A Lattice-based Model for Recommender Systems

Shriram Narayanaswamy and Raj Bhatnagar
Dept. of Computer Science, University of Cincinnati
Cincinnati, OH 45221, USA
narayasm@email.uc.edu, raj.bhatnagar@uc.edu

Abstract

A major challenge in building recommender systems is organizing the recommendation space since, the underlying data organization scheme has a significant impact on the overall performance and capabilities of the recommender system. Lattices offer a natural scheme to encode the recommender space, and their structural properties could be leveraged to efficiently retrieve recommendations. In our lattice-based approach, we present a generic framework and algorithms for building recommender systems. The algorithms convert user ratings into concepts; organize concepts into coherent lattices by imposing constraints based on collaborative filtering, and enable fast querying of lattices for generating recommendations. A lattice-based model also offers interesting insights into the complex higher-level interrelationships between entities in the data. We apply our algorithms on two real-world datasets and demonstrate their capabilities in generating quality recommendations in real-time.

1 Introduction

A *recommender system* is an *information filtering* system that uses people's preferences (collaborative filtering), and/or the content they rate (content-based filtering), to group similar users together. Some popular recommender systems include Pandora [15], MovieLens [13] and Reader's Robot [18]. Recommender systems have been actively explored because of their ability to alleviate the problem of information overload. Collaborative filtering is a popular approach to information filtering and it groups users and/or items based on the reasoning that people with similar preferences like similar items[21]. For instance, successful e-commerce websites such as Amazon.com and Netflix.com recommend additional items to customers using the past history of similar customers. Breese et al. group predictive algorithms for collaborative filtering into two broad categories: memory-based and model-based[6]. In a memory-based approach, recommendations are obtained by aggregating ratings of *similar* users. A number of similarity measures were proposed to group similar users. Popular measures include cosine-based similarity [20], Pearson correlation coefficient [19] and later, extensions such as default voting and case amplification [6]. In model-based approaches, many probabilistic [9][10] and clustering [22] techniques have been employed to represent user preferences. Adomavicius and Tuzhilin present a comprehensive survey of the state of the art in recommender systems [1].

Little attention has been paid to the deep structure inherent in a ratings-database and how its organization may be exploited to enhance the quality of recommendations [12] [2]. Mirza et al. discuss a number of graph-based properties in order to characterize recommender algorithms based on the connectivity they establish. In particular, they discuss hit-buffs which helps describe the connectivity of the graph based on the fact that each buff b_i (e.g. a person) has a set of hits H_i (e.g. a number of movies rated), and each hit_{ij} in H_i may have other buffs b_{ij} (e.g. other people who viewed that movie). This helps connect each buff to another set of buffs. Connectivity can be restricted by the choice of the overlap in hits in order to connect two buffs. In our approach, connectivity is imposed by overlap as well as a proper subset relationship between two nodes.

Adomavicius et al. discuss the use of hierarchies in making recommendations. The motivation for their approach is to use contextual information obtained by aggregating along hierarchies to make accurate recommendations. They assume an underlying multi-dimensional data model similar to OLAP systems and compute aggregations along predefined hierarchies. The power of our approach is in its ability to deal with data that has very little description/metadata. In the context of a movie recommender system, we dont assume the existence of genres. The appearance of genres or other dimensions along pathways in our lattices is a natural consequence of the lattice generation process and indicative of strong correlation in user preferences along those dimensions.

In our work, we have attempted to leverage the hierarchical structure created by general and specialized ratings. A hierarchical structure encoded in a lattice aids in evaluating

candidate recommendations, and choosing one recommendation over another depending on the granularity of a user's query. For example, a user with very few ratings conveys little about his/her preferences, and is made generalized recommendations as opposed to a user with many ratings and thus, a more refined set of preferences.

One of the distinguishing capabilities of this approach is the extraction of latent higher-level knowledge. Exploring pathways in a lattice of movies, for example, could reveal a structure of abstract ontological categories, such as movie genres, and interrelationships among genres. Such higher-level patterns can then be coupled with other dimensions such as age, gender or ethnicity to further discern trends in user preferences. Additionally, a hierarchical structure of concepts enables a quantitative evaluation of the usefulness of an item to a user. Finally, traversing up and down the lattice ensures that we are searching for items based on a common thread/theme, and choosing one item over another does not violate this theme. This is in contrast to cluster-based models wherein sets of items corresponding to varied themes may be aggregated due to partial overlap of items and thus, choosing one set of items over another does not guarantee similar properties. Such complex interrelationships can be easily observed by adopting a lattice-based knowledge representation scheme.

Although concept lattice-based information retrieval (IR) schemes have been proposed earlier [16] [17] [23] [24], they rely on annotations using keywords to index items. We cannot adopt these techniques because collaborative filtering avoids using features/descriptions for each item in a database. Also, the existence of ratings databases in different realms of life creates a need for generic approach that can operate in a domain-independent manner. We strive to address these inadequacies in our two-fold contribution. First, we present a unified approach to concept-based knowledge discovery using a generic and flexible four-component framework to generate recommendations from a ratings database. Second, we present algorithms that can efficiently convert user preferences into concepts, organize them into lattices, and then query the lattices for recommendations. We then present results of applying our algorithms to two different real-world datasets and demonstrate improvement compared to other approaches.

2 Lattice-based approach

Formally, a partially ordered set V is a lattice L=(V,$\leqslant$), when for any two elements x and y in V, the lowest common upper bound x$\bigvee$y and the greatest common lower bound x$\bigwedge$y always exist[8]. Each element in a concept-lattice is a concept. A concept is described by a pair (O,A) where O is a set of objects and A is a st of attributes. Objects are usually some real-world entities and attributes are their charactersistics.

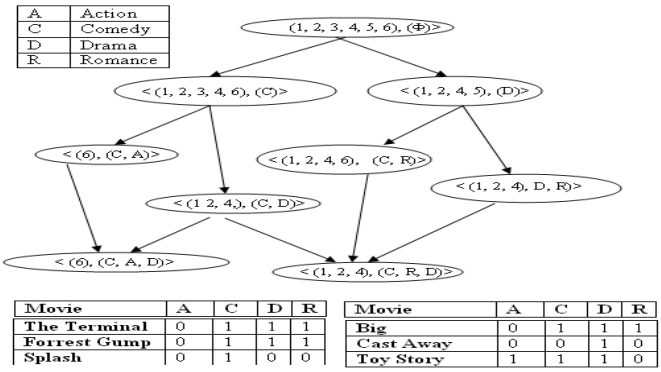

Figure 1: A Sample Concept Lattice

Let us consider a movie recommender system containing movies starring Tom Hanks that implements a concept-lattice. Each concept in the lattice (Fig. 1) is a pair - a set of movie ids (object) and the genre(s) (attribute) associated with the set of movies. {{1,2,3,4,5,6}, ϕ} is the most general concept in the lattice since it contains no attributes (genre, in this case). The concepts {{6}, $\{C, A, D\}$} and {{1,2,4}, $\{C, R, D\}$} are the most specialized since they belong to 3 genres.{{1,2,3,4,6}, {C}} is a parent of {{1,2,4}, {C,D}} since {1,2,3,4,6} $\supsetneqq$ {1,2,4} and {C} $\subsetneqq$ {C,D}. Similarly, {{1,2,4},{C,R,D}} is a child of {{1,2,4,6}, {C,R}} since {1,2,4} $\subsetneqq$ {1,2,4,6} and {C,R,D} $\supsetneqq$ {C,R}. Recommendations are made by zeroing in one regions of interest or subspaces within a matrix of user preferences and movies. Haiyun et al. propose ideas that find such interesting subspaces [23] [24].

3 Framework

The framework in Figure 2 is our four-step approach to building and querying a lattice-based model. Each step is discussed below:

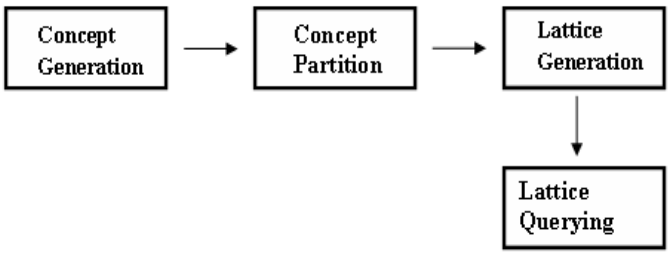

Figure 2: Our framework for recommender systems

3.1 Concept Generation

This component achieves the first step of the process of discovering knowledge from ratings databases, namely, converting the user ratings into concepts. This algorithm

improves upon the basic subspace clustering algorithm presented in SCuBA by Agarwal et al., [3]. The basic algorithm of SCuBA accumulated a large number of patterns rated by an insignificant fraction of users. The pruning step was applied after all comparisons were made, and hence, all the generated concepts had to be held in memory until the final step. We have overcome this space inefficiency by optimizing the comparison process. We prune patterns with certain number of ratings at the end of each iteration of the pair-wise comparison process. We have achieved 60-70% reduction in the space required to store the generated concepts (See Results for experimental results). Although itemset mining algorithms such as APriori(frequent itemset)[4] and MAFIA(maximal itemset)[7] could be applied, these algorithms are designed to be complete (discover all possible itemsets). Since we do not require a complete algorithm, we chose to implement a pair-wise comparison based approach to discover repeated patterns of items. Given a ratings database, the following algorithm converts user ratings into concepts. Each concept is a set of items that has been rated by some minimum fraction of users, called the support of the concept. A lower bound δ on this support specifies the minimum number of occurrences of a pattern in the database to make it frequent, and hence a candidate concept. Each concepts is, thus, a set of rated items. For example, A{1, 2, 3} is a concept named 'A' that contains the items 1, 2 and 3 as its constituents. By virtue of being generated by the following algorithm, the set {1,2,3} has met the minimum support threshold percentage δ.

Algorithm for Concept Generation

Algorithm *conceptGenerate*
Input: A database of user ratings
Output: A set of patterns that occur frequently

```
1.  frequentPatterns ←[ ]
2.  Sort database in decreasing order of number of ratings
3.  for i ← 1 to sizeOfDatabase − 1
4.      for j ←i+1 to sizeOfDatabase − 1
5.          Compute pair-wise intersection X_ij of
            row(i) and row(j)
6.          if local hashtable ∈ X_ij, increment
            value by 1.
7.          else insert key Xij, value ←1.
8.          endif
9.      endfor
10.     lengthI ←number of ratings in row(i)
11.     lengthI1 ←number of ratings in row(i-1)
12.     if lengthI ≠ lengthI1
13.         for ∀ (k,v) entry-pair in local hashtable,
            remove (k,v) if
14.             key k has number of ratings ⩾
                lengthI and value v < δ% of
                sizeOfDatabase
15.         endif
16.     endif
17.     Update globalHashTable ←localHashTable
18.     Clear localHashTable
19. endfor
```

The pruning is done based on the observation that the intersection of two sets P and Q produces a resultant set S of cardinality no more than the cardinality of P or Q. In fact, the cardinality of S is no more than the minimum of P and Q's cardinality. Thus, when ratings are sorted in the decreasing order of the number of ratings per user, computing pair-wise intersections produces intersections of reducing length. For example, a set of 5 rated items can produce at the most 4 common items when intersected with sets of cardinality 5 or lower (assuming no repeating sets). Hence, new intersections of size 5 or more are not possible in future intersections of sets of length 5 or lower. This means we can prune away those intersections of length 5 or more if they do not meet the minimum support criterion.

3.2 Concept Partition

In sparse databases, generated concepts may have few items in common and so, placing them in the same lattice may not be possible if parent concepts are to be proper subsets of children concepts. The optimized lattice search algorithm presented in section 3.4 requires that this parent-child relationship be preserved and forcing dissimilar concepts into the same lattice will violate this relationship and preclude the optimization achieved by our algorithm. The presence of disconnected concepts in a lattice will have undesirable consequences such as slower lattice generation, querying and inaccurate recommendations. This component helps partition concepts into multiple subsets of concepts, each belonging to a different lattice.

3.2.1 The case for multiple lattices

To illustrate the need for concept partition, let us consider the ratings database shown in Table 3. By applying the algorithm described in the previous section, we can generate the following concepts: (E), (A,B), (D,E), (C,D,E), (A,B,C). If we apply the requirement that all children concepts should be proper supersets of their parents (w.r.t rated

	A	B	C	D	E
1	1	1	1	0	0
2	1	1	0	0	0
3	0	0	0	1	1
4	0	0	1	1	1
5	0	0	0	0	1

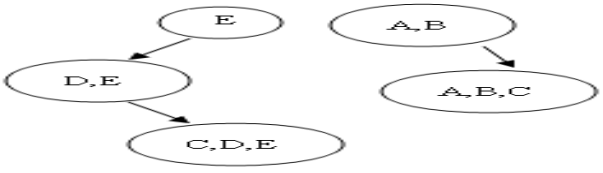

Figure 3: A sparse dataset and its associated lattice

items), there are two disconnected segments in this lattice - {(A,B,C), (A,B)}, and {(E) (D,E) (C,D,E)}. Clearly, these two subsets of concepts belong to different lattices and this algorithm partitions them.

3.2.2 Algorithm for Concept Partition

Algorithm *conceptPartition*
Input: A list of unique patterns
Output: Groups of concepts that belong to different lattices

```
1.  SizeOfConcepts ←number of unique patterns in frequentPatterns. conceptSet[ ][ ]←0.
2.  visited[SizeOfConcepts] ←0
3.  Sort frequentPatterns in the decreasing order of number of ratings in each concept
4.  for i ← 1 to SizeOfConcepts
5.          currentConcept ←frequentPatterns[i]
6.          if visited[i] EQUALS 1 continue; endif
7.          count ←0
8.          for j ←i+1 to SizeOfConcepts
9.                  compareConcept ←frequentPatterns [j]
10.                 if compareConcept ⊊ currentConcept
11.                         visited[j] ←1.
12.                         conceptSet[i][count]
                            ←compareConcept[j].
13.                         Increment count. endif
14.                 endfor
15.         endfor
```

The basic idea of the algorithm is the following. Since we want to find subsets of a given concept and group them into one lattice, we start with concepts having the most ratings. To achieve this, we sort the set of concepts in the descending order of the number of ratings. Next, each concept E is considered for creating a new partition. We then go through the set of concepts and identify all subsets, M, of E. The concept in E is then grouped together with the concepts in M as one partition. We then mark all concepts in M as visited implying that they will not be considered for creating a new partition. This ensures that a partition is not contained in any other partition. We then proceed with the next available concept that is not visited and continue this process until all concepts are visited.

3.3 Lattice Generation

Once concepts are identified, we need an efficient means to organize them into lattices. In this section, we present an optimized algorithm for generating a lattice from a set of concepts. The basic lattice generation algorithm builds a lattice incrementally, searching exhaustively to find the right position to insert a new concept. Although this approach is guaranteed to generate the right lattice, it suffers from the problem of searching the lattice explicitly each time a new concept is added. In this algorithm, we employ a level-wise approach; when a new concept is to be added, the level-wise algorithm reduces the likelihood of searching the entire lattice.

Algorithm for Lattice Generation

Algorithm *latticeGenerate*
Input: A list of unique concepts conceptList
Output: A lattice of concepts

```
1.  Sort concepts in increasing order of concept length
2.  Split concepts into groups such that all concepts in G_i have i rated items
3.  currentLattice←{}
4.  for each group G_i in G
5.          Combine concepts in G_i with currentLattice by calling combineLevels(currentLattice,G(i)).
6.  return currentLattice
7.
8.  combineLevels(...)
Input: A lattice,the set C of concepts to be added
Output: A lattice with concepts in C added
9.  for each concept X in C
10.         L ←bottom-most level of lattice
11.         while true
12.            do
13.                elusiveItem ←X
14.                for each concept Y in L
15.                        if (elusiveItem ⋂ Y = φ)
16.                                continue;
17.                        X = X - Y;
18.                endfor
19.                if X = {φ}
20.                         break;
21.                else
22.                L ←next higher-level of L;
23.         repeat
24.      endfor
```

The primary requirement of a lattice building algorithm is to avoid cyclic redundancy. Cyclic redundancy occurs when ancestors of a node are also added as parents. For instance, consider three concepts A, B and C. Suppose A is a child of B and B, a child of C. We need to ensure that B alone is identified as A's parent, and not C, although it is a valid ancestor of A. This is done in order to ensure that the lineage of a node is traced along a unique path. In order to ensure that there is no cyclic redundancy, basic lattice generation algorithms search for ancestors explicitly and avoid them. This often requires traversing an entire lattice at least once [5]. The motivation behind this approach is to avoid cyclic redundancy implicitly by merging levels incrementally. In order to achieve incremental building, concepts are first sorted in the increasing/decreasing order of number of ratings and then split into groups having equal number of ratings.

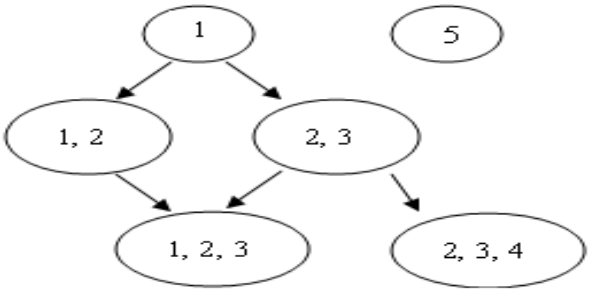

Figure 4: A partially built lattice

The algorithm merges a new concept X with the bottom of the lattice by computing the set difference between X and each concept Y in the bottom level and assigning the difference to X. X, in effect, keeping track of ratings that have not appeared in one or more parents (elusive items). After comparing with all the concepts at the bottommost level, if X is not empty, then one or more items in X have not appeared in any of the concepts Y. We, then, try comparing X with one level higher and so on until the top of the lattice is reached or until X becomes empty. The essence of this algorithm is to keep track of the elusive items and search the lattice until no elusive items exist or all concepts in the lattice have been explored. Consider the following example to understand how cyclic redundancy is avoided: Given A = {1,2,3,4,5}, B = {1,2,3}, C={2,3,4}, D={1,2}, E={1,3}, F={5}. Now, suppose A is to be added to a lattice shown in Figure 4. The bottom-most level has B and C, and they clearly are subsets of A, and hence are added as parents. At this stage, elusiveItem = A = {5}. We now explore, the next higher-level (with ratings of length 2). Here, although, D and E are candidates, they are not added since they are disjoint with elusiveItem. At the next level, F is added as a parent of A and the algorithm terminates. Clearly, keeping track of elusive items avoids cyclic redundancy. Now, if A = {1,2,3,4} instead, then the algorithm would have terminated after the bottom-most level itself and we would not have explored the entire lattice saving considerable time.

3.4 Lattice Querying

Once the lattice is built, we can query the lattice for information. In this case, we can query the lattice for a recommendation based on the past history (previous ratings) of a user. This component uses algorithms that can discover recommendations efficiently from large lattices. One of the key contributions of this paper is the kind of knowledge that can be extracted using our approach. In addition to the basic recommendations, lattices can reveal latent trends along pathways in the lattice. As mentioned before, this can be exploited to infer higher-level domain-specific knowledge such as the association between movie genres and age of users. Also, since all sets of items in a lattice exhibit some common underlying theme, searching a lattice offers us the liberty to choose one item over another within the same lattice without the fear of compromising the kind of profile searched for. This may not be possible in other approaches, especially in clustering schemes where sets of items may be grouped on partial matches in the rated items, because the remaining items in each set may have no real connection to that cluster. This component uses an algorithm that can discover recommendable items efficiently from large lattices using the following property:

Upward Closure: In a lattice, if an item e is not present in a concept C, then it will not be present in any parent of C. Thus, when searching for a minimum of η matching items, if a concept does not have η matches, none of its parents will have η matches and hence need not be considered as candidates for recommendation.

Algorithm for Lattice Querying

Algorithm *latticeQuerying*
Input: A user query Q of items of the form (UserId, RatingId1, RatingId2,...), a minimum match threshold η and a minimum to-recommend threshold δ
Output: A list of items that the user is likely to rate
1. Candidate Nodes ←Nodes in the bottommost level of the lattice
2. Good Nodes ←[ ]
3. while Candidate Nodes is not empty
4. node ←Get next node in Candidate Nodes
5. parents ←Get all parents of node
6. **if** $|\text{node} \bigcap Q| \geq \eta$
7. **then**
8. Append parents to Candidate Nodes
9. Add (node, $|\text{node} \bigcap Q|$) to Good Nodes
10. endif
11. endwhile
12. Sort Good Nodes in descending order of $|\text{node} \bigcap Q|$.
13. **if** Good Nodes is EMPTY return NULL
14. **else**
15. Return the nodes in Good Nodes which has atleast $|\text{node} \setminus Q| \geq \delta$ items to recommend
16. endif

The search for a recommendation starts from the bottom of the lattice and proceeds in the direction of increasing generality until the top of the lattice is reached. Each candidate concept should have at least η items in common with the query Q in order to be a candidate for recommendation. If a given concept does not have η common items, then it can safely be ignored. Also, due to the upward closure property, all parents of this concept can be ignored. This saves a lot of time as compared to ignoring just the candidate concept. Even in a moderately connected lattice, our algorithm can achieve a lot of pruning. Once the lattice is traversed

completely, a number of candidate concepts are discovered and recommendations are made based on factors like number of recommendations desired and the usefulness of the recommendations.

4 Results

We investigate the performance of the presented framework and the algorithms on two real world dataset, namely the Jester dataset and the MovieLens dataset. We apply our algorithms on the two typical yet contrasting datasets, Jester and MovieLens, to analyze their performance. All the experiments have been performed on a 2GB RAM Dell Desktop running P4 3.2 SUSE Linux operating system. The graph in Figure 5 shows the actual number of patterns generated by the basic and optimized concept generation algorithms together for Jester. From the above figures it is evident that the number of concepts generated by the optimized algorithm is significantly lower than by the method employed by SCUBA. Also, average pruning percentages (Figures 6 and 7) are in the range of 60-70% of the generated concepts. The MovieLens dataset model building time is shown in Figure 8. This is comparable with the model-building time observed by Agarwal et al., [3]. We observe that the total model building time is largely linear and has potential for high scalability. The space requirement for the MovieLens datasets have been shown in Figure 9 respectively. In order to study the real time performance of the algorithm, we built a lattice consisting of nearly 10500 concepts (after pruning) using 10000 user ratings from the Jester data set. After the model was built, we randomly chose 1000 user ratings from the remaining 13500 users and created 4 data sets each containing 250 users. These are labeled as Jester Set1 through Jester Set4. In order to measure query performance in terms of speed of processing and accuracy of results, we calculated the following - query processing time, precision and recall.

From each row of the user ratings, we extract a portion of the ratings and label them as query terms. These could be the initial ratings made by a new user of an online rating system. The goal of the recommender system is to predict possible items of interest to the new user. These are referred to as the target terms. Recommendations made by the system are referred to as recommended terms. The following definitions have been applied to calculate the various metrics. For instance,given a list of ratings (Joke1, Joke2, Joke3,...,Joke50), we may use the first 15 (Joke1,...,Joke15) as query terms and (Joke16,...,Joke50) as target terms. Suppose the recommender offers recommender terms = (Joke19 , Joke31, Joke35, Joke56) as recommendations, then precision = $\frac{3}{4} * 100$= 75% and recall=$\frac{3}{35} * 100 \sim 10\%$.

Time: Amount of time in milliseconds required to retrieve a recommendation for a given set of query terms.

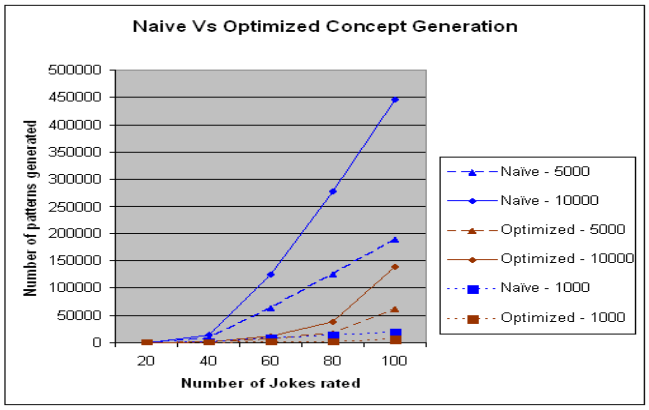

Figure 5: Naive Vs Opt. Concept Generation - Jester

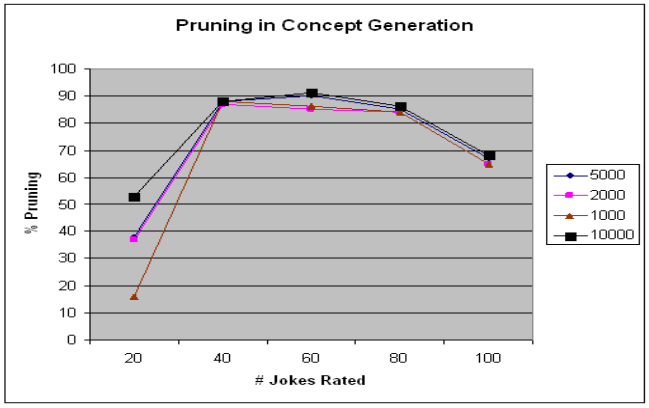

Figure 6: Jester-Pruning to generate concepts

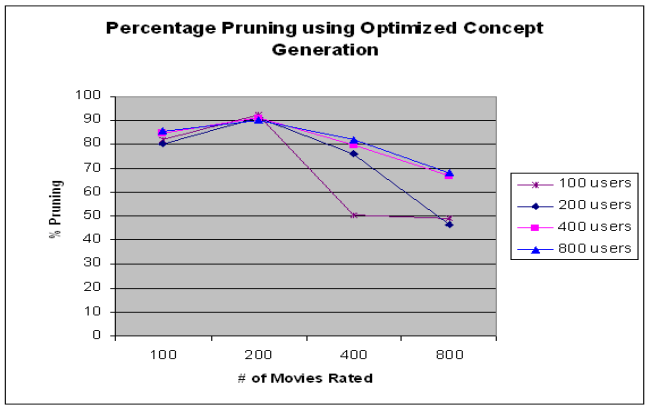

Figure 7: MovieLens-Pruning to generate concepts

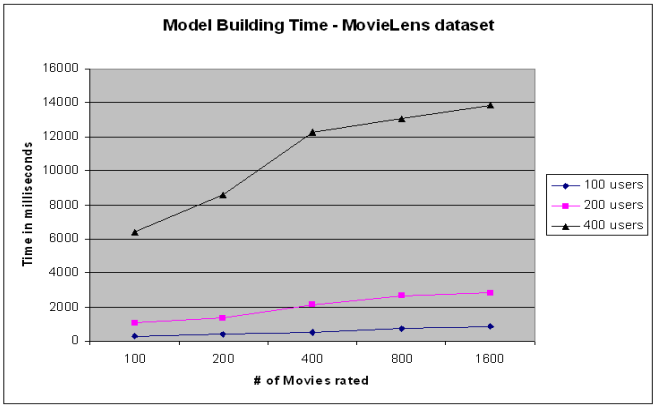

Figure 8: MovieLens: Total Time

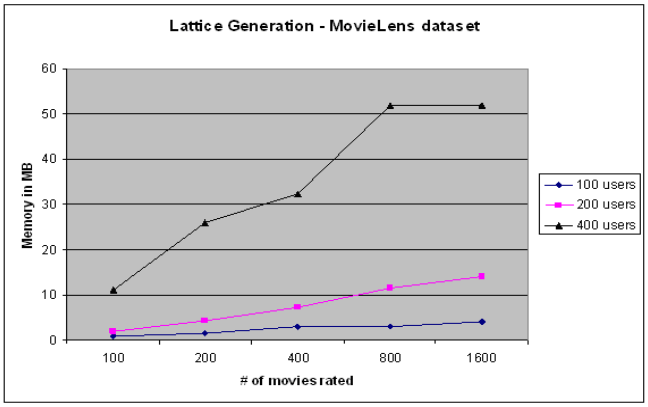

Figure 9: Movielens:Space for concepts

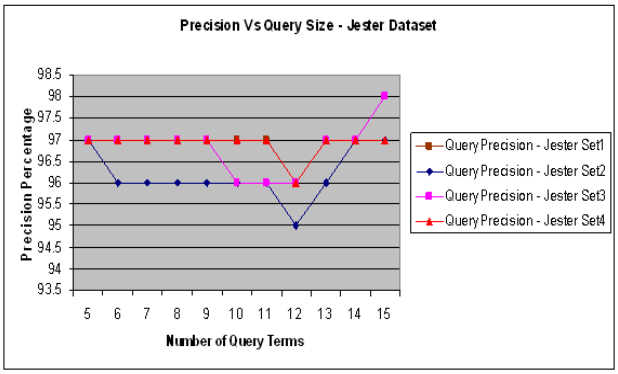

Figure 10: Jester:Precision

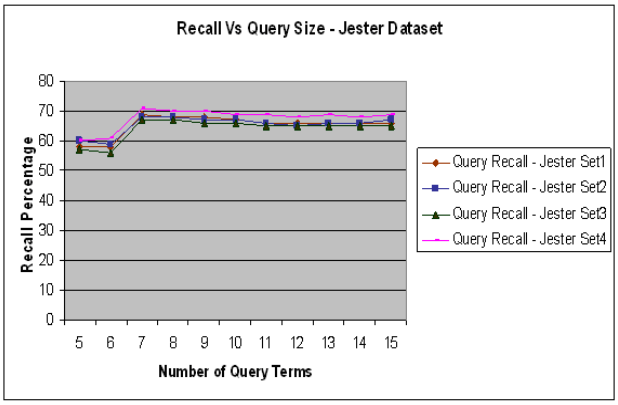

Figure 11: Jester:Recall

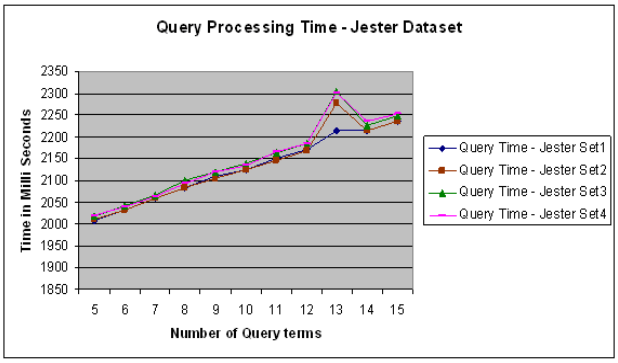

Figure 12: Jester: Query processing

Precision: $\frac{RecommenderTerms \bigcap TargetTerms}{RecommenderTerms} * 100$

Recall: $\frac{RecommenderTerms \bigcap TargetTerms}{TargetTerms} * 100$

The size of the query terms ranges between 5 and 15% of the maximum number of items in the data set. This range was chosen because users are likely to rate very few items before requesting recommendations. As can be noted in Figure 10, the average precision for the Jester data set is about 97% while the average recall in Figure 11 is about 65%. As observed by Agarwal and others [3], recall can be increased by offering more recommendations at the cost of precision. It is essential for users to get the right recommendations rather than get many recommendations. Hence, we decided to tradeoff precision over recall. Finally, in Figure 12 we can observe that the average query processing time is approximately 2 seconds which is acceptable for real-time systems.

4.1 Performance Analysis

The concept generation algorithm uses pair-wise comparisons and has an asymptotic worst-case complexity of $O(n^2)$ where n is the number of users in a ratings database.The concept partition algorithm also has $O(n^2)$ worst-case complexity where n is the number of concepts after pruning. However, even if a dataset is moderately linked, after every iteration many concepts would be marked as visited and hence would not be considered for finding subsets. This reduces the number of concepts to be explored dramatically. Agarwal et. al.[3], use the MovieLens dataset to compute precision of their approach. The SCuBA algorithm[3] shows constantly degrading precision as the percentage of terms increase from 5% to 50%. On the other hand, our approach shows a steady performance independent of the number of items considered 13. Although the performance is sightly poorer at lower percentages, our precision values remain consistent and this is very essential for a good overall system. When the number of query terms increases, the users expect the system to have learnt their preferences well and would not tolerate degrading performance.

5 Conclusion and Future Direction

We believe that the concept based approach is generic and can be adopted by data of any nature especially to data with little or no metadata. The lattice structures discern natural ordering of the concepts which can then be validated by domain experts. Another strength of this approach is that the model-building time and real-time performance are acceptable for any modern application. Our basic model can be easily augmented by adding user statistics to each node in the lattice to guide us in the search for recommendations. If there is a natural partitioning in the items of the ratings database, then they would translate to multiple concept lists after the concept partitioning process and these can be used

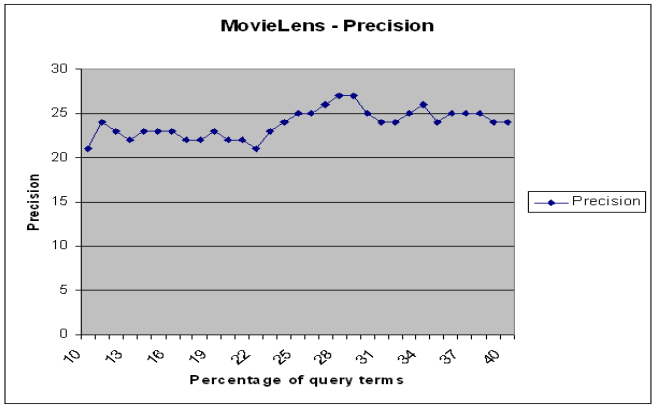

Figure 13: MovieLens: Precision

to produce smaller lattices that can be searched faster and may be in parallel. Also, parallel computation of the best nodes in lattices can be explored.

References

[1] G. Adomavicius and A. Tuzhilin, Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions, IEEE Transactions on Knowledge and Data Engineering, 2005, pp. 734–749.

[2] G. Adomavicius, R. Sankaranarayanan, S. Sen and A. Tuzhilin, Incorporating Contextual Information in Recommender Systems Using a Multidimensional Approach, ACM Transactions on Information Systems, Vol. 23, Issue 1, 2005, pp. 103–145.

[3] N. Agarwal, E. U. Haque, H. Liu, and L. Parsons, GroupLens: A Subspace Clustering Framework for Research Group Collaboration, International Journal on Information Technology and Web Engineering, Vol. 1, 2006, pp. 35-58.

[4] R. Agrawal and R. Srikant, Fast Algorithms for Mining Association Rules, Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, 1994, pp. 487–499.

[5] A. A. Yardi, Concept Based Information Organization and Retrieval, Masters Thesis, University of Cincinnati, 2006.

[6] J. Breese, D. Heckerman and C. Kadie, Empirical Analysis of Predictive Algorithms for Collaborative Filtering, Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI-98), 1998, pp. 43-52.

[7] D. Burdick, M. Calimlim and J. Gehrke, MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases, Proceedings of the 17th International Conference on Data Engineering, 2001, pp. 443-452.

[8] B. Ganter and R. Wille. Translator - C. Franzke, Formal Concept Analysis: Mathematical Foundations, Springer-Verlag New York Inc., 1997.

[9] L. Getoor and M. Sahami, Using Probabilistic Relational Models for Collaborative Filtering, Working Notes of the KDD Workshop on Web Usage Analysis and User Profiling, 1999.

[10] K. Goldberg, T. Roeder, D. Gupta and C.Perkins, Eigentaste: A Constant Time Collaborative Filtering Algorithm, Information Retrieval,Vol. 4, No. 2, 2001, pp. 133-151.

[11] Jester Online Joke Recommender Dataset Webpage, http://ieor.berkeley.edu/ goldberg/jester-data/

[12] B. Mirza, B. Keller and N. Ramakrishnan, Studying Recommendation Algorithms by Graph Analysis, Journal of Intelligent Information Systems, Vol. 20, Issue 2, 2003, pp. 131 - 160.

[13] MovieLens Movie Recommender Webpage, http://movielens.org/

[14] MovieLens dataset from GroupLens Research Group at the University of Minnesota, http://www.grouplens.org/node/12#attachments

[15] Pandora Music Recommender System created by Music Genome Project, http://www.pandora.com/

[16] G. S. Pedersen, A browser for bibliographic information retrieval based on an application of lattice theory, SIGIR '93: Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval, 1993, pp. 270–279.

[17] U. E. Priss, Lattice-based Information Retrieval, Knowledge Organization, Vol. 27, No. 3, 2000, pp. 132-142.

[18] Reader's Robot - Book Recommender System, http://movielens.umn.edu/

[19] P. Resnick, N. Iacovou, M. Suchak, P. Bergstorm and J. Riedl, GroupLens: An Open Architecture for Collaborative Filtering of Netnews, Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work, 1994, pp. 175-186.

[20] B. M. Sarwar, G. Karypis, Joseph A. Konstan and J. Reidl, Item-based collaborative filtering recommendation algorithms, Proceedings of the Tenth International World Wide Web Conference, 2001, pp. 285-295.

[21] U. Shardanand and P. Maes, Social Information Filtering: Algorithms for Automating "Word of Mouth", Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems, Vol. 1, 1995, pp. 210–217.

[22] L. Ungar and D. Foster, Clustering Methods For Collaborative Filtering, Proceedings of the Workshop on Recommendation Systems at the Fifteenth National Conference on Artificial Intelligence, 1998, pp. 112–125.

[23] Haiyun Bian and Raj Bhatnagar. A Levelwise Algorithm for Interesting Subspace Clusters. Proceedings of the 2005 IEEE International Conference on Data Mining, held in November 2005.

[24] Haiyun Bian and Raj Bhatnagar. An Algorithm for Mining Weighted Dense maximal 1-Complete Regions. In the book "Data Mining: Foundations and Practice," Studies in Computational Intelligence Series, Vol. 118/2008, published by Springer, 2008, pp. 31-48.