassfish directory and executing:
    a. `bin/asadmin start-domain`
11. Deploy the web application by copying the war file to: `~/glassfish3/glassfish/domains/domain1/autodeploy`
12. Test that it is running correctly by checking the response from:
    a. http://localhost:8080/wrs-webapp/wrs?method=getCategories