

MSc Computer Science
Department of Computer Science and Information Systems
Birkbeck, University of London

Project

A Recommendation Engine

Amy Peters

This project is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

Abstract

I proposed to implement a recommendation system, using collaborative filtering techniques, for the interactive online community *Deep Underground Poetry*. Using data which has been collected from members, I was able to compute a list of suggested reading which was personalised to each user, and also provide them with details of members similar to them (either in preferences, or writing style). The output produced by the recommendation engine has the capacity to adapt to new information and changes in member data in order to give users up-to-date, plentiful and accurate recommendations. The overall aim of the system was to develop the means to promote interaction between members by increasing the number of views, ratings and comments each poem received, by guiding users towards poems which they should enjoy. The system also promotes social networking, by supporting the idea that the website caters for a member's own personal tastes, and by connecting them to other members with similar interests.

September 2011

Introduction

1.1 Building an Online Community	Page 1
1.2 Why Collaborative Filtering?	Page 2
1.3 Aims and Objectives	Page 2
1.4 Overview	Page 2

Background

2.1 The Netflix Prize	page 3
2.2 Collaborative Filtering Techniques	page 3
2.2.1 The Neighbourhood Approach (k -NN)	page 3
2.2.2 Singular Value Decomposition (SVD)	page 4
2.2.3 Blending Approaches	page 5

Analysis, Requirements, and Design

3.1 Technical Feasibility	page 6
3.2 Functional Requirements	page 6
3.3 Non-Functional Requirements	page 7
3.4 The Data Set	page 7
3.4.1 Onsite Initiatives to Increase Available Data	page 10
3.5 The Existing System	page 11
3.6 System Architecture	page 12
3.5.1 Activity Diagram (for on site user interaction)	page 13
3.5.2 Class Diagram (offline tool)	page 13
3.5.3 Entity-relationship models of MySQL tables	page 14

Development

4.1 Collecting Data	page 16
4.2 Utilizing Implicit Data	page 17
4.2.1 Comment Analysis	page 17
4.2.2 Followers	page 19
4.3 Implementing k-NN	page 22
4.3.1 User-user Matrix	page 23
4.3.2 Item-Item Matrix	page 23
4.3.3 Distance Metrics	page 25
4.4 Implementing SVD	page 29
4.4.1 Tuning the algorithm	page 30
4.4.2 A closer look at factors	page 34
4.5 Blending: Linear Blend	page 34
4.6 Blending: Smart Blend	page 35
4.6.1 Adjusting the Metrics	page 35
4.7 Program Design Decisions	page 39
4.8 Generating Data from Predictions	page 39
4.8.1 Recommended Poems (User-Item)	page 39
4.8.2 Similar Poems (Item-Item)	page 40
4.9 Website Implementation	page 41
4.9.1 Back-end Data Usage	page 41
4.9.2 Front-end for User Test Phase	page 45
4.9.3 Front-end Final	page 46

Conclusion

5.1 Evaluation of Objectives	page 47
5.2 Statistical Evaluation	page 48
5.3 User Opinion	page 49
5.4 Lessons Learned	page 51
5.5 Future Development	page 52

Introduction

1.1 Building an Online Community

In August 2009 I re-launched an online creative writing community called *Deep Underground Poetry (DU Poetry)* [1], which I originally started in 1999. For ten years, the website had been offline and the Internet had changed enormously during this time. However, through re-establishing *DU Poetry* over the last few years, it became apparent that one of the key challenges still remained; finding ways to encourage people to read and comment on other peoples' submissions. Giving members the feedback and attention they crave is a key part of creating a thriving community and retaining an active membership. Discussions related to this issue frequently crop up on the *DU Poetry* "suggestions" forum. In addition, a homepage poll showed that feedback from others was important to more than half of respondents [2].

There are a number of ways in which online poetry communities have tried to increase commenting levels. Some websites force their members to make comments, so as to artificially increase the number of comments made. For example, *GS Poetry* [3] requires members to gain credits by making comments before they are allowed to "showcase" a poem. *GotPoetry* [4] implements a "karma" system, to stop less active members from rating poems without leaving a comment. *DU Poetry* offers incentives, like extra exposure for their poems, to "top-critiquers" (members who make lots of useful comments). Despite these efforts, the average number of comments that each poem receives still remains relatively low; currently less than 1.5 comments per poem on *DU Poetry*. In addition, the risk of enforcing measures is that the quality, usefulness and comprehensiveness of the feedback diminishes. Keen to explore more organic ways to increase the number of comments; I created another *DU Poetry* poll asking whether people were more likely to comment on poems they liked, or poems they disliked [5]. To date, 100% of respondents have answered that they more often comment on poems that they like. My challenge was to find a way for people to more easily and quickly discover poems that they might actually want to read and comment on. I also hoped to address the issue of maintaining exposure for older submissions, which no longer appear on the first few pages of listings, and can therefore go undiscovered by new members.

Another key way in which I hoped my system could improve *DU Poetry* was to discover patterns in order to group similar members together. I compete with numerous other established creative writing websites, so it's important that I continue to develop *DU Poetry* into a strong community, making use of current trends in social networking.

1.2 Why Collaborative Filtering?

Developments in technology (hardware, software and connection speeds) have made it possible to collect vast amounts of data from users and with this has come the increased use of collaborative filtering (CF). Recommendation engines use CF as a method of making automatic predictions about which items will suit a user (filtering), by collecting information about the past actions of many users (collaboration). Unlike conventional media like newspapers and television, where there are a few editors controlling output, the CF model can have infinitely many editors and improves as the number of participants increases [6]. CF techniques don't rely on specific domain knowledge and can be applied to sparse data sets; thereby avoiding the need for comprehensive data collection (it should not be necessary for a user to evaluate a vast number of items).

Many successful websites use collaborative filtering techniques to implement some form of recommendation engine. *Amazon*'s recommender system suggests products to customers based on their past purchases and the items they currently have in their shopping basket. When Greg Linden, the creator of *Amazon*'s recommendation engine, left in 2002 he reported that over 20% of sales came from personalized recommendations, the figure is estimated to be even higher today [7].

I applied similar techniques in order to provide content to *DU Poetry* users which I hoped would be of interest to them, in order to enrich their experience and encourage them to spend time on the website. I wanted to reinforce the idea that the website is for "people like them", by recommending poems which appeal to them and helping them find other members who are similar to them.

1.3 Aims and Objectives

My key objectives were:

- Tune an algorithm to give quantitatively accurate recommendations.
- Provide members with recommendations which they felt were well suited to their tastes.
- Look for patterns in users' tastes in relation to characteristics like their age and location.
- Build an efficient and scalable system which could react instantly to new information.
- Develop the system in response to implicit and explicit feedback from users.

1.5 Overview

Chapter two describes the event which was the catalyst for a large amount of research and development in the area of collaborative filtering in recent years; the *Netflix Prize*. Chapter three analyses the current *DU Poetry* system and data set, and outlines my initial analysis and plan for the system. Chapter four details all aspects of development, testing and tuning. Finally, chapter five discusses my findings and evaluates the work I have done, along with considering future development.

Background

2.1 The Netflix Prize

The Netflix Prize was a competition to find a collaborative filtering algorithm to substantially improve the accuracy of predicting how much someone is going to enjoy a movie, based on their existing movie preferences (1-5 value ratings they have given) [8]. The competition launched on October 2nd 2006 and the winning team was declared on September 18th 2009.

Each participating team was required to produce an algorithm to predict grades for a given set, but they were only informed of the score for half of the data, while the other half was used by the jury to determine potential prize winners. Submitted predictions were scored against the true ratings in terms of root mean square error (RMSE) and the goal was to reduce this error as much as possible. Prizes were based on the improvement achieved over *Netflix's* own algorithm, called *Cinematch*. In order to win the prize of 1 million dollars, a team had to successfully improve upon *Cinematch* by more than 10%. The fact that there was such a substantial prize on offer illustrates how important *Netflix* deemed their recommendation engine to be.

The collaborative filtering approach to recommenders has gained much popularity as a result of the *Netflix Prize* competition [9]. A substantial number of papers and new techniques were developed during, and as a direct result of, the competition and the sharing of ideas which took place.

Unfortunately the data sets used in the *Netflix Prize* are no longer available, and plans to run a follow up competition have also been abandoned, owing to data privacy concerns [10].

2.2 Collaborative Filtering Techniques

I have limited the scope of my project to focus in-depth on two of the most common methods of producing recommendations by collaborative filtering. They are inherently different approaches, one falls into the category of being memory-based (the neighbourhood approach) and the other is model-based. I also look at ways of blending the results of the two approaches.

2.2.1 The Neighbourhood Approach (k -NN)

Neighbourhood based models (memory-based) are traditionally the most common approach to CF [11]. They use some form of the k -nearest neighbours algorithm (k -NN) to compute the relationship between items, or between users. k -NN is a simple machine learning algorithm where an item is classified by the average of its k nearest neighbours, where k is a positive integer; for example, if $k = 1$ then the unknown rating would get the value given by the most similar user who has rated that item. The best value for k is application specific and can only be found by experimenting with the data set. It is typical for a weighted average to be used, with the closest (most similar) neighbours contributing more. To work out the weighing and which are the closest neighbours, a distance metric is used. The main challenge for using k -NN in collaborative filtering is the computational cost and time involved in computing the neighbours.

2.2.2 Singular Value Decomposition (SVD)

The model-based approach using the linear algebra technique of singular value decomposition (SVD) has recently become popular for collaborative filtering due to its accuracy and scalability [11, 12]. Rather than directly setting out to make predictions, instead they use machine learning to characterise the data and uncover the underlying causes (factors) that work in combination to describe the data set [13].

The idea is to decrease the amount of computation required by reducing the size of the matrix. Instead of comparing every object (item or user) to every other object, instead you define each object by a number of factors. For example, when the items are poems, factors which might emerge may be related to how funny or sad a poem is. Rather than having to manually define these characteristics, the algorithm finds patterns in the existing data and discovers the generalisations (also called eigenvectors).

Imagine a data set of 3000 users and 25000 items; this would involve $3000 \times 25000 = 75$ million computations. However, if each object had just 40 factors, that could be used to compare it to other objects, then it would only require $40 \times (3000 + 25000) = 1.12$ million computations [14]. Given a $m \times n$ matrix A with rank r (the number of factors which we want to define items by), the singular value decomposition $SVD(A)$ is defined as:

$$SVD(A) = U \times S \times V^T$$

Matrix S is a diagonal matrix called the *singular* having only r nonzero entries ($r \times r$), which means the effective dimensions of the orthogonal matrices U and V are $m \times r$ and $r \times n$ respectively. The first r columns of U and V represent the eigenvectors associated with the r factors of AA^T and A^TA . For the matrix A it can be said that the columns corresponding to the r values span the column space in U and the row space in V . The following illustration represents this on a very small scale:

$$\begin{array}{c} \text{factor 1} \\ \text{(sad)} \end{array} \quad \begin{array}{c} \text{factor 2} \\ \text{(funny)} \end{array} \\ \begin{matrix} \text{Poem 1} & \left(\begin{array}{cc} 0 & 3 \\ 3 & 2 \\ 4 & 1 \end{array} \right) \\ \text{Poem 2} & \times \end{matrix} \quad \begin{matrix} \text{User 1} & \left(\begin{array}{cc} 2 & 0 \\ 1 & 3 \end{array} \right) \\ \text{User 2} & \end{matrix} = \left(\begin{array}{cc} 0x2 + 3x1 & 0x0 + 3x3 \\ 3x2 + 2x1 & 3x0 + 2x3 \\ 4x2 + 1x1 & 4x0 + 1x3 \end{array} \right) = \begin{matrix} \text{Poem 1} & \left(\begin{array}{cc} 3 & 9 \\ 8 & 6 \\ 9 & 3 \end{array} \right) \\ \text{Poem 2} & \\ \text{Poem 3} & \end{matrix} \quad \begin{matrix} \text{User 1} & \left(\begin{array}{cc} 3 & 9 \\ 8 & 6 \\ 9 & 3 \end{array} \right) \\ \text{User 2} & \end{matrix}$$

For this example (using an r value of 2) the algorithm has identified the two most prevalent characteristics by which to describe users and poems (sad and funny). Each poem is given a value for each characteristic and every user has a value representing how much they like that characteristic. By multiplying the two matrices together we get a representation of the original matrix (which is a “marks out of ten” rating made by each user for each poem). We can see that User 2 has given Poem 1 a high mark and this was factorised into both being given a high value for the factor 2 characteristic “funny”.

One possible advantage of the *low-rank* approximation resulting from SVD is that the resulting data may in fact be better than the original full matrix, as the process may actually filter out some unwanted “noise” from the data [15]. However, the opposite may also be true, especially when applying SVD to collaborative filtering, due to the high proportion of missing data. Addressing only the known values is prone to over-fitting (where there aren’t enough values to consider, so a random error or “noise” value may be given undue influence over results [16]). Variations of SVD have been developed to try and address this problem [17]. One notable example is that of incremental SVD, devised by Simon Funk during the “Netflix Prize” competition [14]. He incorporates a technique for regularising the model in order to identify abnormal values and avoid over-fitting. His technique was widely used by other participants in the competition, and I too used the information he provided as a basis for my own SVD algorithm.

2.2.3 Blending Approaches

Along with optimising each of them individually, I will be looking at different ways to blend my results. When the *Bellkor* team won their progress award during the *Netflix Prize* competition in 2007, their solution consisted of blending the results of 107 different algorithms [18]. Although what they achieved is not within the scope of my project, I have taken on board their findings; which clearly indicate the merits of blending different approaches, instead of just refining a single technique. I am going to look at linear blending, which involves finding the best ratio with which to blend the results produced by different algorithms. I also go on to discover another way of blending the two algorithms, which proved to be faster and more accurate, which I called “Smart Blend”.

Analysis, Requirements, and Design

3.1 Technical Feasibility

The key to the success of *Amazon*'s recommendation engine is that the computationally expensive task of creating the "similar items" table takes place offline. The online part (looking up similar items for a particular user) scales independently of the size of the overall data set, so the system remains fast despite the huge amount of data involved [19]. By producing results in terms of similar items, rather than storing tailor made predictions for each user, *Amazon* is able to respond immediately to new information from users, without having to re-compute recommendations. Similarly, when teams were testing algorithms for the *Netflix Prize*, it would typically take several hours to generate each set of predictions; so it's not feasible to perform these tasks online.

One of my challenges was to give the impression that predictions were generated there and then, whilst actually doing the computationally heavy processing offline. This was important not least because *DU Poetry* resides on shared web hosting and therefore has limited access to the web server's resources. It also ensures that I can deal with a growing data set, without impacting upon the online speed observed by visitors of the website. I chose to write my offline tool in C++ as it gave me more freedom to explore opportunities for optimisation at low-level.

3.2 Functional Requirements

A key requirement was to test a variety of methods and variables to yield the most accurate results. This was done using a predictive accuracy metric, which measures how close the systems predicted ratings are to the true user ratings. My primary test for gauging the accuracy of results was to find the root mean squared error (RMSE). This was the metric used for evaluating entries in the *Netflix Prize* Competition. I also monitored the mean absolute error (MAE), for which the results were consistently well correlated to the RMSE. Both of these metrics square the error before summing it, the result of which is more emphasis on large errors. For example, an error of one point increases the sum by one, but an error of two points increases the sum by four [20]. This amplification of the error contribution made by false positives and false negatives was desirable as it helped avoid the kind of very inaccurate results which could lead to inappropriate recommendations [21]. The ratings data was divided into two sub-sets; the "training" sub-set was used to make predictions for the data contained in the "probe" sub-set. The actual ratings values in the probe sub-set were compared to the predictions and the difference (error) computed in order to evaluate the accuracy of the algorithm being tested.

Another key topic for analysis was how best to use the data I had available. This included looking at the merits of using implicit data, to broaden the amount of information being used. Although implicit data may introduce noise into the data set, it may also be the case that explicit data is prone to noise, because members use on-site features in different ways. One of the interesting aspects of implicit data collection was that it allowed members' behaviour to be mapped, without relying on them to carry out specific tasks and actions. Using implicit data also helped alleviate some of the issues surrounding the sparseness of the data, by enabling me to add to the pool of "known data". The kind of data available, and how it could be used most effectively, was what really set my task apart from other recommendation systems. Although I could look at how similar projects had been approached, to a certain degree my methods were unique to the *DU Poetry* data set and the problem I was trying to solve.

Another important requirement was that my offline tool produced results which could be used to display predictions online. I looked at different ways of generating results; either by creating user-specific predictions, or data about similarities between members and poems. The latter method would allow me to make predictions for a new user right from the first action they take (for example, a “like”), even if they had joined since the offline tool was last run.

3.3 Non-functional Requirements

I looked at ways to optimise the performance of the offline tool in terms of time taken to produce results and also memory usage. I output clock times for each stage of program execution in order to monitor how long different procedures took. This helped me to spot ineffective design decisions during development, which caused the program to run slower than expected. It also affected the tuning process; if an unacceptably long computation cost was incurred for a very small improvement in error, then I choose to use the faster option, as it allowed me to run the offline tool more often.

I produced a user-friendly front end for members, which fit in with the existing look of the *DU Poetry* website. These pages were set into the existing website template. For this reason, the online programming aspect of this project was written in *Perl*; the same language as the rest of the website.

Throughout development I generated lots of statistics by running *SQL* queries to the on-site database (often via the command line). I consistently monitored and tuned the performance of these queries, along with those for collecting existing data and for inserting, sorting and selecting recommendations from predictions generated by the offline tool.

3.4 The Data Set

Poems	25881
Poems with ratings (like or reading list entry)	11818
Poems with comments	16624
Members (not including sign ups who never logged in)	3801
Members with rating data (likes or reading list entries)	1475

Like any website starting out on a new domain name (without a budget for promotion) it has took *DU Poetry* time to gain popularity and establish reasonable search engine rankings. The amount of new members signing up continues to increase month on month, the most recent figure discussed in my proposal was 279 new members for March of 2011. Since then, during the months of April to August 2011, there have been over 350 new sign ups each month. The number of active members; those logging in within the last two months, has also grown by a third from approximately 1000 to 1350. An overview of key statistics, taken on 14th September 2011, can be viewed below.

I was always keen to avoid a “marks out of five” style ratings system for poems, because it didn’t fit well with the creative nature of the content. There is also strong evidence that five-star ratings systems don’t work. *YouTube* abandoned its five star ratings system, switching instead to a simpler “Like / Don’t Like” model, because almost all videos were receiving the maximum (five stars) rating from users [22]. I mentioned in the table above, a number of the features which I implemented in order to gauge a member’s reaction to a poem. These include an anonymous “like this?” option and a personal reading list, to which a member can add their favourite poems. In addition, a member can “follow” another member poet, this means they receive a message in their updates feed each time that member makes a new submission. Poem views are also logged, in order to keep track of which members have viewed which poems. Below is a summary of each of these features, along with usage statistics (from the website’s MySQL database, queried on 14th September 2011):

Feature	Launch Date	Quantity
Reading List Entries	30 th	5965
Poem Views*	11 th	147016
Followers	11 th	5794
Likes**	25 th	18015
Comments***	Site Launch	39513

* I have only included poem views made by members, anonymous visitors are not included.

** “Likes” which also have an associated reading list entry are not included; a reading list entry takes precedence over a “like”.

*** Comments made by the poem author are not included.

A variety of data was used in the recommendation engine; a viewed poem (with no action taken) was counted as a rating of zero, a “like” counted as a one and a reading list entry counted as a two. Including the data on poem views was also useful for determining which poems a user had already looked at, so as not to recommend them poems which they had already read.

Further data was used to “simulate” poem ratings; this was the more subtle explicit data like the authors a member “follows” (their favourite authors). The statistics show that a

favourable action is more likely on a poem written by a followed author. I investigated the effects of assuming a “like” on these poems.

	Un-liked	Liked	Reading List
Followed Author	24061 (63.5%)	11611 (30.6%)	2217 (5.9%)
Un-followed Author	55670 (84.6%)	8884 (13.5%)	1219 (1.9%)

The preceding data snapshot was taken for the period after the “like” feature launched. It shows that over twice as many likes are received on poems by authors the reader was “following”, so we could conclude that theoretically it would be better to count this data (assume a “like” when author is being followed). However, an action is still only taken 36.5% of the time, so this ran the risk of introducing noise into the data set, and skewing the results too heavily in favour of poems by “followed” authors.

I also looked at how I could use comment data to gauge a member’s reaction to a poem, particularly for the period before the “like” feature was launched, where poem comments may be the best indication of whether a member liked a poem or not. The statistics show the behavioural patterns of “likes” and followed authors, in the context of comments made (for the post “like” feature period). A comment is left 19% of the time when a poem is viewed by a member, but this occurs much more frequently when a poem is “liked”.

	Liked Poem	Un-liked Poem	Followed Author	Un-followed Author
Comment Left (19% of the time)	13653 69%	6134 31%	10370 52%	9417 48%
No Comment Left (81% of the time)	7898 9%	75580 91%	26806 32%	56576 68%

I also explored the relevance of zero rated data; members with only zero value ratings (viewed poems but never liked them, or added them to their reading list) and poems with only zeros (viewed, but no actions taken). I set out to assess whether this type of data provided a useful contribution, since it couldn’t directly lead to poems being recommended; however, it had the capacity to help link poems and members together, and find patterns during the training process. The following data illustrates the quantity of zero rated data, i.e. where views exists (the poem has been viewed at least once, or the member has viewed at least one poem) but no actions have been taken:

Members with no likes or reading list entries.	1324
Members with no likes, reading list entries, followed authors or comments made.	520
Poems with no likes or reading list entries.	12961
Poems with no likes, reading list entries, author followers or comments left.	1321

The statistics above supported the case for looking closely at data related to comments and followers, in order to increase the number of members and poems for which there was non-zero value data available. If we were only looking at “likes” and reading list entries, there would be nearly 13000 poems which would not stand a chance of being recommended, this reduces by up to 90% when taking into account data for “followers” and comments.

My data set presented many challenges, not least the sparseness of the data. Below I have compared the *DU Poetry* training set data (assuming the probe set contains 10,000 entries) with the *Netflix* training set, which was provided to participants of the *Netflix Prize* competition.

	Known Ratings in Training Set	Movies / Poems	Users / Members	Sparseness
Netflix	100,480,507	17,770	480,189	98.8%
DU Poetry (no zero only value ratings)	105,601	17,892	1638	99.6%
DU Poetry (all ratings)	147,194	25,579	2880	99.8%
DU Poetry (all ratings including un-viewed poem follower data)	587,007	26,564	2899	99.2%

The statistics above illustrate the problem that collaborative filtering attempts to overcome; trying to “fill in the gaps” for the overwhelming majority of unknown information. In each of the cases considered (with different types of data included) the *DU Poetry* training sets all had a slightly larger percentage of unknown ratings than that of the *Netflix Prize* training set. This made my task particularly challenging.

3.4.1 Onsite Initiatives to Increase Available Data

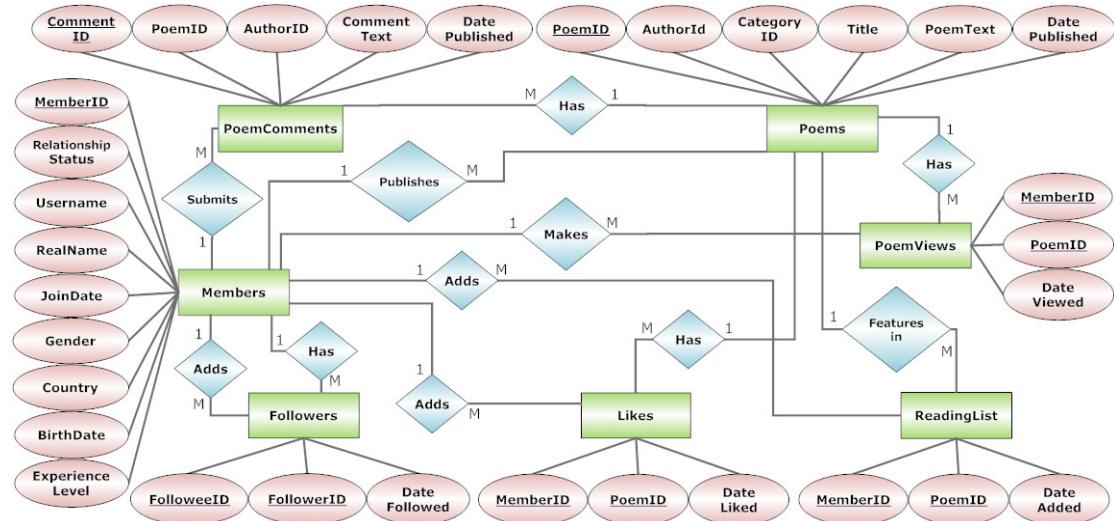
In order to increase the amount of known ratings, I ran two “poem reading challenges”. I incentivised members to get involved by using the existing trophy system that is used to reward winners of member led poetry competitions. For each challenge they complete, they are awarded a trophy which appears on their profile and increases the trophy count which appears under their avatar image, across the *DU Poetry* discussion forums.

The first challenge was called “DU Archives” and generated random poems, via a link, which were published prior to the “like” feature being launched. The second initiative was called “Undiscovered” and generated a link to a random poem which had been viewed by less than 5 other members. My aim with both challenges was to increase the amount of data relating to potentially overlooked poems, to try and ensure that as many poems as possible could be included in the recommendations made. One of the main aims of the recommendation engine was to let members know about poems they may never have otherwise found out about. New members are less likely to know about poems posted before they joined, so it was crucially important that there was data available for all poems throughout the lifetime of *DU Poetry*, in order to connect them to members who might enjoy reading them. Both challenges involved reading a designated minimum number of poems (75 and 100 respectively). In order to help ensure that people didn’t abuse the system, by quickly refreshing the links, I used a timed logging system to discount views made less than 10 seconds apart. The table below shows the extra data generated as a direct result of the poem reading challenges (queried 14th September 2011):

Challenge Name	Poem Views (unique poems)	Likes	Reading List Entries	Comments
DU Archives	2318 (1801)	452	25	155
Undiscovered	1959 (1843)	426	22	193

3.5 The Existing System

The following diagram shows relevant database tables and columns from the existing system, and how they are related to each other.



An entity-relationship model using Chen notation, showing relevant tables and attributes in the existing *DU Poetry MySQL* database.

3.6 System Architecture

The offline C++ tool takes the known ratings output from the on-site database, training the data to produce output files ready to upload back into the database. Main features include:

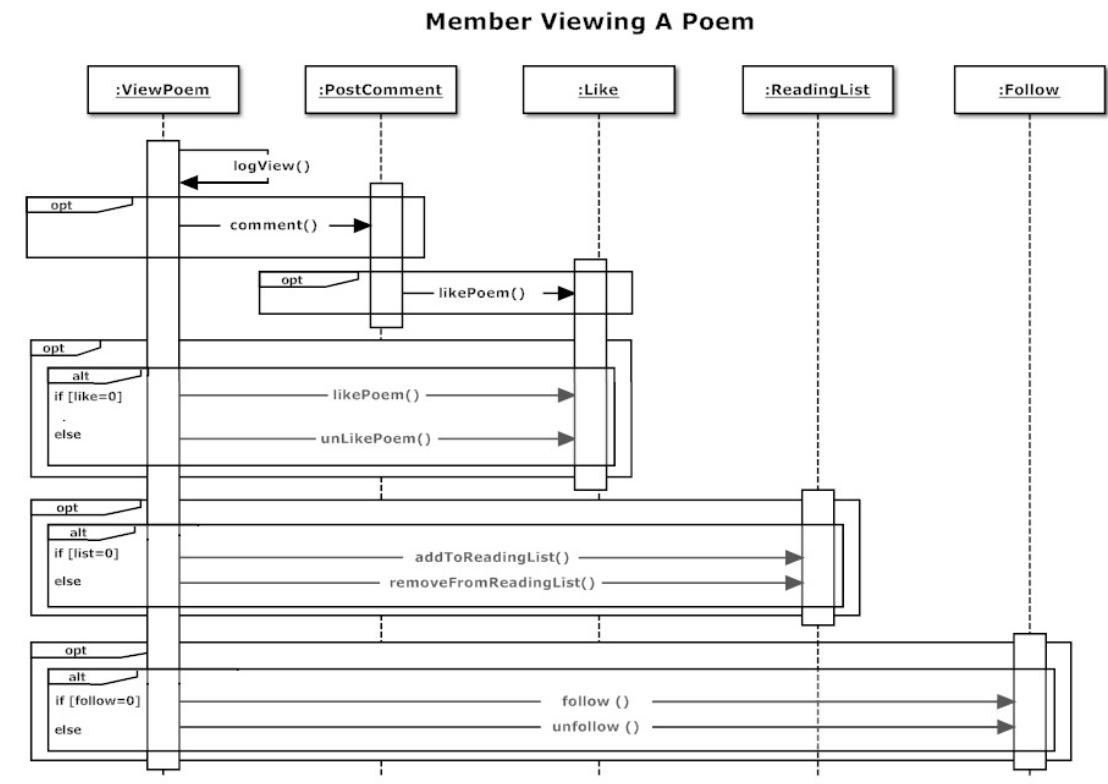
- Set creation module for generating probe and training sub-sets for testing and tuning algorithms.
- Engine module for training and computing final results.
- Classes for Poem, Member and Rating objects.

On-site Perl scripting consists of:

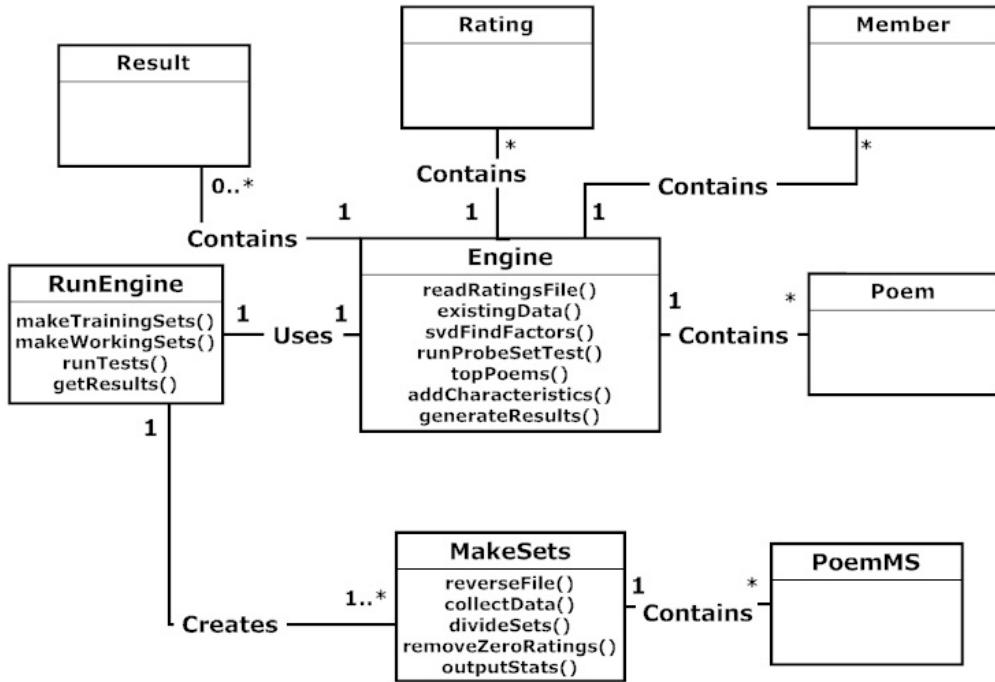
- Data generation subroutine for collecting ratings data from the on-site SQL database and collecting poem characteristics data.
- Comment data subroutine for collecting poem comment text content for analysis.
- Upload results subroutine for inserting results from offline tool into the on-site database.
- Recommendations test page subroutine for generating front-end test page for members included in the user evaluation.
- Favourite recommendations subroutine to deal with a member's selection of their favourite list of recommendations.
- Questionnaire and submit questionnaire subroutines to generate front-end page for members to complete the questionnaire, and back-end for checking and inserting the results into the database.
- Recommendations page subroutine to generate the front-end of final recommendations page.

3.4.1 Activity Diagram (a member viewing a poem)

The following sequence diagram describes the interactions which take place when a member views a poem. A member viewing a poem can leave a comment (and simultaneously a “like” via a checkbox on the comment form). They can also leave a “like” as a separate action (or undo an existing “like”) or add/remove the poem from their reading list. They can also opt to “follow” the author poet (which can also be done via the author’s profile).



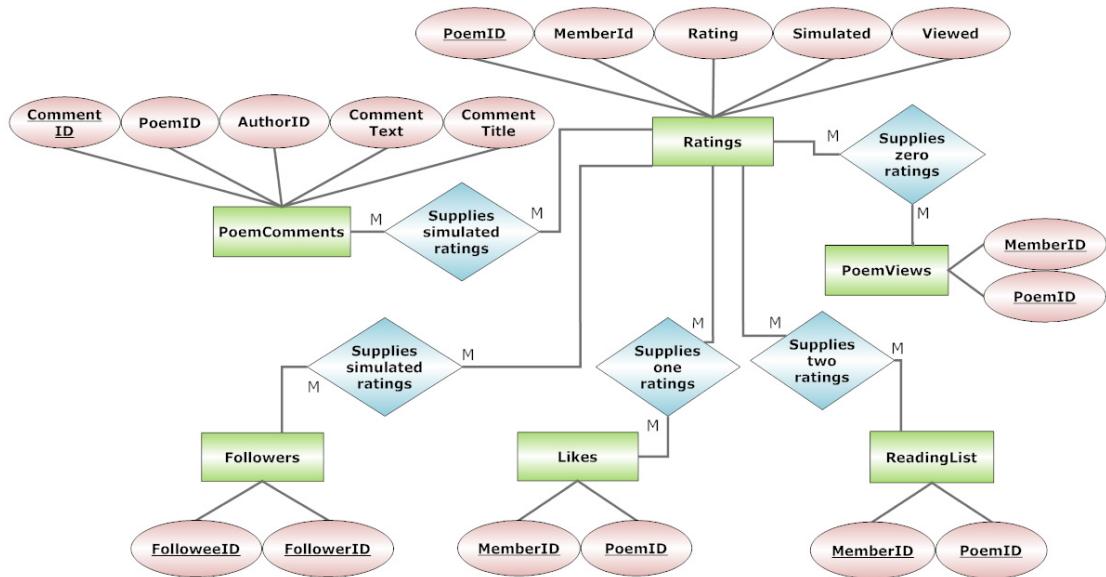
3.4.2 Class Diagram (offline tool)



The class diagram model shows the classes in my program, along with the main public methods in each class. “MakeSets” is called from “RunEngine” each time a ratings set needs to be used for training; it generates the two sub-sets (training and probe). The engine computes the predictions for both training and final results.

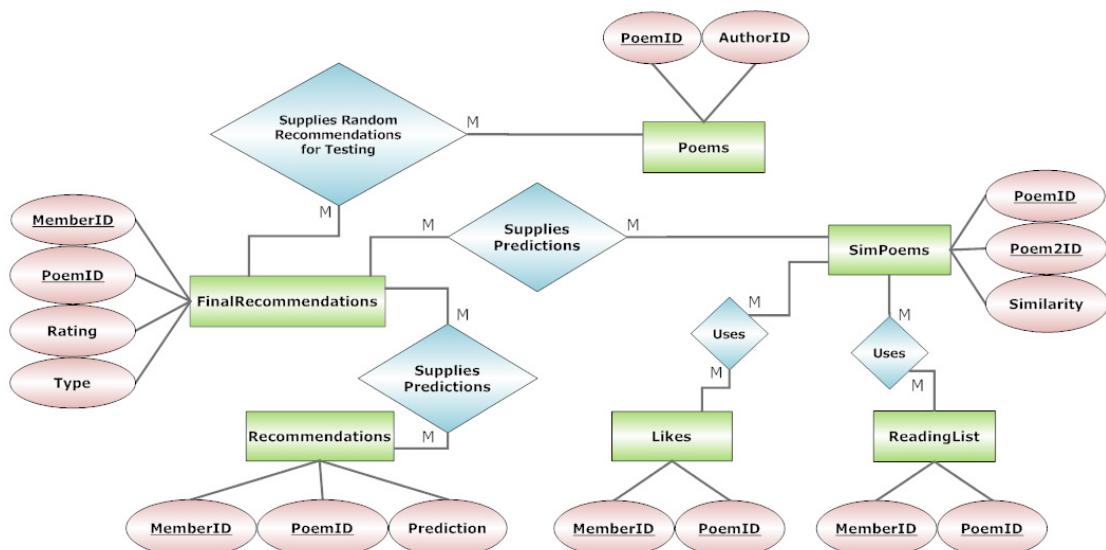
3.4.3 Entity-relationship Models for Proposed MySQL Tables

The following diagram shows the “ratings” table which was created to store existing and simulated ratings just prior to outputting to file (using Perl scripting). This file is then used by the offline tool to train the existing data. The diagram illustrates which existing tables the ratings information comes from. Although retaining this information is not necessary, as it largely duplicates existing information and could be obtained using a sub query; it is simpler and more useful (for performing analysis) to store it in a table temporarily. I have also included the ratings value attributed to the different kinds of known information. For example, a poem view (where there is no “like” or reading list entry) counts as a zero value rating.



An entity-relationship model using Chen notation,
showing relevant tables used for generating known ratings data.

In contrast with the previous diagram, the following model shows the process at the other end; dealing with input data generated by the offline tool. The “Recommendations” table is where member-poem specific recommendations are uploaded and the “SimPoems” table contains uploaded data relating to the most similar poems. The latter requires using the existing “likes” and reading list entry tables in order to generate member specific predictions based upon current user preferences. The “FinalRecommendations” table was used during testing to contain the different types of predictions used in the member evaluation process. This table was not needed to produce final (post test phase) recommendations, as they could be taken directly from the “Recommendations” table, or sub-queried from data in the “SimPoems” table. The “Poems” table was used to produce a control group of random predictions for testing.



An entity-relationship model using Chen notation,
showing only relevant tables and columns for providing recommendations to members.

Development

4.1 Collecting Data

I used existing data from *DU Poetry's MySQL* tables relating to poem views, “likes” and reading list entries. For testing and tuning the algorithms I divided the data into two sub-sets, one for training the data and making predictions, and a probe set for comparing actual ratings to the predictions made (computing the error).

Initially I assumed that data for poems with only zero value ratings (viewed but never “liked” or added to a reading list) and data for members with only zero ratings (poems read but never “liked” or added to their reading list) would not be of any use, so I tried to filter these ratings out. I attempted to perform this filtering using *SQL* queries but this proved very slow; I was having to deal with a few thousands poems at a time to avoid exceeding my web host’s threshold on terminating long running queries. I was able to resolve this performance issue by investigating the explain plan for the query, and as a result, adding an index to the poem ID in the “likes” and “favs” (reading list entries) tables. Although the poem ID already comprised part of a composite key for these tables (together with the member ID), this was not being used effectively in performing the select query.

I wrote a *Perl* script to perform the queries and output the results to a text file. The resulting script retrieved the data very quickly. In light of this I decided that it would be advantageous to download all ratings data (including items with only zero value ratings) and write a C++ component (MakeSets) to generate the probe and training sets. This also allowed me to build in lots of options (for example, the number of sets to generate and whether or not to allow the different types of zero valued ratings). This component ensured that I could easily create sets in a variety of ways in order to comprehensively test the implications of including different types of data.

Although, there may have been isolated/unconnected data in the original data set, when selecting suitable probe set entries, I ensured that I was giving the algorithms a fair chance to make predictions. There was little point including items in the probe set which had no corresponding data in the training set. Producing a good probe set was especially important in gauging the true error, because the testing RMSE was expected to be worse than that of the final results. This was owing to the reduced amount of data; a proportion of the available data was in the probe set, and therefore unavailable for training. I incorporated a variable for setting a minimum number of training set entries required for a poem to be considered for inclusion in the probe set. Furthermore, I added a variable to set a maximum number of entries per poem allowed in the probe set, primarily to give variety and help ensure a good distribution of data about each poem across the two sub-sets.

Another consideration when making the sub-sets was to try and give as true a representation as possible of the ratio of ratings witnessed in the full set. I made it so the sets would be generated in such a way as to match the distribution of values across the two sub-sets. This was important because a different make up of values in a training set makes an impact upon the final RMSE achieved and I wanted to be able to compare the error produced by the algorithm, with that produced using an average rating as a prediction. I also incorporated the means to use existing probe sets and generate corresponding training sets (the contents of the input file, with the specified probe set entries omitted). This meant I could test different training sets against the same probe set, as long as all items (members and poems) in the probe sets were also present in training. This gave me a variety of options so that I could more effectively test and tune the algorithms, and the impact of the inclusion of different types of training data.

In order to further improve the diversity of the training and probe data, I wrote a procedure to reverse the ratings file in order to start selecting potential probe set entries from the highest poem id (most recently submitted poems) first. Every alternate probe set is generated using the reversed ratings file. For the purposes of tuning the algorithms I typically generated only two sets, in order to give variety but also mindful of the extra time needed to run multiple tests.

4.2 Utilizing Implicit Data

One of the key challenges I faced, was to overcome the limits of a sparse and potentially noisy data set, by finding the best possible way to use the available data to achieve my aims; to provide accurate and diverse recommendations for each member. In order to produce sufficient data for training, I used implicit data relating to which poems each member had viewed. However, this introduced a large number of zero value ratings into the data. These can be considered as a negative reaction to a poem because there is no other way to gauge this kind of reaction (there is no “dislike” option). It is expected that if a member likes a poem, they will take an appropriate action to record this (register a “like” or add the poem to their reading list). However, there may be a number of other reasons why a member did not take a positive action, for example a technical problem, or they simply forgot to make use of the feature. I wanted to look at ways of using other data in order to make a best guess as to whether the visitor did indeed like the poem, despite not taking an action. All of these “simulated” ratings were flagged in the downloaded data file to ensure that they did not qualify for inclusion in probe set, and were therefore effectively being tested to see if they could improve the training process.

A major difficultly with assessing the effects of including different types of data is that the RMSE is not always comparable, as the make up of the probe and/or training sets could be different in each case. If the same probe set is used to evaluate training sets with different ratios of ratings, then the probe set requests may favour one more than the other. Yet if the probe sets are different, it could also be deemed not to be a fair comparison. To help distinguish the true effect of data changes, I have computed the average RMSE for each test; the error witnessed when using the average rating seen in the training set as a prediction for each probe set entry. The improvement in RMSE over this set specific average may be considered a useful indicator in accessing whether or not to include certain types of implicit data in the final training process. Another consideration when evaluating the results was the quantity of poems and members included in the computations. The more items I could include the better, as I wanted to be able to make predictions for as many members as possible, and include as many poems as possible in these predictions.

4.2.1 Comment Analysis

The first area I looked at was poem comments, which is a large resource of over 38,000 comments made by more than 2000 members (not including comments on an author's own poems). Intending to make use of the existing MySQL full text search index on both poem comments and comment titles (used for the website's search facility), I considered how to go about choosing suitable words to indicate a positive reaction to a poem. Rather than just guess at which words to use, I wanted to carry out some analysis on the text content of comments. For this analysis, I downloaded the content of all comments (and titles), not including those made prior to the “like” feature being launched. I created two sets, one for comments where the author “liked” the poem and the other for comments where they had not. I used text analysis software to count the prevalence of words in the two sets [23]. Many of the most frequently occurring words appeared to be positive, which supported my earlier

findings that people comment more frequently on poems they enjoy. In further support of this, there were 2.25 times more “liked” comments than “un-liked” comments (14320 to 6358). I multiplied the “un-liked” comment results to give comparable word counts. Words with the highest percentage difference across the two sets (more than 50%), where the count was higher in the “liked” set, were as follows:

Word	Freq (liked)	% Difference	Freq (un-liked)	Manual Analysis (25 un-liked samples)		
				positive	neutral	negative
1 excellent	340	372.22	72	19	5	1
2 list	260	231.63	78.4	6	6	13
3 review	277	188.54	96	8	15	2
4 beautifully	210	176.32	76	21	4	0
5 loved	558	164.45	211	21	4	0
6 amazing	719	157.71	279	21	2	2
7 enjoyed	604	137.80	254	22	2	1
8 beautiful	1131	136.12	479	23	2	0
9 wonderful	428	130.11	186	20	4	1
10 love	3198	96.32	1629	18	7	0
11 brilliant	184	91.67	96	20	5	0
12 perfect	265	90.65	139	18	4	3
13 awesome	374	87.00	200	20	4	1
14 great	1920	71.74	1118	19	5	1

Green rows indicate selected words and red rows are words which I deemed unsuitable because they failed the manual test. The manual test consisted of looking at a random sample of 25 comments in the “un-liked” set containing the given keyword, and deciding if the comment was positive, neutral or negative. This allowed me to get an indication of the context in which the words were being used and therefore confirm if they were suitable for generating simulated “likes”. I was able to convert over 7000 zero value ratings to a one, by matching the keywords above for comments made prior to the “like” feature being launched. I extended this to test for all matching comments, irrespective of the date published, this generated a further 2000 conversions from a zero to a one rating.

	Smart Blend RMSE	Average RMSE (Smart Blend Improvement)	Average Rating Prediction Made
Including Comment Simulated Ratings (all dates)	0.4932	0.5254 (6.1%)	0.244
	0.4942	0.5195 (4.9%)	0.271
Including Comment Simulated Ratings (pre-like feature)	0.4975	0.5251 (5.3%)	0.235
	0.4877	0.5129 (4.9%)	0.262

Not Including Comment Simulated Ratings	0.5017	0.5270 (4.9%)	0.180
	0.4650	0.4869 (4.5%)	0.183

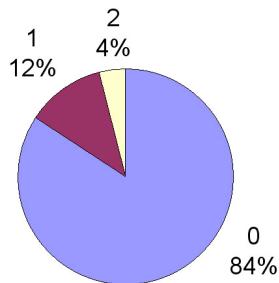
Due to the difficulties in making direct comparisons between data sets with different distributions of ratings values, I tested the error using two different kinds of sub-sets. Both used identical tuning settings across the three examples. The top row of each example also used identical probe sets across the three tests (the ratings which are in some cases simulated were not included in these probe sets). The bottom row uses probe sets unique to the data, which exactly match the distribution of rating values present in there respective training sets (a probe set contained the same percentage of each value present in its corresponding training set).

I have also included the RMSE achieved when using the average of all the ratings in the training set as a prediction for each of the probe set entries, along with the percentage improvement over this value, which was achieved through training. You can also see that the average value of the recommendations made increases, as you would expect, when more high values are present in the training set. From the results I witnessed, I was satisfied to go ahead and use comment simulated ratings, for all dates, in my tuning and final training process.

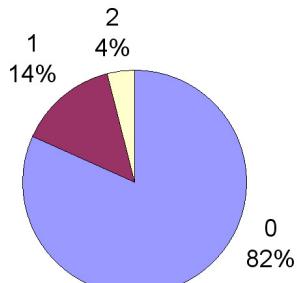
4.2.2 Followers

I wanted to investigate the possibilities for using data about which authors a member had decided to “follow”. Following an author indicates that they like their poems because it provides an update every time the followed author publishes a new poem. It also places a link, serving as an endorsement to the followed author, on the member’s profile. It’s worth noting that there were 190 members who had chosen to follow at least one other member, but had never “liked” a poem. This amounted to a potential of 23005 poems (1011 viewed) for which a rating could be simulated, and therefore recommendations made to that member, despite the fact that they had never taken an explicit action on any poems.

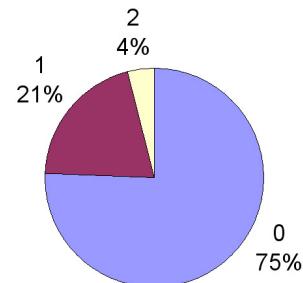
The pie charts below show the breakdown of zero, one and two ratings observed within each set. In all examples, zero only rated data has been included (poems and members with only zero ratings). The first contains no simulated ratings for followed authors. The second includes them only when a comment has been left and the poem was viewed prior to the launch of the “like” feature. The third includes them regardless of whether a comment was left. The fourth and fifth pie charts consider all follower data regardless of view date. A major advantage of including simulated follower data is that the percentage of one value ratings increases, helping to address the large percentage of zero ratings found in the data set.



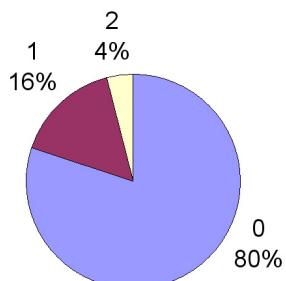
No Followers



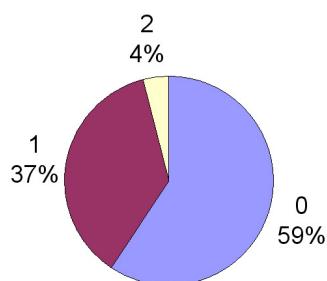
Pre-like Followers (where a comment was left)



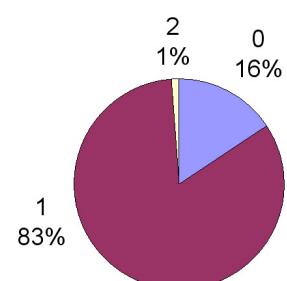
Pre-like Followers



All Followers (where a comment was left)



All Followers (read poems)



All Followers (including un-viewed poems)

The final (bottom right) pie chart shows the effect of including all follower data, regardless of whether the poem had been viewed by that member or not. This vastly increased the total number of observed ratings, increasing from 153013 (in the other examples) to 566864, with the majority of these coming from simulated ratings. This increased data set was populated overwhelmingly with one value ratings. The ratings data also required an extra field to show if the poem had been read, as any poem not read by a member, is a candidate for a recommendation. Furthermore, including all this extra “follower” related data had the potential to skew the final recommendations in favour of poems written by authors already being followed by a member. This was undesirable, as it could limit the diversity of recommendations, and frequently suggest poems which the member should already know about (via their updates page). The following table shows the results of testing each data set.

	Smartblend RMSE	Average RMSE + Improvement of Smartblend
No followers Information Included	0.4658	0.4871 0.0213 (4.37%)
Followed authors poems get 1 value rating (pre-like feature and with comment only)	0.4736	0.5020 0.0284 (5.66%)
Followed authors poems get 1 value rating (pre-like feature only)	0.5067	0.5311 0.0244 (4.59%)
Followed authors poems get 1 value rating (any date and with comment only)	0.4821	0.5106 0.0285 (5.58%)
Followed authors poems get 1 value rating (any date)	0.5242	0.5710 0.0468 (8.20%)
Followed authors poems get 1 value rating (all, even un-viewed poems)	0.4059	0.3996 -0.0063 (-1.58%)

All tests were run with the same settings and matching ratios of ratings across the probe and training sub-sets. Zero only value ratings were included and a total of 24888 poems and 2813 were witnessed (slightly more poems were present in the final set, which included un-viewed poem data and included 25673 poems).

Although the lowest RMSE was achieved by including all follower data, including un-viewed poems, this test also achieved the lowest improvement on the average RMSE, in fact performing worse. The make up of this set, which was almost entirely populated by ratings of the same value, lends itself to a low RMSE. However, this doesn't necessarily mean that including this data is beneficial, as the wealth of simulated data causes the explicit data to be diluted by less useful implicit data [24]. This may explain why the improvement witnessed is negative.

The largest improvement on the average RMSE was achieved when all "follower" data for only viewed poems was used, yet this provided the worst overall RMSE. I felt at this stage that the results of my analysis with "follower" data had been inconclusive and I furthered my investigations later by testing two different approaches in the final on-site user testing.

4.3 Implementing k-NN

I implemented the k -NN algorithm by comparing the similarity between items, either members or poems, using a variety of distance metrics. Using the generated similarity scores, together with known ratings, I produced a weighted prediction for each line in the probe set. Along with the capability to compute a predicted rating based upon a weighted score (similarity) for every poem, I also wanted to have the option of setting a number for k (the number of nearest neighbours to consider). This offered no benefit in terms of memory usage or performance, as the program still needed to work out all the similarities (which is the computationally heavy aspect of k -NN), and then include an extra step to sort the top k most similar results. It required an optional modification to how the similarity vector was created:

```
1. if (numNeighbours > 0) // zero means consider all neighbours
2. {
3.     partial_sort(eKnnTopKSim.begin(),
4.                 eKnnTopKSim.begin() + numNeighbours, eKnnTopKSim.end(), compareSim);
5.     eKnnTopKSim.erase (eKnnTopKSim.begin() + numNeighbours,
6.                         eKnnTopKSim.end());
7. }
```

The `compareSim` function on line 4 simply returns a boolean value to indicate if the first value is greater than the second. I used this code variation to test a number of values for k but was unable to yield an improvement over considering all neighbours. All tested values of k performed worse, although at 200 neighbours the difference in RMSE was negligible. However, there was no benefit of reduced computation time, so considering all neighbours was the best option.

Number of weighted k nearest neighbours	RMSE (item based k -NN)	% of zero value predictions using Euclidean distance	Time in seconds to complete
All	0.5145	12.2%	409
25	0.5323	16.9%	426
50	0.5256	13.3%	422
100	0.5227	12.4%	412
200	0.5189	12.2%	415

4.3.1 User-user Matrix

Due to the small number of users compared with items (poems), this was the faster approach to k -NN, taking only a few minutes to generate a full matrix of user-user similarities. However, this method produced a very high RMSE as there weren't enough similarities between members. Many of the probe set predictions were whole numbers, indicating that either the user had no shared ratings with members who rated the requested poem (yields a zero), or all members with commonly rated poems gave the requested poem the same rating (100% similarity). This is characteristic of a sparse data set and in particular the *DU Poetry* data set, where there are many more poems than there are members. I wanted to be able to provide useful predictions for lots of members, including those who had only given a few ratings; therefore the usefulness of this user-user approach was limited.

RMSE	Time (seconds to complete one probe set of 10000 entries)
0.6381	216

4.3.2 Item-Item Matrix

I then looked at tackling k -NN using an item-item (poem-poem) approach; looking at the other poem ratings given by members who rated that poem, to find the similarity between poems. This opened up a larger number of available similarity scores. The problem with this method was the computation time, rather than a member-member matrix of around 3000 x 3000 (9 million comparisons), it required dealing with approximately 25000 x 25000 (625 million comparisons). In reality, when constructing a full similarity matrix, these comparisons only needed to be computed half as many times; I didn't re-compute a comparison of poem x and poem y when working out row y (having already worked out row x). However, there was still an enormous difference in computation time between user-user based k -NN and item-item based k -NN.

RMSE	Time (seconds to complete one probe set of 10000 entries)
0.5145	5317

Maintaining a full item-item similarity matrix in memory was not possible once the number of poems reached a certain level. An "abnormal program termination" error was given by the Borland compiler, which I discovered related to this memory issue. Once reaching approximately 20,000 poems, there was not enough space available to address the entire vector (on my 32-bit machine). I solved this problem by only keeping a single vector in memory, holding similarities for one poem, compared to all other poems, but only those which had ratings in common with that poem. I created the vectors at the time a prediction was being made.

Along with information about item similarities, a matrix (2D vector) of all known ratings also needed to be stored within the recommendation engine. This has the potential to use a large amount of memory space, although less than the item-item similarity matrix, as the known ratings matrix takes the form user-item (and there are far less users than items). I wanted to look at other options for providing a more scalable solution by avoiding the need to store information about all the unknown ratings, which comprise the majority of the matrix. It's also worth mentioning that I opted to use the "short" data type instead of "int", in order to half the number of bytes required to store each rating.

I tested various alternative approaches; firstly I tried using a "map" from the standard C++ library, in which to store all known ratings, using a "pair" to create a key from both the poem id and member id related to the rating. However, this method proved to be extremely slow (over 100 times slower than the 2D vector). I was able to speed it up slightly by implementing a very simple hashing method instead of using "pair" to hash the key [25]. My custom key took the form:

```
myKey = poemid * max_memberid + memberid
```

However, I was able to improve performance more significantly by using a separate map within each Poem object, for ratings relating to that poem. This meant lookups were being performed within a smaller map each time, and I could use a single value (member id) as a key. This method was still over 47 times slower than the 2D vector matrix.

Next I investigated hash based storage methods, as I knew they offered good random access speeds via indexing. I used the "unordered map" feature from the *Boost* library [26] because it was significantly faster than the version in the *Visual C++* standard namespace. It also offered a benefit over the standard version in that it had built in support for using "pair" as a key (in the same way the "map" object does). As before, I also tested this using my own hashing method (detailed above) and using a separate map for each poem. The "unordered map" was significantly faster than the map, but still over nine times slower than the 2D vector.

The final method I attempted involved having a vector of pointers within each Poem object, pointing to the applicable Rating objects. These were stored in order of ascending member id so they could be looped through until the member id value was found or passed. My results for each of the approaches outlined, and the time taken to make 15 predictions using item based k-NN and Pearson correlation are as follows:

Method	Time (seconds)
Full 2D vector matrix	28
Map, all ratings (using "pair" as key)	2841
Map, all ratings (using custom method to hash the key values)	2025
Map, per poem	1324
Boost unordered map, all ratings (using "pair" as key)	261
Boost unordered map, per poem	188
Unordered map, all ratings (using custom method to hash the key values)	162
Vector of pointers to ratings objects, per poem	248

Due to the fact that none of these methods provided a viable option, with even the fastest being nearly 6 times slower than the original 2D vector matrix, I concluded that a

hardware solution may be my best option. With my current data set, the full matrix can be maintained in memory, however as the number of members and poems grows it may become necessary to upgrade to a 64-bit machine and operating system in order to address the full vector in memory [27].

4.3.3 Distance Metrics

A metric is a function which defines a distance (or similarity) between items. I chose a selection of the most commonly used metrics in k -NN in order to tune my algorithms. On a sparse data set, working out distance can be problematic because there may not be many ratings in common between items. For example, two poems with one rating in common, which is the same value for both, may be deemed as perfect similarity. However, a poem which shares many ratings that match (and one or two that don't) may be given a lower value, despite potentially being a much more solid match than the first, which may have been skewed by a single inaccurate noise rating. Unfortunately this cannot be remedied easily, as giving more weight to relationships with lots of ratings in common would have the undesirable effect of recommending popular items more often, at the expense of relatively undiscovered (less often rated) items.

Euclidean Distance

This is the straight line distance between two points, and is given by the Pythagorean formula. Below is the equation for Euclidean distance, along with my function, which gives a result between 0 (no common ratings found, no similarity) and 1 (perfectly similar) for each item. I have written it in such a way as to avoid any potential divide by zero errors.

$$d = \sqrt{\sum_{i=1}^k (X_i - Y_i)^2} \quad 1.0 / (1.0 + \text{sqrt}(\text{sumSquares}))$$

The value *sumSquares* is the sum of the squares of the distance between each rating that the items have in common. This is achieved by looping through each item (member for item k -NN or poem for user k -NN) and adding to *sumSquares* each time a common rating is found. Considering the line below, for item based k -NN; *id1* and *id2* would be two different poems, and *i* is the current member in the loop.

```
sumSquares += pow(eKnnMatrix[i][id1] - eKnnMatrix[i][id2], 2);
```

Manhattan Distance

This is the distance between two points in a grid based on a horizontal or vertical path along grid lines, in contrast to the diagonal or "as the crow flies" path of Euclidean distance. Similarly, it gives a result between 0 (no similarity) and 1 (perfectly similar) for each item.

$$d = \sum_{i=1}^k |X_i - Y_i| \quad 1.0 / (1.0 + \text{distance})$$

– A Recommendation Engine
27 of 58

The value *distance* is the sum of the distance between each rating that the items have in common. I use the *fabs* function to return the absolute float value of the distance, this removes the sign on negative values, making all distances positive (the direction of the distance doesn't matter and it avoids an incorrect sum). As with Euclidean distance, a zero value is returned if the items have no ratings in common.

```
distance += fabs(eKnnMatrix[i][id1] - eKnnMatrix[i][id2]);
```

Average Based Similarity

I included a simple averaging function, which returns the similarity computed from the average distance across all shared ratings.

```
1.           / (1.0 + (sumDistance / ratingsInCommon))
```

Where the following code is executed each time a common rating is found:

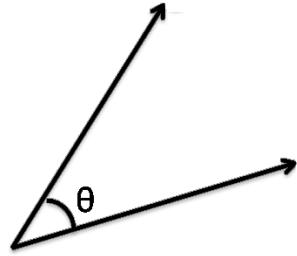
```
sumDistance += fabs(eKnnMatrix[i][id1] - eKnnMatrix[i][id2]);
ratingsInCommon++;
```

This average based similarity function yielded almost identical similarity scores to Euclidean distance.

Cosine Similarity

Cosine similarity is an evaluation of the similarity between two vectors by measuring the cosine of the angle between them, to determine whether they are pointing in roughly the same direction. The resulting similarity ranges from -1 meaning exact opposites and 1 meaning perfect similarity, with 0 indicating independence (no correlation).

$$\text{sim}(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$



The magnitude is the sum of the squares of each poem's values (lines 3 and 4) and the dot product is the sum of multiplying each of the common values (line 5).

```
1. val1 = eKnnMatrix[i][id1] + 0.1;
2. val2 = eKnnMatrix[i][id2] + 0.1;
3. id1sumsq += (val1 * val1);
4. id2sumsq += (val2 * val2);
5. sum += (val1 * val2);
```

I added a small value to the known rating values, as the function wasn't accounting for zero value ratings properly, and was subsequently returning exactly zero (no correlation) scores more often than expected. This also remedied the possibility for divide by zero errors in the final result:

```
result = sum / (sqrt(id1sumsq * id2sumsq));
```

Cosine similarity performs badly where there are few ratings in common, as it looks for patterns. This issue is discussed further in the following section on Pearson correlation.

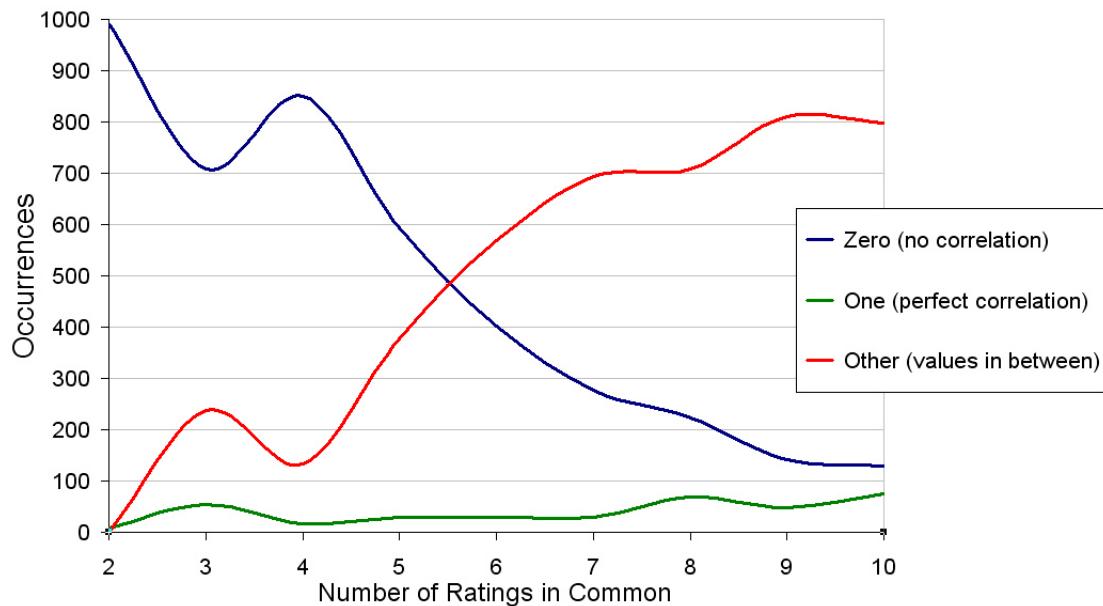
Pearson Correlation Coefficient

The Pearson product-moment correlation coefficient is a measure of the correlation (linear dependence) between two variables. As with cosine similarity it returns a value between $+1$ and -1 inclusive.

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

Due to the nature of a correlation measurement, the Pearson coefficient performs badly in cases where there were only a few items in common, because it relies on finding a correlation between the ratings given. It tends to yield a zero (no correlation) very frequently in cases with a low number of ratings in common. The more useful "other" results, which lead

to a good variance in similarity scores, are not witnessed in volume until the number of shared ratings increases sufficiently enough for the algorithm to observe correlations between items.



The graph shows similarity score results for 1000 of each case (number of ratings in common) using item based k -NN (members who have rated both poems).

Metric	RMSE
Euclidean	0.5151
Manhattan	0.5209
Average	0.5142
Cosine	0.5638
Pearson	0.5679

RMSE results for each metric, using identical item k -NN settings and sets in each case.

The results show that the Euclidean distance and average function perform well. As mentioned, the correlation methods (cosine and Pearson) performed badly.

4.4 Implementing SVD

I based my SVD implementation on highly regarded suggestions made by Simon Funk during the *Netflix Prize* [14]; his ideas were widely used and adapted by other teams in the competition. The program starts by working through each of the defined number of factors, until such a time as a chosen minimum number of iterations have been reached. The program may go beyond the minimum number of iterations if a defined amount of improvement in RMSE is still being achieved (compared with the previous iteration). Each iteration loops through all of the known ratings, first working out a prediction based on existing values. During training, the prediction is made up of the sum of three parts. The first is the Rating object's cache; the final prediction made at the end of training the previous factor, which takes into account the results of all factors trained so far.

```
double sum = (cache >= 0) ? cache : ePoems[poemid].wAvgRating;
```

If there is no cached value (we are working on the first factor) then a weighted average of the poem's known ratings is used instead. The weighting makes use of the regularisation parameter "svdReg" (discussed later) and the average of all observed ratings, in order to reduce the potential impact of noise values, within ratings received by that poem.

```
ePoems[i].wAvgRating =
(averageRating * svdReg + ePoems[i].sumRating) /
(svdReg + ePoems[i].numRating);
```

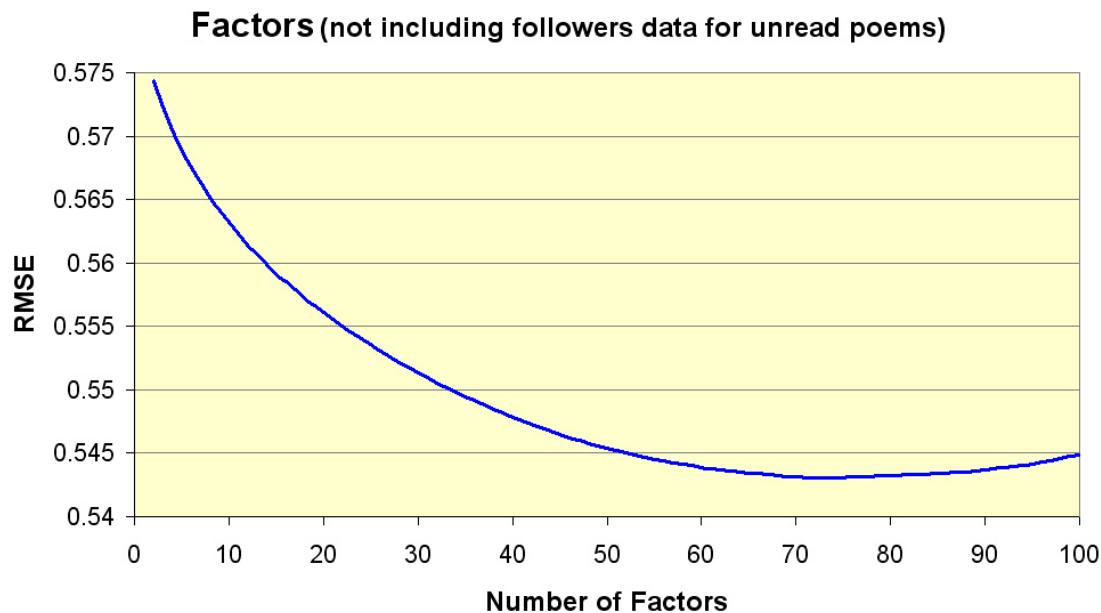
The second part of the prediction adds in the influence of the rating we are working on; the product of the most current factor values for the poem and the member involved. The third part of the prediction adds in the factor initialisation values for any factors which we haven't trained yet, as the predicted rating needs to take into account all factors. The RMSE between the known rating and the predicted rating is then computed and the existing and new information is cross trained, using the method suggested by Simon Funk [14].

```
eMemberFactors[f][memberid] += (float)(lRateMod * (err * pf - svdReg * mf));
ePoemFactors[f][poemid] += (float)(lRateMod * (err * mf - svdReg * pf));
```

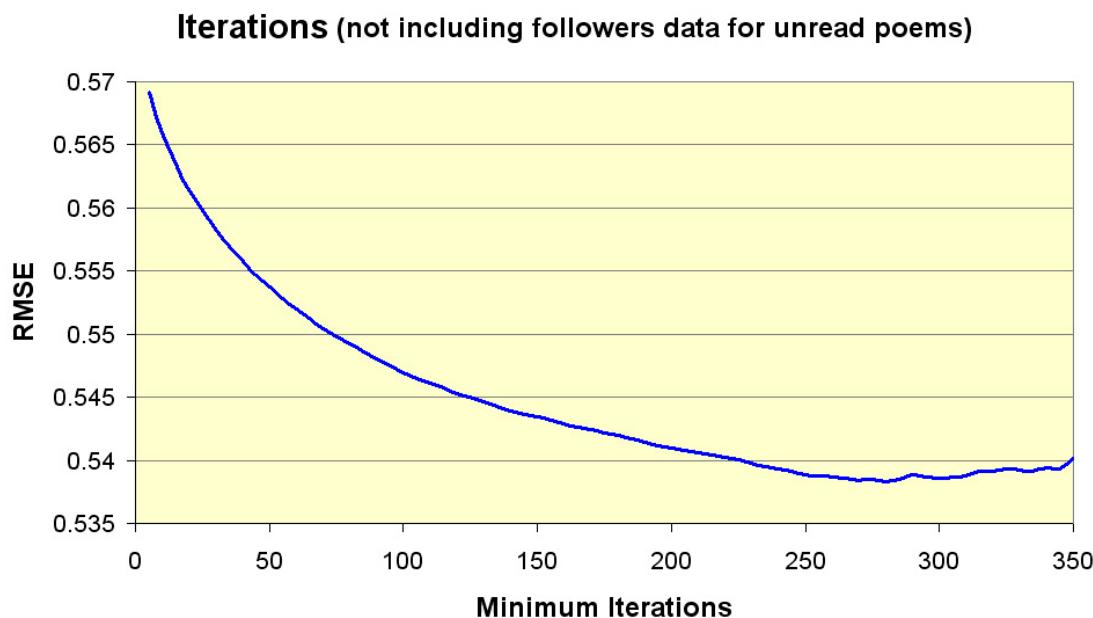
"lRateMod" is the learning rate, with an addition I made to give an option for varying it during training, "err" is the RMSE mentioned above and "svdReg" is the regularisation parameter at the core of Simon Funk's modified method of SVD. The learning rate and regularisation parameter are discussed in more detail later. At the end of training each factor, the current predicted value (taking into account all training so far) is computed for each known rating and stored in their respective "cache" variable (within the Rating object).

4.4.1 Tuning the Algorithm

As a starting point for tuning my algorithm I looked at the number of factors (features) to generate. There was a steady decrease in RMSE up until an optimum number of 71 factors, after which the RMSE started to increase again. It took an average of 2.665 seconds of extra processing time per factor, per training set (of approximately 132,000 ratings). I ran tests with several different training set variations (different data simulations, for example) and found a similar pattern, with the optimum number of factors always emerging as either 71 or 72.

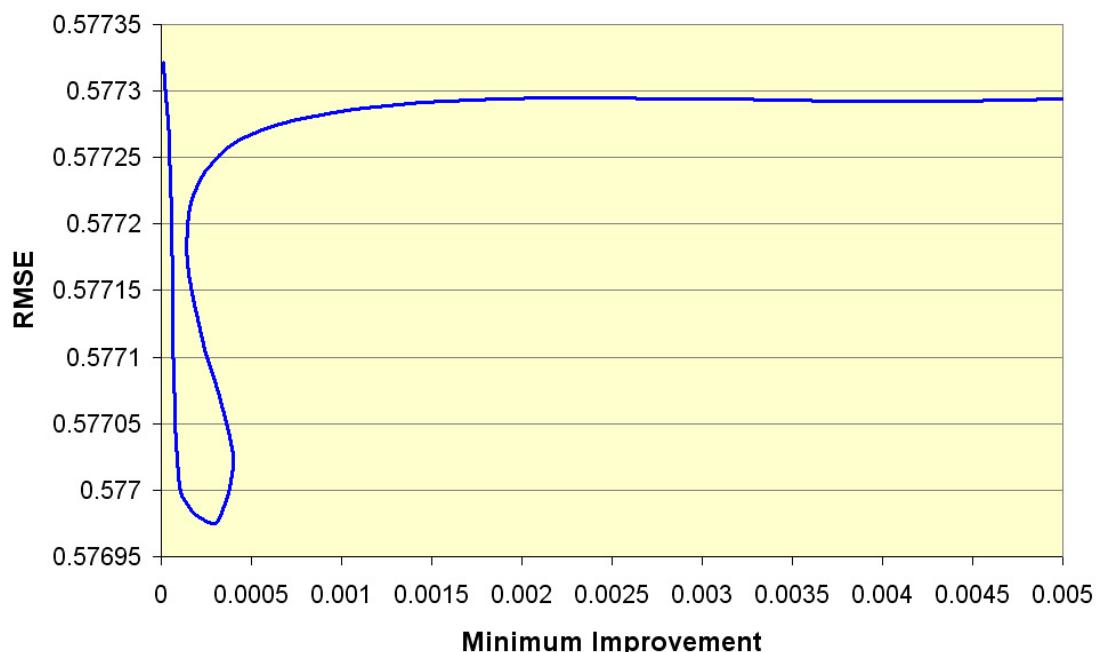


Next I looked at the minimum number of iterations required to train each factor. This determines when to stop the training process; if it is set too high, it could cause over-fitting and if set too low it may mean we never reach the optimal solution [28]. The results produced a similar curve, and RMSE was lowest at 280 iterations. When more data was included (for example un-viewed poem “follower” data) this increased slightly to 295 iterations. It took just over two seconds extra for each additional iteration (per training set).



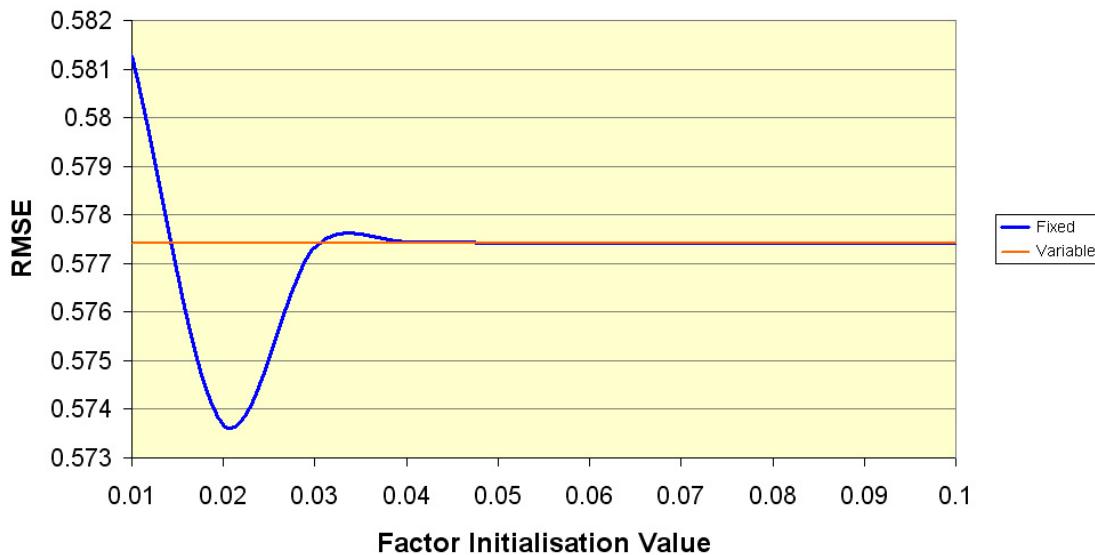
To supplement the “minimum iterations” setting, I also included a “minimum improvement” variable. This allows the program to go beyond the minimum number of defined iterations, on condition that the error achieved is significantly lower (by the defined “minimum improvement” value) than the previous error. This had the advantage of ensuring that the process of training the current factor was not ended prematurely, if significant improvements were still being made. The graph below shows how the minimum improvement variable affects RMSE, using pure SVD over one training set example. Any value above 0.005 yields an equivalent RMSE because there are no occasions where the RMSE of the iteration improves upon the previous iteration sufficiently.

Graph showing how the “minimum improvement” variable affects RMSE



Next I considered the factor initialisation value, which is the starting value attributed to each poem and member for each of the defined number of factors. Along with testing a range of different factor initialisation values, I wanted to see the effect on RMSE of varying the initialisation value for each factor, along with any changes in the variety of poems ranking highly for each factor. I added a small random “salt” value (+/- the specified initialisation value) to give the variation.

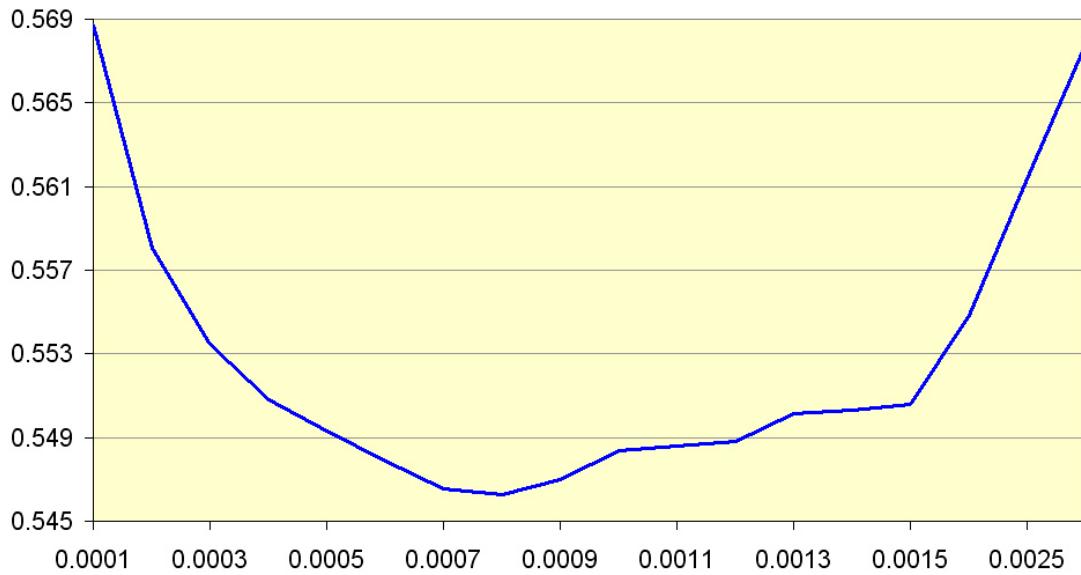
The Effect of the Factor Initialisation Value



Aside from a marginally improved RMSE using a fixed initialisation rate at 0.02, there was little to choose between the two approaches (fixed and variable initialisation values). The effect of the initialisation value on the algorithm overall can be deemed as very small. It levels out completely above 0.08; I also tried much higher values (for example 25) and achieved the same resulting RMSE. This makes sense in the context of SVD; as long as the values attributed to each item for each factor work in relation to each other then the actual values themselves are inconsequential. I analysed the top ten fifty poems for each of the 71 factors generated in each test, using an initialisation value of 0.05 (where the RMSE was the same for fixed and variable). Of the 3550 poems which resulted, although the order and values were different, the same 238 different poems appeared for both the varied and fixed initialisation values.

The learning rate parameter represents the step size that the gradient descendent (first-order optimisation) algorithm takes to find the local minimum. Simon Funk suggested a learning rate of 0.001, however for my data set the optimum learning rate throughout the tuning process always emerged as 0.0008. The choice of learning rate had no impact on execution times.

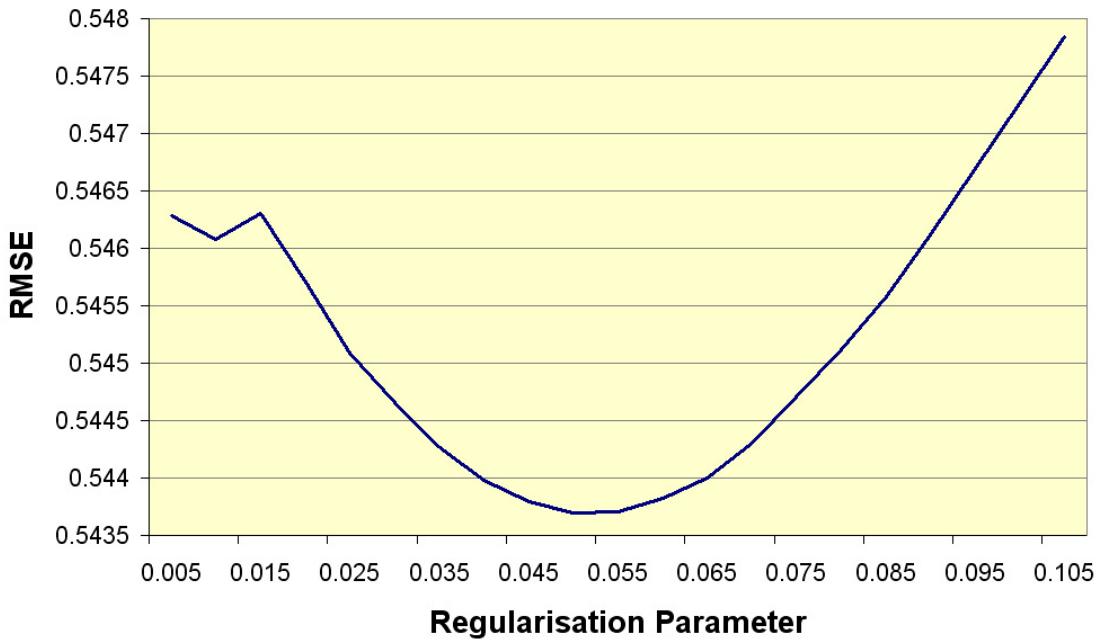
Learning Rate (not including followers data for unread poems)



I noted that a number of approaches used for the *Netflix Prize* and beyond had favoured lowering the learning rate value gradually during the training process [29]. I added an option to vary the learning rate each time a new factor was trained. I tried decreasing it (multiplying by 0.95) and for comparison I also ran a test which increased it (multiplying by 1.05). Both of these variations performed worse than keeping the learning rate constant.

Learning Rate	SVD RMSE
0.0008 (constant)	0.5483
0.0008 (decreasing)	0.5536
0.0008 (increasing)	0.5548

Finally, I considered the “regularization parameter” which is crucial in Simon Funk’s modified approach to SVD. It is used to minimize over-fitting when dealing with a sparse data set; it stops noise values from having too much influence over the training process. It represents the extent to which the program pulls back on the error with respect to the previously computed value, to stop any one value having undue influence over the training process. The number chosen had no effect on computation time and an optimum value of 0.05 emerged.



4.4.2 Analysing the factors

One criticism of SVD (compared with k -NN) is that it's not explainable, in that you can't provide a meaningful explanation as to how you arrived at the predictions made [11, 13]. I wanted to investigate the factors further to see if I could observe any patterns. In order to analyse the factors, I performed a sort on the results, to get the poems which had the highest values for each factor (the number of poems to consider is configurable), and then combined this with selected data relating to the poem. This included; authors age, experience level, the length of the poem, the number of poems the author has published and the poem's category. However, I was not able to find any correlations in the results; if a value dominated in the "top poems" output, it also dominated when considering the membership as a whole. This could be attributed in part to a lack of diversity among members; for example, of the 1607 members who have completed the "country" field in their profile, 1036 of them reside in the U.S.A.

4.5 Blending: Linear Blend

Keen to see if I could improve upon my single approach RMSE values, the first method I tried for blending was to perform a linear blend on the results of the two methods. Initially I set a 50/50 blend ($0.5 * \text{result}$) for each of the two results sets. This was adjusted for each of the sets in turn, starting from 0 to 1 in steps of 0.1 to find the optimum point (where RMSE was lowest). Once I had found the optimum level for the first set I used this (rather than the 0.5 starting value) to tune the contribution from the second set. This method incorporated the flexibility for the two values not to equal one, which meant it could correct a situation where the results were coming out too high or low overall. The blending process was fast on my 10,000 entry probe sets, which allowed me to shorten the step increment to 0.01 and further improve my results.

	RMSE
--	------

Pure SVD RMSE	0.5478
Pure k-NN RMSE	0.5245
Linear Blend 0.7 / 0.4	0.5236
Linear Blend 0.67 / 0.38	0.5188

4.6 Blending: Smart Blend

I was keen to consider other ways of blending the algorithms, in order to try to further improve upon my results. I decided to use the factor data generated by SVD to compute the distance scores between poems (or members, if using a user based approach). Then, as with regular k -NN, these similarities would be used to produce the weighting by which to multiply known ratings.

This also meant that a similarity score could be produced for every poem, even if they had very few ratings (or even no ratings) in common. This helped to produce “similar poems” data for my final results, by linking poems together more easily regardless of the amount of shared information. I implemented this method for both user and item based approaches, but as with previous results, the item based approach was more successful. I called this method *Smart Blend*.

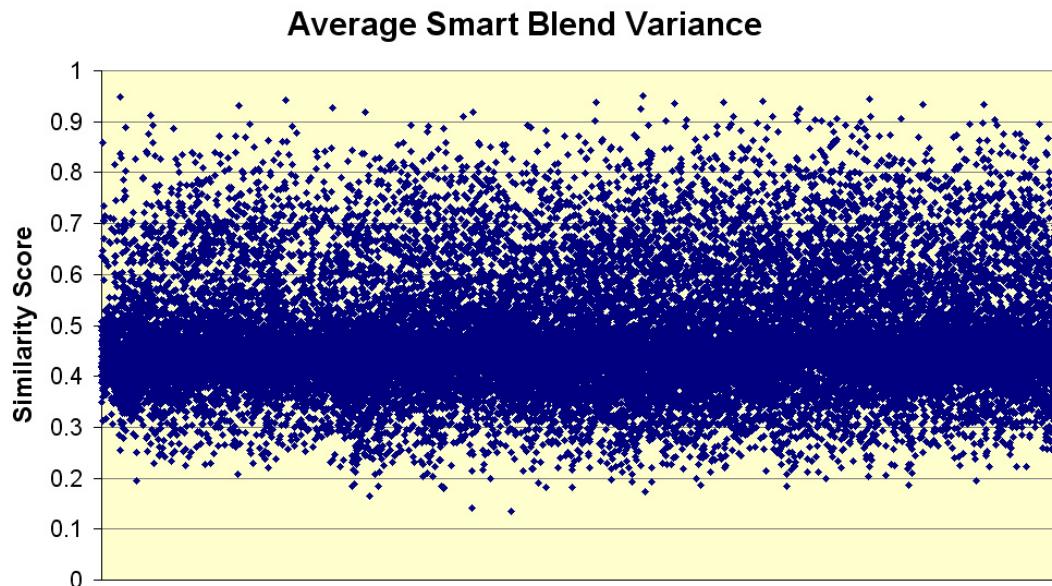
4.6.1 Adjusting the Metrics

The metric functions used for k -NN needed to be adapted to create versions suitable for the *Smart Blend* approach. The values attributed to each item for each factor were different kinds of values to that of the known ratings. Ratings values fall between 0 and 2, but factor values range from [stats here]. Where I previously added one to the Euclidean distance division in the k -NN metric, to avoid potential divide by zero issues, I adjusted this value to be 0.1 so as not to impact on the variation between similarity scores when using Smart Blend. Using the 1.0 value was diluting the differences too much. However, for Manhattan distance the 1.0 addition worked just as well for the Smart Blend version as the k -NN version. In order to achieve appropriate variances I outputted sample similarity scores and adjusted each of the Smart Blend metrics where possible to achieve good variance within the expected similarity score ranges; either 0 to 1, or -1 to 1 (for the correlation functions). The following table shows a comparison of similarity scores for the same example pair of poems in each case. It shows results for each metric, for item k -NN and for item based Smart Blend.

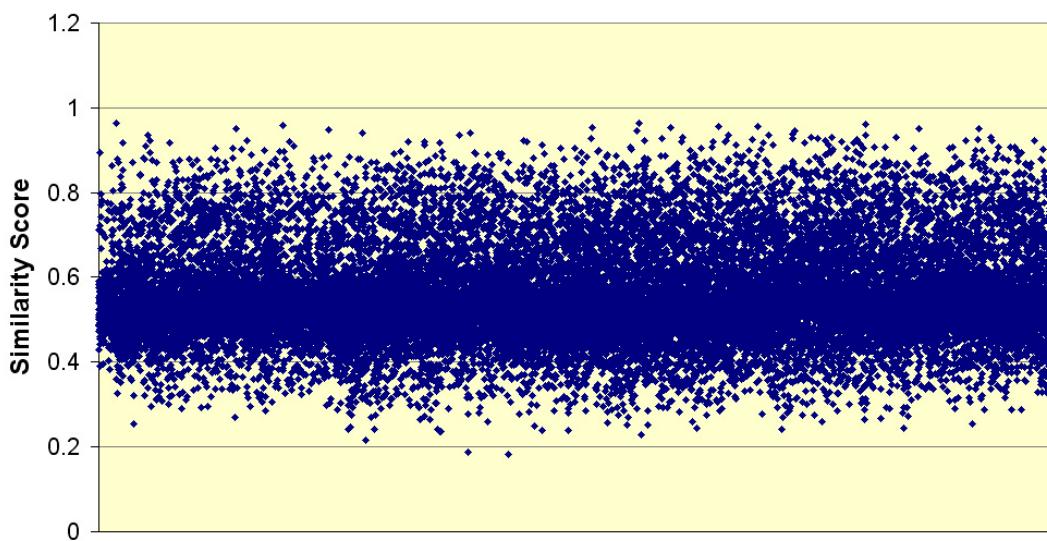
Metric	Case 1 2 shared ratings	Case 2 4 shared ratings	Case 3 6 shared ratings	Case 4 8 shared ratings	Case 5 10 shared ratings
Euclidean k-NN	0.67	0.8	0.46	0.89	0.59
Manhattan k-NN	0.5	0.5	0.125	0.5	0.125

Cosine k-NN	0.77	0.63	0.20	0.91	0.20
Average k-NN	0.67	0.8	0.46	0.89	0.59
Pearson k-NN	0	0	-0.5	0.88	-0.5
Euclidean SB	0.74	0.15	0.27	0.33	0.34
Manhattan SB	0.81	0.19	0.34	0.38	0.42
Cosine SB	0.99	0.59	0.88	0.99	0.96
Average SB	0.74	0.14	0.27	0.30	0.34
Pearson SB	0.87	0.28	0.53	0.97	0.84

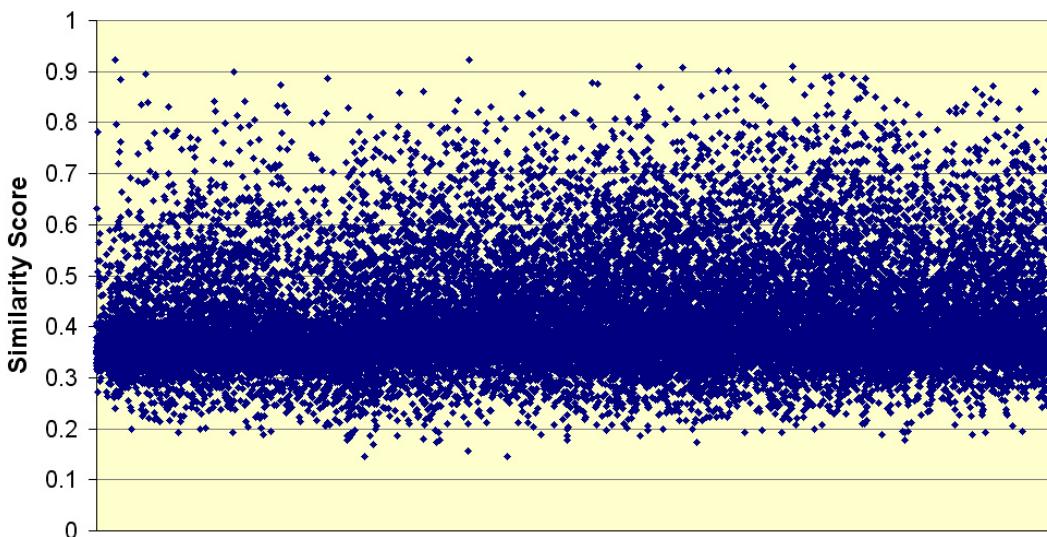
I have also included a scatter chart to show the variance in similarity scores achieved with each *Smart Blend* metric for a single poem (compared to all other poems). I chose a poem which had seven ratings, which was the average number per poem found in the data set. The selected poem also had a good mixture of known ratings, which reflect the makeup of the set overall (four zero ratings, two one ratings and one two rating).



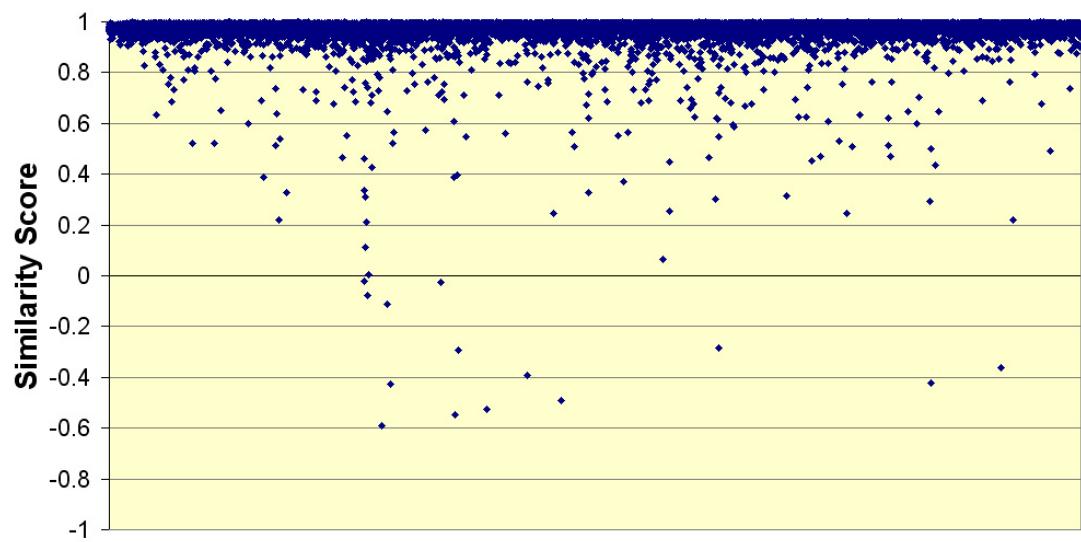
Manhattan Smart Blend Variance



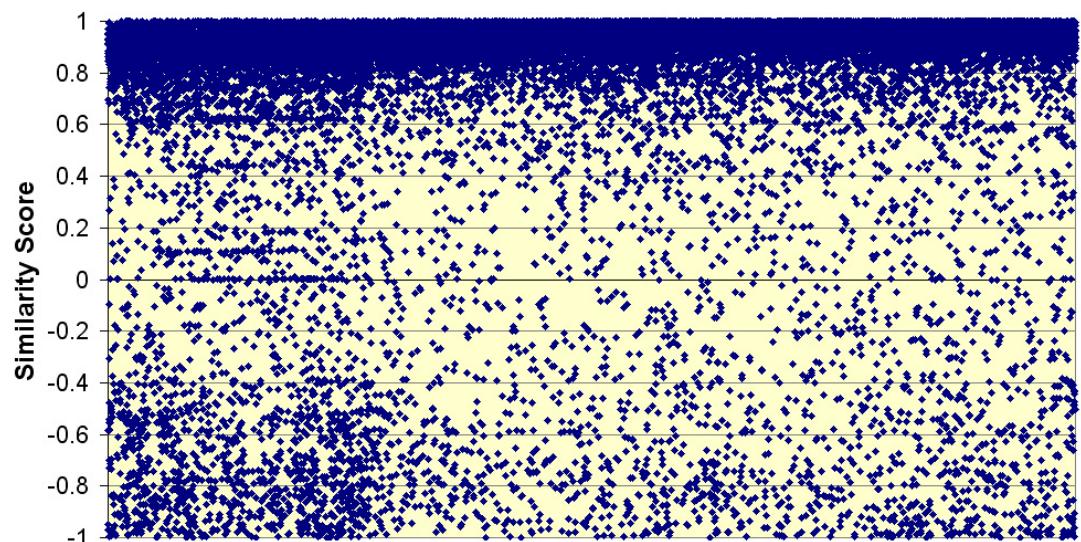
Euclidean Smart Blend Variance



Cosine Smart Blend Variance



Pearson Smart Blend Variance



Metric	RMSE
Euclidean	0.4933
Manhattan	0.5032
Cosine	0.4946
Average	0.5044
Pearson	0.4989

4.7 Program Design Decisions

Originally I used arrays, rather than vectors, throughout my program as I was aware that there may be a small performance advantage [30]. However, this added the burden of having to re-query the database each time I changed the input files (in order to find out the number of poems and members). Bearing in mind that I plan to run the offline tool regularly, it was not ideal to have to log into the database, gather these details, each time I wanted to generate new results. In light of this, I changed my arrays to vectors so that I could resize them dynamically. This also proved beneficial as I started to add options for running my program using different approaches. I could have all the different types of vectors which may be needed and initialise them to size zero. The program then only needed to expand the ones being populated for the approach being used. For example, the k -NN similarity vector is not needed if testing a pure SVD approach.

I attempted to avoid code bloat by having the metrics in a separate file and using templates to make a single function for each metric that was suitable for multiple different approaches (pure k -NN and Smart Blend). However, because these distance functions are being used repeatedly, the extra computational cost of making them universal slowed things down significantly. In the end it proved better to make the functions specific to the methods being used in order to ensure the minimum computational cost was incurred each time a similarity score was generated.

4.8 Generating Data from Predictions

Throughout the tuning process I tested my algorithms by creating predictions for member/poem combinations requested in the probe set. This method would be the most obvious way of generating final results; producing a prediction for each member, for each poem they haven't read (and didn't write). However, this would mean that any new data, or indeed new members, would have to wait until the offline tool was run again in order to receive predictions reflecting the new information. To work around this issue, I also wanted to consider the method used by *Amazon*; producing data about similar items and using that to generate predictions. This would mean that predictions given could seemingly change and respond immediately to new information from members. Each time a member takes an action to indicate a positive response to a poem; similar poems can be retrieved from the database and recommended if suitable. New poems however would have to wait until the result set was recomputed in order to be included. However, recently posted poems appear at the top of the *DU Poetry* listings and are therefore likely to receive lots of attention when first posted, so I didn't feel that not including them immediately in recommendations was a problem. I will now discuss how I implemented both approaches in my offline tool.

4.8.1 Recommended Poems (user-item predictions)

This appeared to be a straight forward way to produce recommendations. However, the main challenge was to filter the results to ensure that the amount of output data didn't get too large, whilst producing enough suitable recommendations for each member. Initially I set a minimum value for which the prediction had to achieve in order to be considered for the result set. With this set at 1.01 (just over the "like" value), the resulting files were 36.3mb with 2.16 million entries (no "follower" simulated ratings) and 52mb with 3 million entries ("follower" simulated ratings for all viewed poems). Along with producing large output files, the results only contained recommendations for 166 and 197 members respectively. I realised that this was due to predicted ratings being closely correlated with the average rating that a member had given to all poems they had rated. This average includes zeros when they have viewed a poem but not taken an action. I adjusted my method of generating predictions, so that a top n

number of highest value predictions for each member could be defined. The problem with this method was that for item based algorithms it had to store all predictions until the end of the process, in order to avoid having to re-compute the poem similarity vector multiple times (the program loops through each poem, then every member in relation to that poem). The program ultimately ran out of addressable memory before completion. I solved this issue by only saving predictions which were at least 10% better than the average rating given by that user (with an exception case for users with an average very close to the maximum rating). Anything less than this level of improvement is a bad candidate for a recommendation anyway, as it isn't significantly higher than their average poem rating. I then sorted the stored results for each member in order to output the top n results. With n set at 100, this significantly reduced the size of the output files to 3.3mb with 170,000 entries and 3.7mb with 190,000 entries, yet provided recommendations for 1892 and 1893 distinct members respectively.

4.8.2 Similar Poems (item-item similarity)

Outputting results for similar poems was somewhat easier to implement (for item based methods) as it could be done straight after creating the similarity vector for each poem. This meant there was no need to store the poem similarities for longer than normally expected. I set a high similarity score of 0.97 to cut down the amount of data being generated. This produced a 26.6mb file with 1.37 million entries, which included data for 11,404 poems. I was able to reduce the size of the data file, and lower the minimum required similarity score, by modifying my existing code. I used the method for creating a similarity vector when a number is specified for k (rather than considering all k -NN) as it performs a sort on the similarity vector to find the most similar neighbours. I included a variable for defining a maximum number of similar items to output for each poem. This restricted the size of the final output file, but also ensured that data for all poems were included (where other poems existed which met the minimum similarity score). Setting the minimum similarity score at 0.75 and a maximum number of 100 similar poems per poem, I generated an output file of 19.7mb with 981,000 entries. This included data for 16,000 different poems (nearly 5000 more than the previous output, and closer to the total of 25,000 poems in the data set). If a user based algorithm is chosen instead, the similarities between members are output at the time of generating the similarity vectors.

In order to output similarities for the opposite approach (for example, member similarities, when using an item based approach) I added an extra phase to the result generating procedure. I opted to make use of existing code by re-making the initial vectors in line with the opposite approach, since this only a few seconds to complete, and then looping through creating a similarity vector for each item, and outputting the most similar items, as outlined above.

4.9 Website Implementation

The on-site part of the development process involved Perl scripting to fit in with the existing website. I wrote subroutines to generate pages and communicate with the database, to log on-site test phase data and upload/download data from the database to interact with the offline tool.

4.9.1 Dealing with the Results Data

The recommendations data is uploaded to the existing *DU Poetry MySQL* database. As the results of my research on the inclusion of simulated “follower” data were inconclusive, I generated two different sets of member specific recommendations for testing. One with no “follower” simulated data (which achieved a low RMSE in testing) and one with simulated “follower” data for poems viewed on any date (which achieved the highest improvement over its average RMSE). I used Perl to insert the output from the offline tool into the recommendations database table, and a separate table into which the results for similar poems were inserted. In order to generate predictions from the similar poems data, I queried each member’s reading list entries and found the best matching poems to those on their list.

Total Entries (up to 50 per member from each table)	239956
Distinct Members Overall	2042
Distinct Members* (not including “follower” simulated ratings)	1892
Distinct Members* (including “follower” simulated ratings)	1893
Distinct Members* (from similar poems)	1205
Distinct Poems Overall	24251
Distinct Poems (not including “follower” simulated ratings)	15851
Distinct Poems (including “follower” simulated ratings)	17634
Distinct Poems (from similar poems)	16361

*Distinct members are those who have at least 6 suitable recommendations for testing (non duplicates).

It was encouraging to see that a wide range of poems were being recommended (nearly all of the 25841 in the data set). This satisfied my goal to include as many poems as possible in the recommendations given.

I reduced the sets down to a top 50 of each per member in order to perform some analysis on the level of duplication experienced amongst the highest ranked predictions of each type. I included a “duplicate” column in the final results table in order to track the level of duplication and at what point it occurred. Duplicate entries were not suitable for testing as I wanted to generate three unique lists, to properly test the differences between the three types of prediction data.

Total Duplicates (out of 239956 total entries)	12698 (5.29%)
Duplicates Between Poem Recommendations (“follower” sim and no “follower” sim)	9094
Duplicates incurred using similar poems (from member’s reading list entries)	2465

Duplicates incurred using similar poems (from member's "likes")	4092
Overwritten duplicates (duplicates appearing more than one)	2953

The level of duplication witnessed across the sets was around 5%, which shows that the results produced by each method are very different from one another.

For testing I chose to use six poems per list, for each of the three lists (and a fourth "control" list of randomly generated poems). Each member who took part in testing would be required to view 24 poems, and then select their favourite of the four lists. In total there were 1055 members for which a test set had been achieved (six entries suitable for each of the lists). There were a further 915 members for whom there were entries, but not enough to fill each type of list. I then analysed the prevalence of recommended poems for which the member was following the author poet.

	Poems by followed authors	Poems by un-followed authors
List 1 (not including "follower" simulated ratings)	1092 (1.1%)	94863 (98.9%)
List 2 (including "follower" simulated ratings)	2671 (3.1%)	84002 (96.9%)
List 3 (from similar poems)	505 (0.9%)	56823 (99.1%)

Although my concerns over the overall number of "followed" poet recommendations being high were unfounded, there was still an issue with regard to some specific members' recommendations.

	Number of members with > 3 recommendations for followed authors' poems
List 1 (not including "follower" simulated ratings)	1 (0.1%)
List 2 (including "follower" simulated ratings)	22 (1.2%)
List 3 (from similar poems)	1 (0.1%)

As expected, the results that were trained with "follower" simulated ratings included, contained a higher prevalence of followed authors' poems. Due to the numbers being low overall, and wanting to test the full implications of different data set ups, I decided not to filter these ratings out of the user evaluation process. My only reservation was that for several members, their entire recommendations for list 2 comprised of poems by author's they were following.

Next I wanted to look at the diversity of recommendations being made. I queried the database to find out how many members received three or more recommendations by the same author.

	Number of members with n recommendations by the same author		
	$n = 4$	$n = 5$	$n = 6$
List 1 (not including "follower" simulated ratings)	2	2	1
List 2 (including "follower" simulated ratings)	23	1	0
List 3 (from similar poems)	1	1	0

I couldn't find a definitive reason for why members were receiving this lack of variety in their recommendations, however they generally related to members with no "likes", or "likes" for poems mostly by the same authors. Of the 31 cases; 12 had no "likes", 5 liked poems by less than five different authors, 10 liked poems by less than ten different authors and 4 liked poems by more than ten different authors.

Finally, I deleted any recommendations for members who had less than the six required for testing each list. Since MySQL does not currently support deleting from a table using select from the same table in a sub-query, I created a temporary (connection based) table for storing member id's for members which did not meet the criteria.

```
CREATE TEMPORARY TABLE fordeletion

SELECT distinct r.memberid FROM final_rec r

WHERE (select count(*) from final_rec WHERE r.memberid =
memberid and followers = 1 and type = 0) < 6

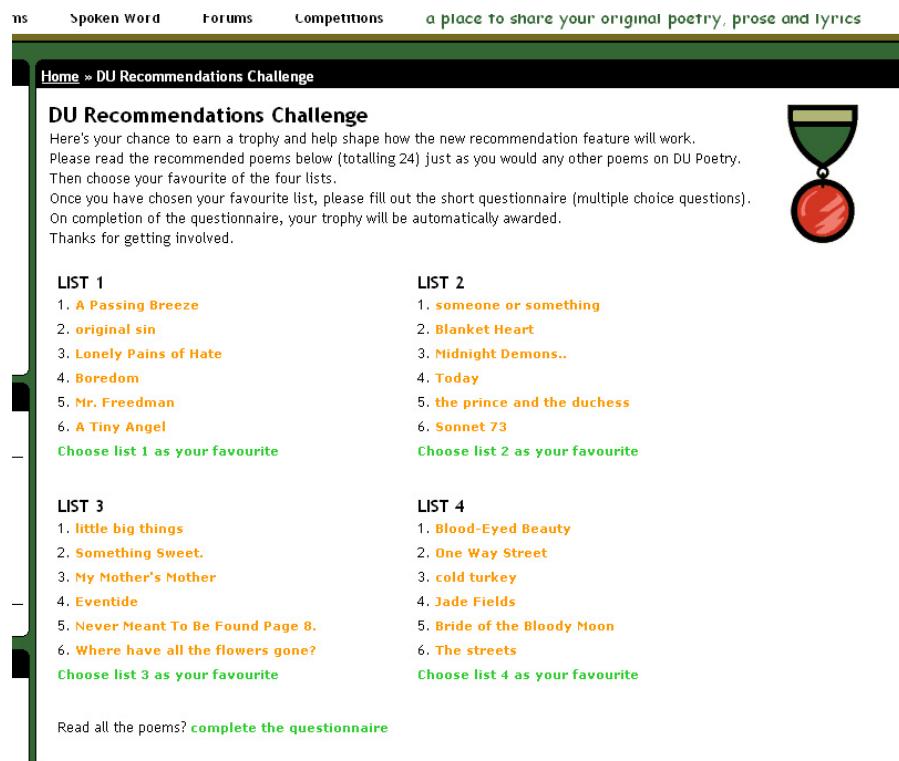
OR (select count(*) from final_rec WHERE r.memberid = memberid
and followers = 0 and type = 0) < 6

OR (select count(*) from final_rec WHERE r.memberid = memberid
and type > 0) < 6;
```

I then deleted everything from the recommendations table containing member id's in the "fordeletion" temporary table. Finally I added recommendations for each potential participant to form a "control" list, by generating random un-viewed poems, which were not already present in one of the three existing lists. Of the final data across the four lists there were a total of 29286 recommendations which included 14823 different poems. The poem recommended most often appears in 16 different members' test data. I think that overall this represented a good amount of variety in the poems being recommended.

4.8.2 Front-end for User Test Phase

I scripted a test page, accessible by all members who had enough data for testing. This provided links to their recommended poems, along with links for selecting their favourite list. After which they are invited to complete a multiple choice questionnaire (discussed in the conclusion section). I kept the design clean and in keeping with the rest of the *DU Poetry* website. In order to randomise the order in which the lists were shown to members, to curb the opportunity for members to discuss their choices and influence each other, I displayed the four lists in four different possible orders, based on member id.



The screenshot shows a web page titled "DU Recommendations Challenge". At the top, there is a navigation bar with links for "Home", "Spoken Word", "Forums", "Competitions", and a tagline "a place to share your original poetry, prose and lyrics". Below the navigation bar, the main content area has a dark header bar with "Home > DU Recommendations Challenge". The main content starts with a section titled "DU Recommendations Challenge" containing instructions about the challenge, including how to earn a trophy and the steps to participate. To the right of the text is a graphic of a gold medal with a red ribbon. The page then displays four sections, each labeled "LIST 1", "LIST 2", "LIST 3", and "LIST 4", each listing six poem titles. Below each list is a green link to "Choose list 1 as your favourite", "Choose list 2 as your favourite", "Choose list 3 as your favourite", and "Choose list 4 as your favourite" respectively. At the bottom of the page, there is a link "Read all the poems? complete the questionnaire".

Screen shot showing the recommendation engine test page for members

4.8.3 Front-end Final

The final recommendations page was generated from user-specific predictions using all “follower” simulated data for viewed poems, as this performed best in the member evaluation process (discussed later). If there are not enough unread recommendations to show, then further recommendations can be generated from the “similar poems” table, using a member’s current “likes” and reading list entries. This also means that recommendations can be made right from the very first “like” or reading list entry that a member makes. Along with poem recommendations, I also show a list of up to six similar members. This list is generated initially from the similar members table, and if there are not enough entries to show, it then attempts to find members with a similar writing style or level. This is done by searching the poems written by the viewer, and finding similar poems (using the “similar poems” table) and showing the authors of these poems. Poets the viewer is already following are not displayed.

The screenshot shows a web page titled "a place to share your original poetry, prose". The top navigation bar includes links for "Spoken Word", "Forums", "Competitions", and "Home". Below the navigation, a breadcrumb trail indicates the user is at "Home > Recommended Reading". The main content area is divided into two sections: "DU Recommendations" on the left and "Poets you may like..." on the right. The "DU Recommendations" section lists 18 poems with their titles and authors. The "Poets you may like..." section displays six small profile pictures corresponding to the poets listed in the recommendations section.

Poet	Image
The Eyes Of a Psychopath by Mike Amoroso (mike1974)	[Profile Picture]
torture me, i'll torture you by Sarah Louise Ellis (Blood_Merchant93)	[Profile Picture]
This is life ? by Maria (truth_is_freedom)	[Profile Picture]
Lord Of Shallows by MDT	[Profile Picture]
Gone by Graham Brodie (graham_brodie)	[Profile Picture]
The First Wave by Highway62	[Profile Picture]
The confines of a cave by Michael Granger (Sir-Writes-A-lot)	[Profile Picture]
the same page by Albylynch	[Profile Picture]
90's Kid by MDT	[Profile Picture]
To the Woods by Abra (Abracadabra)	[Profile Picture]
Portrait in Silhouette by heterodynemind	[Profile Picture]
No Regrets by Tess Stoops (forever-for_real)	[Profile Picture]
you and i by if-only-i-knew	[Profile Picture]
Lines of cold by Tredon56	[Profile Picture]
Visceral Voyage by Umi Chan (maikeru555)	[Profile Picture]
The Blue Rose by bluerose	[Profile Picture]
river remembered with Angharad by Graham Brodie (graham_brodie)	[Profile Picture]
A Voice from the Tower by Borderline1978	[Profile Picture]
Innocent thoughts... by Nik-Forever (emoboi24)	[Profile Picture]
Accidental Heart Attacks by naturemithya	[Profile Picture]

Screenshot showing the final member recommendations page.

Conclusion

5.1 Evaluation of Objectives

Throughout development I tuned and tested the algorithms I produced against the quantitative measure of RMSE, always seeking to lower the amount of error achieved. This gave me a benchmark by which to improve the accuracy of the system. However, the real measure of the success of my project can only come from the user's own experiences of how suitable the recommendations were. Despite the level of technical and mathematical detail involved in the task, the end result is very much about the social benefit that collaborative filtering can provide to the *DU Poetry* community. The overall objective of my task was to provide a tool which could be used to take *DU Poetry* forward in the future by helping the website to grow. I believe collaborative filtering can do this by strengthening the community, keeping the site interesting for members and helping like minded people find each other more easily.

The data set itself provided massive challenges due to the sparseness of the data and the noise incurred due to differences in the ways members' use implicit and explicit means to show their reactions to the poems they read. Despite this, I was able to combine two very different approaches in order to achieve a good level of improvement over using the average rating as a prediction; which could be likened to providing random recommendations. This ensured that in quantitative terms, the predictions made should certainly be an improvement over a random means of selection.

Originally I set out to try and find trends in member characteristics like age and location. This proved challenging for a number of reasons including; the lack of diversity among members and the fact that not all members supply this optional profile information. However, having incorporated into my program the means of carrying out this kind of analysis, I can continue to monitor this in the future as the data set grows and changes.

I considered different approaches for generating output, mindful of the desire to be able to produce recommendations that could adapt instantly to new members and new information. I tried the approach used by *Amazon* to produce data related to similarities, rather than user-specific predictions. I tested the two different approaches alongside each other during the member evaluation process, to see if there was any penalty in using this approach, with respect to the quality of predictions made. By using similarity information, instead of producing finite predictions, it will be less important to run the offline tool regularly in order to keep supplying new predictions.

I put a lot of focus on how best to use the existing data, as I felt this was a really key part of achieving the best possible results. It proved difficult to assess the benefits of including different types of data, as the statistical analysis of RMSE couldn't give the full picture; as I wasn't comparing like with like. However, I was able to be reasonably confident in the merits of including simulated comment based data. The results of including "follower" data were less conclusive. In light of this, I chose two different approaches (no "follower" simulated data and "follower" simulated data for all viewed poems) and included them both in the member evaluation process. Despite achieving a very low RMSE when including simulated data for un-viewed poems, I discounted this method because it put an overwhelming emphasis on simulated data, which diluted the explicit actions taken by members to a level which I felt was unreasonable.

5.2 Statistical Evaluation

I incentivised members to take part in the user evaluation process, with the promise of a trophy for those who completed it. In order to receive the award they had to read all 24 of their recommendations, and complete a questionnaire. I ran queries to check how members reacted to poems recommended to them, in terms of which actions they took on the selected poems. In total 135 members took part, 81 of which completed the challenge. I was surprised to find that commenting levels were very low (compared with typical behaviour) across each of the four lists, and believe this may be due to the nature of the challenge; participants eager to complete it (and acquire the trophy) rather than spend time leaving comments. All three lists produced from recommendations performed significantly better than the random control group in terms of “likes” and reading list entries. Both list two and three also equalled or exceeded the percentage occurring on average for these actions. Bearing in mind that member’s may typically spend time reading poems by authors they already know about and like, the control group may be a better indication of a truly average reaction to a random poem.

	Poem Views	Comments (percentage of views)	Likes (percentage of views)	Reading List Entries (percentage of views)
Recommendation Test Phase	2408	82 (3.3%)	450 (18.1%)	51 (2.0%)
List 1 (user specific recommendations not including “follower” simulated ratings)	588	21 (3.6%)	99 (16.8%)	6 (1.0%)
List 2 (user specific recommendations including “follower” simulated ratings)	596	22 (3.7%)	139 (23.3%)	22 (3.7%)
List 3 (from similar poems data inc. “follower” simulated ratings)	615	19 (3.1%)	135 (22.0%)	21 (3.4%)
List 4 (random control group)	609	20 (3.3%)	77 (12.6%)	2 (0.3%)
Average Levels for comparison (post “like” feature launch)	112153	23.3%	20.4%	3.4%

5.3 User Opinion

I included a questionnaire in the user evaluation process in order to further gauge users' reactions to recommendations, and also pose some questions which would help me decide how to develop the feature further in the future. The following screen shot shows the questionnaire supplied to participating members.

Home > Questionnaire

There are seven questions, please complete all of them. Results are handled anonymously.

1. Which of the following methods do you use for finding poems on DU Poetry?

	Rarely	Sometimes	Often
Random poem link	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
General/all poems listings	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Category specific poem listings	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Sorted listings (e.g. popular poems in last 30 days)	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
"My Updates" links to poems published by poets you follow	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Poems by a specific member (from link on their profile, comment or forum post)	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>

2. How often do you find a poem you enjoy reading on DU Poetry?

Very Often	<input type="radio"/>
Often	<input type="radio"/>
Sometimes	<input type="radio"/>
Not Often	<input type="radio"/>
Hardly Ever	<input type="radio"/>

3. Which of the following statements best applies to you?

I enjoy reading other peoples' poems more than writing my own.	<input type="radio"/>
I like equally reading other peoples poems and writing my own.	<input type="radio"/>
I prefer writing my own poems to reading other peoples' poems.	<input type="radio"/>

4. If recommendations could be improved by adding an anonymous "dislike" button for poems, would you use it?

Yes No

5. Should the number of dislikes a poem has received be shown?

Yes (to everyone)	<input type="radio"/>
Yes (to author only)	<input type="radio"/>
No (to nobody)	<input type="radio"/>

6. Would you like to be shown members who are similar to you?

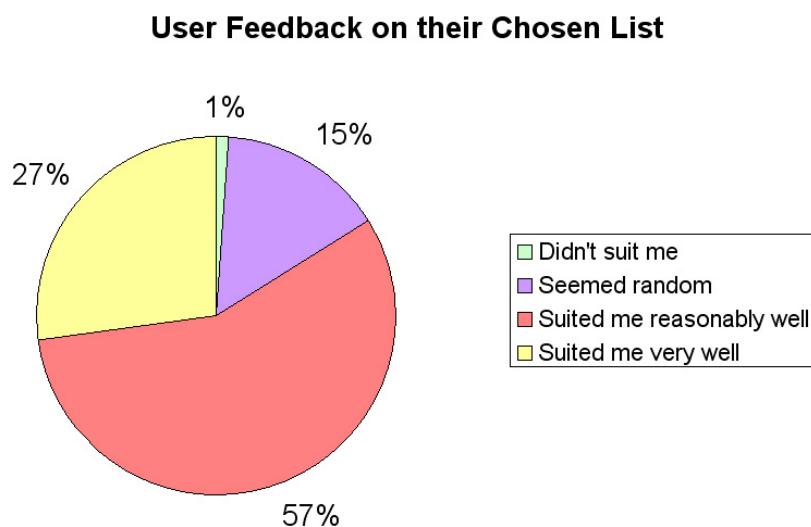
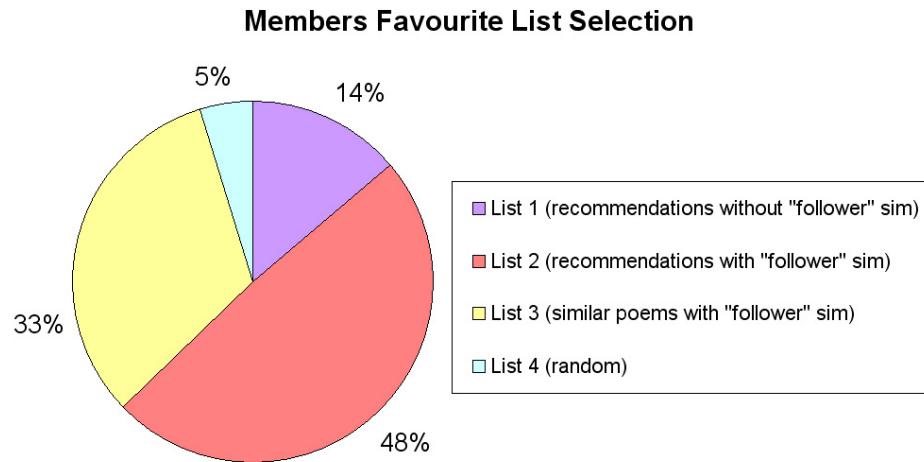
Yes members with a similar taste in poems to me	<input type="radio"/>	No, not interested	<input type="radio"/>
Yes members with a similar writing level/style to me	<input type="radio"/>	No, not interested	<input type="radio"/>

7. Please consider the list of poem recommendations which you chose to be the best. Which statement best reflects your opinion of these recommendations?

These recommendations suited me very well (much better than random browsing)	<input type="radio"/>
These recommendations suited me reasonably well (slightly better than random browsing)	<input type="radio"/>
These recommendations seemed random (no better than random browsing)	<input type="radio"/>
These recommendations didn't suit me at all (worse than random browsing)	<input type="radio"/>

Thanks for taking the time to complete this questionnaire.

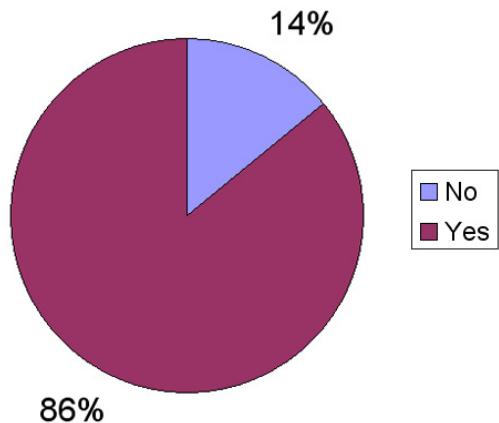
Before completing the questionnaire members were required to select their favourite of the four lists. The resulting choices and member's opinions of their selected list were as follows:



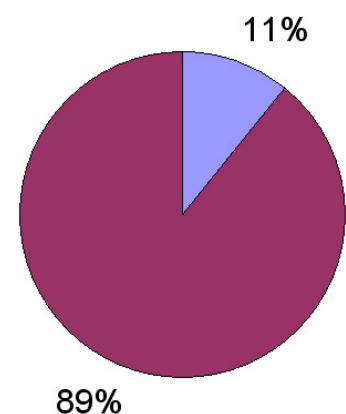
The statistics for which list was chosen mirror the results I got in the previous section, with lists two and three (those including “follower” simulated data) doing best. The user-specific recommendations in list two out performed those generated from the “similar poems” table. Overall 84% of respondents perceived their favourite list to have provided recommendations which were better than random. However, a small number of respondents chose the random list as their favourite, this could be a reflection of poor recommendations in these cases, or simply that the random selections provided happened to be exceptionally suitable.

I also asked users if they were interested in being shown members who were similar to them. The results are shown in the following pie charts.

Show me members with a similar writing style to me?



Show me members with a similar taste in poems to me?



The results for both questions were closely related, and the majority of respondents were interested in seeing members similar to them. In light of this, I included a selection of similar members alongside the poem recommendations on the final recommendation engine webpage. First it searches the “similar members” table which gives results for members with similar tastes. If it doesn’t find enough matches then it goes on to look for members with a similar writing style, using the “similar poems” table together with details of poems written by the member.

5.4 Lessons Learned

Although, it proved frustrating to carry out the majority of testing and development on a five year old 32-bit laptop, I feel that helped me think carefully about the code I was writing and ultimately adopt efficient programming habits. When my machine ran out of memory, it forced me to re-evaluate my code, and consider different approaches, sometimes forfeiting a small amount of extra execution time, in order to ensure the solution was scalable and met the needs of the data set. It also helped me develop a good understanding of the capabilities of different types of hardware and the importance of having a machine capable of the task at hand. While optimising a program where possible is important, a hardware solution could potentially be far more efficient than what is possible through coding techniques alone [31]. To generate a full result set on my machine takes approximately 4.5 hours, when I tested this on a slightly newer 64-bit machine this reduced to less than 3 hours. I would expect that with an up-to-date and more powerful computer this would reduce further still.

Version control proved very important in this project, as I was testing different ways of doing tasks, I ended up with a lot of different versions of my code. Often I was running tests over-night with different settings and code, so it was important to keep track of which versions were being used, in order to produce comparative results and not lose track of updates. Some of the methods I used to generate statistics are now obsolete; for example to test access times of different storage methods, or tuning variables which in the end didn’t yield the anticipated RMSE improvement. However, where possible, I tried to leave in support for different approaches in cases where it didn’t impact upon the execution time of the optimum algorithms, this was in case I want to re-test them in the future as the data set grows and changes.

I also learned the importance of not losing sight of individuals when embarking on a

large scale collaborative process. *DU Poetry* is still a relatively small website, with a core of active members who are very loyal and involved. The user testing process saw a popular and active member, who appeared to have plentiful data regarding their “likes”, excluded from the evaluation because they did not have enough suitable entries for testing. In a community based on people interacting with one another, members can easily feel disenfranchised if they have been left out of the process, and feel they are being treated differently. Striking the balance between generating enough data to provide recommendations for as many members as possible, while still keeping the file sizes and the amount of rows in the database at an acceptable level, is something which I continue to actively monitor.

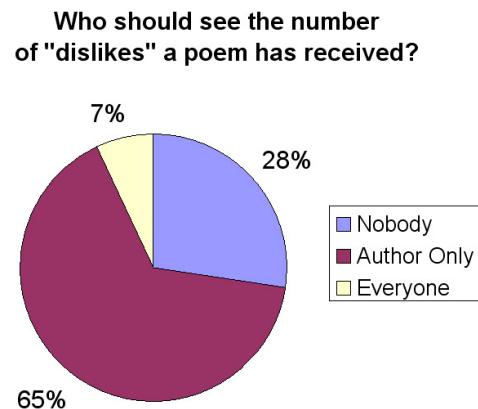
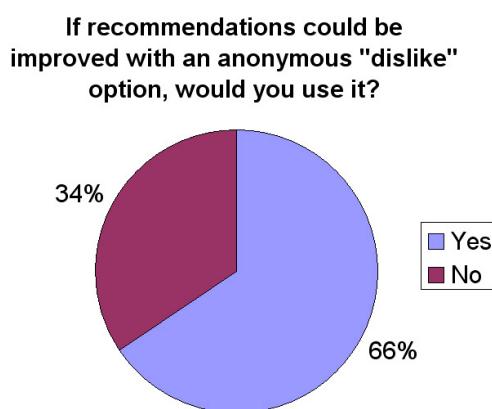
This project was ambitious for me with regards to the complexity of the mathematical components, having not studied maths since GCSE, and having not taken the information retrieval module option. However, the challenges involved proved very rewarding and I now have a new found understanding of mathematical notation and can make sense of some of the complex research papers published by teams during the *Netflix Prize*, which originally baffled me.

Along my technical competency improving, this process also allowed me to analyse the *DU Poetry* dataset in depth, and as a result, gain a much better understanding of how members use the website. Going forward I feel that I have much improved knowledge with which to make informed decisions about the direction of the development of the website and the *DU Poetry* community.

5.5 Future Development

My recommendation engine is a new feature for *DU Poetry*, and it may take some time to fully access how members respond to it, how often they use it and how much of an asset it ultimately becomes. Although I currently contain the feature on its own page, I intend to trial providing recommendation in other parts of the site; for example, similar poems displayed on a poem’s page, or similar members displayed to users alongside their updates feed.

Within the user evaluation questionnaire, I posed a couple of questions regarding a “dislike” option for poems. I wanted to see if this is something members would use, should it be introduced and whether the number of “dislikes” a poem has received should be made public or not.



Around a third of respondents said they would use a “dislike” option if it were available. Since the launch of the “like” feature, the amount of useful data collected continues

to grow, eventually I plan to try and move away from using (or place less emphasis on) zero value data relating to poem views, as it has a capacity for noise due to the different ways members use the website (for example, some make use of the like feature more than others). Adding a “dislike” option may be a good choice for increasing the amount of useful data collected. I added in the question about whether the number of dislikes should be shown. Even though the majority of authors wanted to know, in reality I would have to think very carefully before making this information public. Members may also be more reluctant to use the “dislike” option (even if it were to be anonymous as planned) if they knew it may impact negatively on the author. It could also be open to abuse.

Overall I feel I have achieved the aims set out in my proposal, and there has been a lot of interest and positive feedback from members since the recommendations page launched. I intend to continue to use collaborative filtering techniques in the future in order to further personalise user experiences, and develop *DU Poetry* into a truly social community.

References

- [1] Deep Underground Poetry, <http://deepundergroundpoetry.com>
- [2] Deep Underground Poetry, *Deep Underground Poetry Poll: Does it matter to you whether people read and leave feedback on your poetry?* <http://deepundergroundpoetry.com/polls/10/>
- [3] GS Poetry, <http://www.gspoetry.com>
- [4] GotPoetry, <http://www.gotpoeuy.com>
- [5] Deep Underground Poetry, *Deep Underground Poetry Poll: Which do you most often comment on?*, <http://deepundergroundpoetry.com/polls/12>
- [6] Read Write Web, *Collaborative Filtering: Lifeblood of the Social Web*, http://www.readwriteweb.com/archives/collaborative_filtering_social_web.php
- [7] Geeking with Greg, 35% of sales from recommendations, <http://glinden.blogspot.com/2006/12/35-of-sales-from-recommendations.html>, 2006
- [8] *Netflix Prize: Home*, www.netflixprize.com
- [9] Yehuda Koren and Robert Bell, *Advances in Collaborative Filtering*
- [10] Wikipedia, *Netflix Prize*, http://en.wikipedia.org/wiki/Netflix_Prize
- [11] Yehuda Koren, *Factor in the Neighbors: Scalable and Accurate Collaborative Filtering*, Yahoo! Research, 2009
- [12] Badrul Sarwar, George Karypis, Joseph Konstan, John Riedl, *Incremental Singular Value Decomposition Algorithms for Highly Scalable Recommender Systems*, University of Minnesota, 2002
- [13] Toby Segaran, *Programming Collective Intelligence: Building Smart Web 2.0 Applications*, O'Reilly, 2007.
- [14] Simon Funk, *Netflix Update: Try This At Home*, <http://sifter.org/~simon/journal/20061211.html>, 2006.
- [15] M. W. Berry, S. T. Dumais, G. W O'Brian, *Using Linear Algebra for Intelligent Information Retrieval*, SIAM Review, 1994
- [16] Wikipedia, *Overfitting*, <http://en.wikipedia.org/wiki/Overfitting>
- [17] Arkadiusz Paterek, *Improving regularized singular value decomposition for collaborative filtering*, Warsaw University, 2007
- [18] Robert Bell, Yehuda Koren, Chris Volinsky, *The BellKor solution to the Netflix Prize*, AT&T Labs – Research, 2007
- [19] G. Linden, B. Smith, J. York, *Amazon.com recommendations: Item-to-item collaborative filtering*, IEEE Internet Computing, 2003.
- [20] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, And John T. Riedl, *Evaluating Collaborative Filtering Recommender Systems*, University of Minnesota and Oregon State University
- [21] Netflix Prize FAQ, <http://www.netflixprize.com/faq>
- [22] The Official YouTube Blog, *Five Stars Dominate Ratings*, <http://youtube-global.blogspot.com/2009/09/five-stars-dominate-ratings.html>, 2009

- [23] Hermetic Word Frequency Counter, <http://www.hermetic.ch/wfc/wfc.htm>
- [24] Xavier Amatriain, Josep M. Pujol, and Nuria Oliver, *I like it... I like it not: Evaluating User Ratings Noise in Recommender Systems*, Telefonica Research, 2009
- [25] Stack Overflow, *Hash function for a pair of long long?*, <http://stackoverflow.com/questions/738054/hash-function-for-a-pair-of-long-long>
- [26] Boost C++ Library, <http://www.boost.org>
- [27] Microsoft Support, *Comparison of 32-bit and 64-bit memory architecture for 64-bit editions of Windows XP and Windows Server 2003*, <http://support.microsoft.com/kb/294418>
- [28] *Implicit Alternating Least Squares SVD*, <http://web4.cs.ucl.ac.uk/staff/T.Jambor/blog/>
- [29] *Blending Netflix predictions*, <http://elf-project.sourceforge.net/netflixBlending.html>
- [30] *Pseudo Associative, Performance of STL vector vs plain C arrays*, , <http://assoc.tumblr.com/post/411601680/performance-of-stl-vector-vs-plain-c-arrays>
- [31] *Performance Programming*, <http://sol-biotech.com/code/PerformanceProgramming.html>, Keith Oxenrider, 2004