Efficient Collaborative Filtering with Locality-Sensitive Hashing

Daniel Goldbach

Abstract

Recent research into recommender systems has yielded three useful classes of techniques: content-based recommendation, collaborative filtering, and dimensionality reduction. The second of these, collaborative filtering, involves pairing an item t to user u based on a preference shown to t by other users similar to u. This involves searching for similar users to u using a nearest neighbours search. Unfortunately, conducting a nearest neighbours search in a high-dimensional space is prohibitively expensive. Instead, we can use *approximate* nearest neighbour searches to efficiently find many (but not necessarily all) nearest neighbours; in practice, this is sufficient. I investigate the application of traditional locality sensitive hashing techniques to this approximate nearest neighbour search by experimenting with the Netflix Prize data set. I conclude that locality sensitive hashing is a viable approach for efficient collaborative filtering.

1 Motivation

The advent of digital vending through online services such as eBay, Amazon, and Netflix has granted to consumers the opportunity to purchase from an enormous selection of stock. Traditional marketing solutions no longer suffice; modern businesses must leverage new technologies to remain competitive. In particular, the literature has identified the *long tail* problem: the fact that popular items now make up only a tiny proportion of the available stock. Rather than uniformly marketing these 'hits' to the entire customer base, it is more profitable to match items to individual customers based on the items' perceived *utility* to those customers. Recommender systems are computer programs that attempt to discover these matchings.

The most widely studied class of recommender systems is collaborative filtering. Collaborative filtering, as the name suggests, finds potential item matches for user u by considering existing item matches for users similar to u – *neighbours* of u. In the absence of user-provided metadata, we can find neighbours of u simply by identifying other users whose existing matches (e.g. purchases, item ratings, product views) are similar to u's own. To draw an analogy, imagine if we have a friend named Thomas who enjoys 1960's French romances and quirky English comedies. We can recommend movies to Thomas by simply finding 1960's French romances and quirky English comedies that he hasn't seen – this corresponds to the content-based approach to recommender systems. Alternatively, if we have another friend called Jane who *also* likes 1960's French romances and quirky English comedies, we can instead ask her to recommend to Thomas other movies that she enjoys – this corresponds to the collaborative filtering approach. Of course, in this scenario we begin with the knowledge that Jane has similar taste to Thomas. A naive algorithm that finds candidates with similar taste to Thomas involves comparing the list of movies enjoyed by Thomas

	t_1	t_2	t_3	t_4	t_5
u_1		3		1	4
u_2			5	5	4
u_3	1	3		2	
u_4			4		
u_5	1	4			

Fig. 2.1: An example utility matrix. Each utility value indicates a user's rating for an item. Note the sparsity of the matrix – only 1% of the utility matrix cells in the Netflix Prize dataset contain values.

to the list of movies enjoyed by each candidate. If the total number of movies in consideration is large, conducting these comparisons is expensive.

We require a means of *reducing the dimensionality* of the space. Rather than comparing *all* of the movies enjoyed by Thomas to *all* of the movies enjoyed by each candidate, it would be useful to generate a compressed representation of each person's taste in movies: a *signature* of their taste. This way, we can determine whether two users have similar taste by simply comparing their taste signatures.

This notion forms the basis of collaborative filtering through locality-sensitive hashing.

2 Similarity and Utility

The *utility matrix* represents the perceived *utility* or 'usefulness' of each item to a user. Each utility value is calculated based on information given by the user (e.g. an item rating, a "thumbs up", or a purchase) and may be normalised based on other factors (e.g. the user's mean rating or the item's mean rating). The recommendation problem thus consists of predicting missing entries in the utility matrix.

Successful collaborative filtering requires us to define an effective similarity function between two users. The similarity function will depend on the format of utility values; Linden et al. [4] for example, have success using cosine similarity where utilities are boolean values indicating item purchase. This is somewhat analogous to the usage of cosine similarity for document similarity estimation in information retrieval.

The Netflix Prize dataset uses 'star' ratings (integers from 1 to 5) given by users to films. I initially considered using a threshold method to translate these ratings into boolean values. For example, a star rating of 3, 4 or 5 could translate to a 1 to indicate the user's preference for that film, and other ratings or a lack of a rating could translate to 0. The resulting set membership interpretation of utility invites the use the Jaccard similarity, Dice similarity, Hamming distance etc. I eventually decided against this utility representation for a number of reasons:

- A significant amount of information gets lost in translation. We would a prefer a method that only reduces information when absolutely necessary.
- There is no intuitive mapping from star ratings to boolean values. Even a perfectly chosen threshold value will still consider missing values equivalent to negative ratings, when the two are clearly disjoint notions.

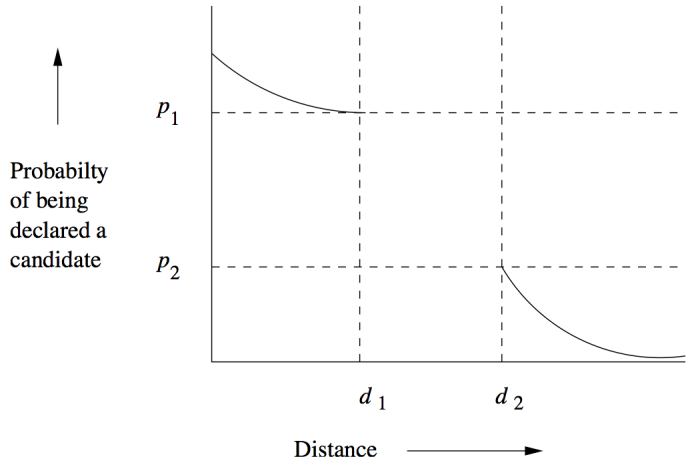

Fig. 3.1: Behaviour of a (d_1, d_2, p_1, p_2)-sensitive function (courtesy of Rajaraman and Ullman [5])

An alternative approach utilises statistical correlation measurements to compare users. These metrics are designed to test similarity between data sets, so intuitively they fulfil our requirement. I experimented with the Pearson and Spearman correlation coefficients[1]. Although both metrics produced reasonable similarity estimates, computation of these coefficients is a linear time process that does not scale to high-dimensional data. The treatment of missing values (i.e. the films unrated by each user) here also remains an unsolved problem.

I eventually decided that cosine similarity was the metric most suited to this problem. I was initially unsure about its efficacy for comparing numerical vectors (knowing only of its use for comparing logical vectors in IR), but Rajaraman and Ullman [5] recommend its use. Unlike the statistical metrics mentioned above, cosine similarity can be estimated using sub-linear methods.

In fact, I later discovered that the cosine similarity between two vectors containing centered data (data that has been normalised to have mean 0 and standard deviation 1) is exactly equal to their Pearson correlation coefficient. (Lee Rodgers and Nicewander [3]) My data is centered so my choice of cosine similarity is justified.

3 Theory of Locality-Sensitive Hashing

Introduction

A boolean (d_1, d_2, p_1, p_2)-sensitive locality-sensitive hash function $f : \mathbb{R}^M \to \{0, 1\}$ with respect to some distance function $D : \mathbb{R}^M \times \mathbb{R}^M \to \mathbb{R}$ satisfies two properties:

- If $D(a, b) \leq d_1$ then $\Pr[f(a) = f(b)] \geq p_1$
- If $D(a, b) \geq d_2$ then $\Pr[f(a) = f(b)] \leq p_2$

If we have a mutually independent random set of these hash functions $F = \{f_0, \ldots, f_{k-1}\}$ (a "family" of hash functions) we may define the "AND-equality" operator $\overset{\wedge}{=}$ in the expression

[1] The Spearman coefficient is the Pearson coefficient applied on the ranks of datapoints in the data set.

$F(a) \overset{\wedge}{=} F(b)$ to mean that $f_i(a) = f_i(b)$ for all $i = 0..k-1$. Similarly we may define the "OR-equality" operator $\overset{\vee}{=}$ in $F(a) \overset{\vee}{=} F(b)$ to mean that $f_i(a) = f_i(b)$ for at least one $i = 0..k-1$. It follows that

- If $D(a,b) \leq d_1$ then $\Pr\left[F(a) \overset{\wedge}{=} F(b)\right] \geq p_1^k$
- If $D(a,b) \geq d_2$ then $\Pr\left[F(a) \overset{\wedge}{=} F(b)\right] \leq p_2^k$

This has the effect of *lowering* our probability thresholds while maintaining d_1 and d_2 fixed. Importantly, the rates at which probabilities are lowered are inversely proportional to their original values. For instance, using $k = 3$, the probabilities 0.8 and 0.4 are lowered to 0.512 and 0.064 respectively.

In a similar fashion, the use of OR-equality *increases* our probability thresholds in a manner that affects larger probability values more than smaller values.

- If $D(a,b) \leq d_1$ then $\Pr\left[F(a) \overset{\vee}{=} F(b)\right] \geq 1 - (1-p_1)^k$
- If $D(a,b) \geq d_2$ then $\Pr\left[F(a) \overset{\vee}{=} F(b)\right] \leq 1 - (1-p_2)^k$

Again using $k = 3$, we see that probabilities 0.8 and 0.4 increase to 0.992 and 0.784 respectively.

If we instead begin with an ordered set of l mutually independent hash function families $\overline{F} = \{F_0, \ldots, F_{l-1}\}$ we can apply OR-equality over AND-equality, with the effect of simultaneously decreasing low probabilities and increasing high probabilities! To clarify, we begin with

$$\overline{F} = \{\{f_{0,0}, f_{0,1} \ldots, f_{0,k-1}\}, \{f_{1,0}, f_{1,1}, \ldots, f_{1,k-1}\}, \ldots, \{f_{l-1,0}, f_{l-1,1}, f_{l-1,k-1}\}\}$$

And we define OR-equality over this set of sets to mean

$$\overline{F}(a) \overset{\vee}{=} \overline{F}(b) \iff \exists i \in \{0,1,\ldots,l-1\}\, \forall j \in \{0,1,\ldots,k\}\, f_{i,j}(a) = f_{i,j}(b)$$

It follows from our earlier observations about the behaviour of the AND-equality operator and the OR-equality operator that

- If $D(a,b) \leq d_1$ then $\Pr\left[\overline{F}(a) \overset{\vee}{=} \overline{F}(b)\right] \geq 1 - \left(1 - p_1^k\right)^l$
- If $D(a,b) \geq d_2$ then $\Pr\left[\overline{F}(a) \overset{\vee}{=} \overline{F}(b)\right] \leq 1 - \left(1 - p_2^k\right)^l$

In other words, this composition of OR-equality with AND-equality allows us to *amplify* a (d_1, d_2, p_1, p_2)-sensitive hash function to a $\left(d_1, d_2, 1 - \left(1 - p_1^k\right)^l, 1 - \left(1 - p_2^k\right)^l\right)$-sensitive hash function.

Random Hyperplanes

Consider two vectors $\mathbf{a}, \mathbf{b}$ in $\mathbb{R}^M$. Although they exist in an M-dimensional space, they both lie on the same plane. If we randomly choose a hyperplane P in $\mathbb{R}^M$ passing through the origin, P must intersect with this plane in at least one position. The probability that P passes between $\mathbf{a}$ and $\mathbf{b}$ – i.e. P's angle with $\mathbf{a}$ is in $(0, \pi/2)$ and P's angle with $\mathbf{b}$ is in $(-\pi/2, 0)$, or vice versa – is linearly proportional to the angle between them. If we have a boolean hash function $f_P(\mathbf{x})$

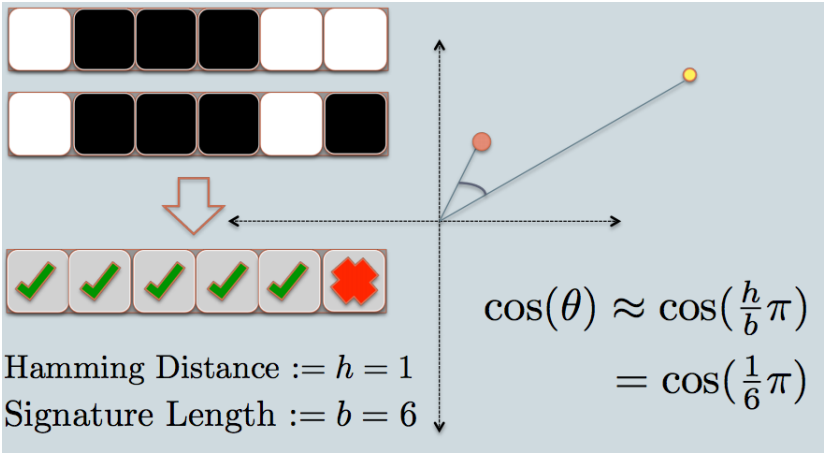

Fig. 3.2: Intuition behind the random hyperplane technique for LSH (courtesy of Van Durme and Lall [7])

that outputs 1 iff P passes to the left of $\mathbf{x}$, and the angle between $\mathbf{a}$ and $\mathbf{b}$ is θ, it follows from basic geometric and probabilistic principles that $\Pr[f_P(\mathbf{a}) \neq f_P(\mathbf{b})] = \frac{\theta}{\pi}$ and by complement that $\Pr[f_P(\mathbf{a}) = f_P(\mathbf{b})] = 1 - \frac{\theta}{\pi}$. In other words, f_P is a $(\theta_1, \theta_2, 1 - \frac{\theta_1}{\pi}, 1 - \frac{\theta_2}{\pi})$-sensitive function. Furthermore, the cosine of the angle between $\mathbf{a}$ and $\mathbf{b}$ (i.e. the cosine similarity of $\mathbf{a}$ and $\mathbf{b}$) is given by $\cos(\pi - \pi \Pr[f_P(\mathbf{a}) = f_P(\mathbf{b})])$.

In practice, rather than explicitly generating random planes we instead generate random vectors. The sign of the dot product of a random vector $\mathbf{v}$ with vector $\mathbf{a}$ is a bijection onto $f_P(\mathbf{a})$ where P is the plane perpendicular to $\mathbf{v}$ that passes through the origin. Indeed it is sufficient to generate random vectors whose components are all 1 or -1.[2]

It is also convenient to represent the outputs of a set of hash functions as a binary string. For instance, if we have hash functions $F = \{f_{P_1}, f_{P_2}, f_{P_3}\}$ with $f_{P_1}(\mathbf{a}) = 0$, $f_{P_2}(\mathbf{a}) = 1$, $f_{P_3}(\mathbf{a}) = 1$ we can concisely notate these results simply as `011`. We call this binary string the *sketch* of $\mathbf{a}$. Intuitively, $F(\mathbf{a}) \overset{\wedge}{=} F(\mathbf{b})$ iff the sketches for $\mathbf{a}$ and $\mathbf{b}$ match completely, and $F(\mathbf{a}) \overset{\vee}{=} F(\mathbf{b})$ iff the sketches for $\mathbf{a}$ and $\mathbf{b}$ match in at least one position.

A Nearest Neighbours Data Structure

Using the LSH techniques described above with sketches of length kl, we can determine with reasonable probability and in time complexity $O(kl)$ whether the cosine similarity between two vectors is below θ_1 or exceeds θ_2. However, if we are to find neighbours to some vector $\mathbf{v}$, we still need to loop through all $N-1$ other vectors to compute their approximate cosine similarity, leading to an $O(Nkl)$ algorithm. N, being the number of available items in the data set, is potentially very large. If possible, we would like a nearest-neighbour algorithm whose query complexity is independent of N.

For this purpose we will develop a data structure reminiscent of the inverse index in information

[2] I was initially skeptical of this claim by Rajaraman and Ullman [5] because it is not mathematically justified. However, following tests I found that empirically it appears to hold true.

retrieval. The index consists $2^k l$ key-value pairs. Each key is a tuple (i, s) where $0 \leq i < l$ and s is a binary string of length k, and each value is a set of element IDs. The element with vector $\mathbf{v}$ is found in the set with key (i, s) iff the substring of $\mathbf{v}$'s sketch in indices $[i, i + k)$ is equal to s. In order to query for the nearest neighbours of $\mathbf{v}$ we simply take the union of all the sets with keys (i, s) for each $i \in \{0, 1, \ldots, l - 1\}$, s being the i-th consecutive k-length substring in $\mathbf{v}$'s sketch. The probability that a particular neighbour of $\mathbf{v}$ (a "neighbour" being a vector that makes an angle with $\mathbf{v}$ of less than θ_1) is in the result set of this operation is $1 - (1 - p_1^k)^l$, and the probability that a particular "distant" vector (a vector that makes an angle with $\mathbf{v}$ of *more* than θ_2) is in the result set of this operation is $1 - (1 - p_2^k)^l$.

Choosing k, l

Recall that increasing k slightly reduces the probability of a true positive match and greatly reduces the probability of a false positive match. Inversely, increasing l greatly increases the probability of a true positive match and slightly increases the probability of a false positive match. An increase in k thus corresponds to an increase in precision (albeit sacrificing some recall) whereas an increase in l corresponds to an increase in recall (albeit sacrificing some precision). Although our LSH family becomes arbitrarily more precise and thorough as we increase k and l, we also incur a decrease in overall efficiency as many components of the system depend on k and l:

- The nearest neighbours index has space complexity proportional to $2^k l$.
- The time complexity of computing a sketch is proportional to kl.
- The time complexity of indexing a vector into the nearest neighbours index is proportional to kl.
- The time complexity of querying the nearest neighbour index for neighbours of some vector is proportional to kl.

Furthermore, improperly chosen values of k and l will result in an imbalance between precision and recall. When k is too large, too few neighbours will be found. When l is too large, too many false positives will be returned.

Through experimentation I found that suitable values k lie within the range 8-15 and appropriate values of l lie within 70-300. These values provide an appropriate balance between precision and recall. Using these values, the algorithm generally returns a result set of 300-1000 users, of which approximately half are true neighbours. There are over 433,000 users in the entire dataset.

4 Method

Dataset Overview

The Netflix Prize dataset consists of a training set and a test set (the 'probe' set). Both data sets consist of a list of quadruples of the form (movie ID, user ID, rating, date of rating).[3] Together, the training set and test set describe approximately 433,000 users, 17,770 films and 100,000,000 ratings. The size of the test set is 1.5% of the size of the training set.

[3] Although several Netflix Prize winners found date of rating to be useful in their models, I did not have time to explore its impact. None of my experiments used date of rating in any capacity.

Processing the full training set is a computationally intensive operation, so throughout my experiments I worked with various subsets of the full dataset. In this report I refer to the number of users and the number of movies across both datasets as N and M respectively.

Algorithm

1. Read in the training set, and build an "inverse index" is built that maps from each user ID to a list of that user's ratings (much like a postings list'). Normalise each of user u's ratings by subtracting u's mean rating and dividing by the standard deviation of u's ratings. Convert each "postings list" to a sparse vector[4] in $\mathbb{R}^M$ whose i-th entry is u's normalised rating for the i-th movie (if u rated the i-th movie) or 0 otherwise.

2. Initialise the nearest neighbour index by generating kl M-dimensional random hyperplanes. The fastest way of accomplishing this is to generate a $kl \times M$ matrix where each element is randomly chosen from $\{-1, 1\}$. Each column of this matrix corresponds to a hyperplane perpendicular to that column.

3. Index each user's sparse ratings vector into the nearest neighbour structure. Specifically, for each user u with sparse ratings vector $\mathbf{v}_u$:

   (a) Construct $\mathbf{v}_u$'s sketch by taking the matrix product of the $kl \times M$ random matrix with $\mathbf{v}_u$ and compressing the resulting $\mathbb{R}^{kl}$ vector $\mathbf{s}_u$ into its sketch – the binary string of length kl whose j-th entry is 1 iff the j-th entry of $\mathbf{s}_u$ is positive.

   (b) For $i = 0$ to $l - 1$:

      i. Let the string s be the substring of $\mathbf{v}_u$'s sketch from index ki to $k(i+1)-1$ inclusive.

      ii. Add u to the set with key (i, s).[5]

4. For each triple (u, t, r) of user, movie, rating respectively in the test set, attempt predict u's rating r' for t *without* consulting r. Add the squared error $(r - r')^2$ to our set of error values. More specifically, for each (u, t, r) in the test set:

   (a) Retrieve all potential neighbours of u from the nearest neighbour index by the union operation described in section 3. For each potential neighbour u':

      i. Compute the real cosine similarity between u''s vector and u's vector. If the similarity exceeds the predetermined threshold $\cos\theta_2$, add u' to the "near-enough neighbours" set.

   (b) Calculate the weighted average rating r_w for t by considering the rating of each "near-enough neighbour" u' of u that has rated t. Weigh u''s rating by the inverse of the cosine *distance* between u' to u (that is, the weight of u''s rating in the weighted average is the fraction $(1 - \text{cosine-sim}(u', u))^{-1}$). Note that each rating by u' was normalised, so we should not be surprised if t has a negative average rating at this stage (we would expect this to occur if t was generally received poorly).

[4] A sparse vector is a data type representing a mathematical vector. A sparse vector has exactly the same interface as a standard vector, but its operations are optimised for the case where most elements are 0.

[5] The obvious choice for the data structure underlying the nearest neighbours index is a hash table.

(c) Calculate r' by scaling r_w by u's mean rating and standard deviation (i.e. undo the normalisation process). Bound the result within the interval $[1, 5]$ and let this result be r'. Add $(r - r')^2$ to our set of error values.

5. Output the root of the mean of all error values across the test set – that is, the total root-mean-squared error.

Time Complexity

Here I attempt to calculate an expected time complexity for the algorithm.

When we take the weighted average of u's neighbours, we weigh each neighbours' contributions by their respective cosine similarities to u. Our nearest neighbours data structure does not store information about cosine similarities explicitly, so we must compute cosine similarities in the original vector space (the rows of the $N \times M$ utility matrix). Each computation of cosine similarity thus runs in $O(M)$.

We now classify users based on their relationship to some user u. There are users whose cosine similarity to u exceeds $\cos\theta_1$ – we will call these the "true neighbours" of u. There are users whose cosine similarity to u is less than $\cos\theta_2$ – we will say these users are "distant" to u. The remaining users have cosine similarities to u that lie between $\cos\theta_2$ and $\cos\theta_1$ – we will call these users "acquaintances" of u for lack of a better term.

When we query for neighbours of u, our arithmetic in section 3 tells us that each true neighbour has at least probability $1 - \left(1 - p_1^k\right)^l$ of membership in the result set, and similarly that each distant user has at most probability $1 - \left(1 - p_2^k\right)^l$ of membership in the result set. Furthermore, each acquaintance has at most probability $1 - \left(1 - p_1^k\right)^l$ and at least probability $1 - \left(1 - p_2^k\right)^l$ of membership in the result set. If we make the simplifying (but perhaps incorrect) assumption that the angles between u and u's acquaintances are evenly distributed in $[\theta_1, \theta_2]$ then by my calculations the probability that a randomly chosen acquaintance is in the result set is $(1 - \alpha - \beta) N \left[1 - \left(1 - 2^{-k} \left(p_1 + p_2\right)^k\right)^l\right]$. If some fixed fraction α of all users are neighbours, we expect the number of elements in the result set to be at least $\alpha N \left[1 - \left(1 - p_1^k\right)^l\right] + \beta N \left[1 - \left(1 - p_2^k\right)^l\right] + (1 - \alpha - \beta) N \left[1 - \left(1 - 2^{-k} \left(p_1 + p_2\right)^k\right)^l\right]$.

Querying nearest neighbours across n users (e.g. the process of evaluating the algorithm's performance on a test set of n user-movie-rating triples) gives a total cost of $nNM \left(\alpha \left[1 - \left(1 - p_1^k\right)^l\right] + \beta \left[1 - \left(1 - p_2^k\right)^l\right] + (1 - \alpha - \beta) \left[1 - \left(1 - 2^{-k} \left(p_1 + p_2\right)^k\right)^l\right]\right)$. Keep in mind that the only variable terms in this expression are k and l.

Building the index also has a cost: for each user, we take the matrix product of our $kl \times M$ random matrix with each user's $\mathbb{R}^M$ column vector[6] for a total of $O(NMkl)$. The average total cost of indexing and evaluating then is

$$O\left(NMkl + nNM \left(\alpha \left[1 - \left(1 - p_1^k\right)^l\right] + \beta \left[1 - \left(1 - p_2^k\right)^l\right] + (1 - \alpha - \beta) \left[1 - \left(1 - 2^{-k} \left(p_1 + p_2\right)^k\right)^l\right]\right)\right)$$

I question the usefulness[7] of this calculation.

[6] The column vector was originally a row in the utility matrix, but we transpose it to make it suitable for the matrix multiplication.

[7] also the correctness

5 Performance Evaluation

Unfortunately I did not have the computational resources to test the algorithm on the full training set of 100,000,000 ratings. Instead I ran it on a subset of the first 1000 movies, corresponding to 6% of the total dataset. The algorithm was implemented in pure Python and its runtime ranged from 1 hour to 1 day. The algorithm achieved a RMSE of 0.9638. For comparison, predicting the value of 3 for all test cases yields a RMSE of 1.236. A naive algorithm that simply uses the movie's mean rating as its guess scores 1.054. Netflix's own *Cinematch* scores 0.9514 – only 0.012 units lower.

In addition, it is worth noting that

- This approach was 'pure' in the sense that it only applied one class of recommendation technique. All Netflix Prize winners applied ensemble methods and dataset-specific optimisations rather than general purpose algorithms.
- This approach did not attempt to use dates of ratings or the names of movies, two pieces of supplementary information provided in the data.
- This approach is far better suited to the problem of finding many recommendations for one user – in practice, a far more common use case – than the problem of estimating a particular user's rating for a particular movie.

This result demonstrates that locality-sensitive hashing is a viable technique for large-scale collaborative filtering problems.

6 Further Discussion

Optimal *AND/OR* Compositions

Section 3 describes the composition of OR-equality AND-equality. Recall the definition

$$\overline{F}(a) \overset{\vee}{=} \overline{F}(b) \iff \exists i \in \{0, 1, \ldots, l-1\}\, \forall j \in \{0, 1, \ldots, k\}\, f_{i,j}(a) = f_{i,j}(b)$$

Also recall that if $\Pr\left[f(a) = f(b)\right] = p$ then $\Pr\left[\overline{F}(a) \overset{\vee}{=} \overline{F}(b)\right] = 1 - \left(1 - p^k\right)^l$. This construction 'amplifies' high values of p while 'flattening' low values of p (refer to fig. 6.1).

Instead of composing OR-equality with AND-equality, we can also do the reverse. That is,

$$\overline{F}(a) \overset{\wedge}{=} \overline{F}(b) \iff \forall i \in \{0, 1, \ldots, l-1\} : \exists j \in \{0, 1, \ldots, k\}\, f_{i,j}(a) = f_{i,j}(b)$$

The $\forall$ and $\exists$ symbols have been interchanged. Using similar principles as before, we calculate $\Pr\left[\overline{F}(a) \overset{\wedge}{=} \overline{F}(b)\right] = \left(1 - (1-p)^k\right)^l$ – this construction *also* amplifies high p values and flattens low p values. Fig. 6.1 plots both curves. Why do we choose to compose the operators in the former order and not the latter in the algorithm?

For one, it is much harder to develop an efficient nearest neighbour data structure that composes AND over OR. When composing OR over AND as we have done so far, each vector is indexed once under exactly l keys. The query operation is performed by taking the union of the buckets for exactly l keys. However, an equivalent indexing based around AND$\circ$OR is not immediately apparent. A particular k-block of a sketch – e.g. `0100` if $k = 4$ – will compare equal to $2^k - 1 = 15$ size-4 blocks under OR-equality[8]. Should this imply that each vector must be indexed under $(2^k - 1)l$ keys?

[8] All binary strings of length k share except for `1011` share at least one bit with `0100`.

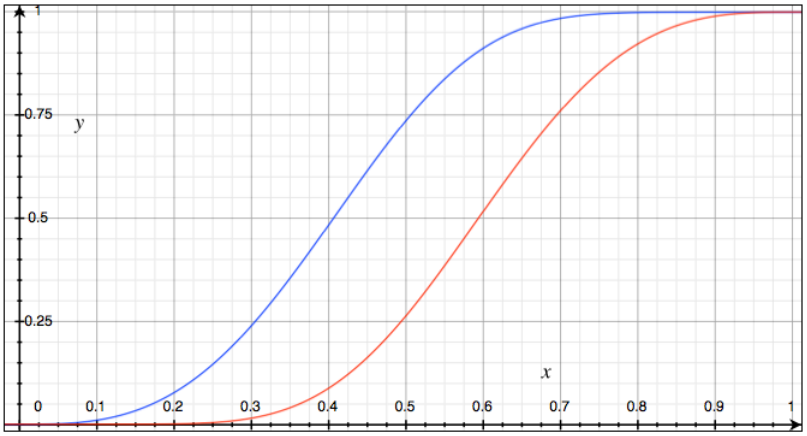

Fig. 6.1: The curves $y = 1 - (1 - x^k)^l$ and $y = (1 - (1 - x)^k)^l$ for $k = 3, l = 10$ in the domain $[0, 1]$. Note the distinctive sigmoid shape that amplifies high values and flattens low values.

(k, l)	p_1	p_2
$(3, 10)$	0.97	0.74
$(6, 30)$	0.94	0.38
$(10, 70)$	0.71	0.07
$(13, 300)$	0.79	0.04

(a) OR-equality composed over AND-equality

(k, l)	p_1	p_2
$(3, 10)$	0.69	0.26
$(6, 30)$	0.96	0.62
$(10, 70)$	1.0	0.93
$(13, 300)$	1.0	0.96

(b) AND-equality composed over OR-equality

Tab. 1: Some values for p_1 and p_2 for fixed $\theta_1 = \pi/3, \theta_2 = \pi/2$ and various k, l.

This is not computationally feasible.

Suppose that we disregard this limitation and instead consider a much simpler scenario. We are given two vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^M$ and their corresponding sketches s_1 and s_2, and we wish to calculate the probability that $\mathbf{v}_1$ and $\mathbf{v}_2$ are neighbours and the probability that they are distant. We will set $\theta_1 = \pi/3$ and $\theta_2 = \pi/2$, so the vectors are neighbours if cosine-sim$(\mathbf{v}_1, \mathbf{v}_2) \geq \cos \frac{\pi}{3}$ and they are distant from each other if cosine-sim$(\mathbf{v}_1, \mathbf{v}_2) \leq \cos \frac{\pi}{2}$.

Using the formulas established in section 3 we list some values for p_1 and p_2 in tables 1.

Our sketch is of size kl. Since the computations of both AND-equality and OR-equality run in time $O(kl)$ it stands to reason that we desire to minimise kl while maintaining reasonable values for p_1 and p_2. Because the OR∘AND and AND∘OR operators mirror each other so closely in design, we might expect them to perform their duties equally well – in other words, we might expect the minimum value of kl required to achieve certain thresholds for p_1 and p_2 across both operators to be similar. However, we find that this is not necessarily true. When $\theta_1 = \pi/3$ and $\theta_2 = \pi/2$ we see empirically that OR∘AND achieves $p_1 = 0.440, p_2 = 0.002$ with $kl = 1000$ $(k = 5, l = 200)$. AND∘OR on the other hand requires $kl = 17432$ to achieve roughly equivalent probability values of $p_1 = 0.339, p_2 = 0.002$ $(k = 19, l = 917)$. What does this mean? OR∘AND can achieve the same value of p_2 and a *better* value of p_1 with *one-seventeenth* of the resources required by AND∘OR.

This would seem to make OR∘AND objectively beter than AND∘OR, at least for these θ_1, θ_2 values.

Until now we have only considered composition constructions of order 2, but in fact we can compose our basic AND/OR operators to arbitrary depths. For instance, we could define the following higher-order equality operator:

$$\overline{F}(a) \stackrel{?}{=} \overline{F}(b) \iff \forall i \in \{0, 1, \ldots, l-1\}\, \exists j \in \{0, 1, \ldots, k-1\}\, \forall h \in \ldots : f_{i,j,h,\ldots}(a) = f_{i,j,h,\ldots}(b)$$

We can narrow the space of possibly distinct forms of $\stackrel{?}{=}$ by observing that the $\exists$ and $\forall$ operators parallel the logical disjunction and logical conjunction operators respectively, and both of these logical connectives are associative. This shows that AND-AND composition can always be rewritten as AND-composition and likewise OR-OR composition can always be rewritten to OR-composition. Here is one such rewriting method:

$$\begin{aligned}(\forall i \in \{0, 1, \ldots, k-1\} : \forall j \in \{0, 1, \ldots, l-1\} : f_{i,j}(a) = f_{i,j}(b)) \iff \\ \big(\forall i \in \{0, 1, \ldots, kl-1\} : f_{\lfloor i/k \rfloor, i \bmod k}(a) = f_{\lfloor i/k \rfloor, i \bmod k}(b)\big)\end{aligned}$$

$$\begin{aligned}(\exists i \in \{0, 1, \ldots, k-1\} : \exists j \in \{0, 1, \ldots, l-1\} : f_{i,j}(a) = f_{i,j}(b)) \iff \\ \big(\exists i \in \{0, 1, \ldots, kl-1\} : f_{\lfloor i/k \rfloor, i \bmod k}(a) = f_{\lfloor i/k \rfloor, i \bmod k}(b)\big)\end{aligned}$$

so all $\stackrel{?}{=}$ operators can be represented by merely alternating consecutive quantifiers between the universal and the existential. This drastically reduces the space of possible $\stackrel{?}{=}$ operators. I believe the number of possible $\stackrel{?}{=}$ operators of depth d is in fact equal to the number of distinct permutations integers $\alpha_1, \alpha_2, \ldots, \alpha_d$ for which $\prod_{i=1}^{d} \alpha_i = kl$.[9] Just as OR-AND composition significantly outperforms AND-OR composition, it is not unreasonable to suspect that some of these unknown higher-order operators could significantly outperform OR-AND composition. Of course, this is purely theoretical speculation – it is unlikely that this higher-order operator would form the basis of an elegant nearest neighbours data structure.

7 Related Work

Andoni and Indyk [1] introduce the nearest neighbours data structure described in section 3. However, their research uses the data structure to query for a *single* nearest neighbour. Consequently they use different formulae to explore the problem.

Terasawa and Tanaka [6] describe "spherical LSH", another application of locality-sensitive hashing to nearest neighbours search using cosine similarity. Rather than explicitly considering cosine similarity, they formulate the problem as a nearest-neighbour search on the unit hypersphere.

In the past, successful recommender systems have used methods from statistical clustering. Ester et al. [2] presents the seminal density-based algorithm DBSCAN, a method of clustering based around nearest neighbour queries. Using the nearest neighbour index described in section 3 we can cluster M movies or N users in $O\left((N+M)kl\right)$. This is much faster than traditional hierarchical or centroid-based clustering methods.

[9] The only variables are the d sets over which we quantify, and the product of the size of these sets must equal kl. Counting the number of sequences $\alpha_1, \ldots, \alpha_d$ is in itself a curious mathematical problem.

8 References

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. IEEE, 2006.

[2] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, volume 96, pages 226–231, 1996.

[3] Joseph Lee Rodgers and W Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[4] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[5] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2012.

[6] Kengo Terasawa and Yuzuru Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *Algorithms and Data Structures*, pages 27–38. Springer, 2007.

[7] Benjamin Van Durme and Ashwin Lall. Online generation of locality sensitive hash signatures. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 231–235. Association for Computational Linguistics, 2010.