Summer 2019

Effect of Neighborhood Approximation on Downstream Analytics

Saranya Soundar Rajan
San Jose State University

EFFECT OF NEIGHBORHOOD APPROXIMATION ON DOWNSTREAM ANALYTICS

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Saranya Soundar Rajan

August 2019

The Designated Thesis Committee Approves the Thesis Titled

EFFECT OF NEIGHBORHOOD APPROXIMATION ON DOWNSTREAM ANALYTICS

by

Saranya Soundar Rajan

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

August 2019

David C. Anastasiu, Ph.D.	Department of Computer Engineering
Mahima Agumbe Suresh, Ph.D	Department of Computer Engineering
Gheorghi Guzun, Ph.D.	Department of Computer Engineering

ABSTRACT

EFFECT OF NEIGHBORHOOD APPROXIMATION ON DOWNSTREAM ANALYTICS

by Saranya Soundar Rajan

Nearest neighbor search algorithms have been successful in finding practically useful solutions to computationally difficult problems. In the nearest neighbor search problem, the brute force approach is often more efficient than other algorithms for high-dimensional spaces. A special case exists for objects represented as sparse vectors, where algorithms take advantage of the fact that an object has a zero value for most features. In general, since exact nearest neighbor search methods suffer from the "curse of dimensionality," many practitioners use approximate nearest neighbor search algorithms when faced with high dimensionality or large datasets. To a reasonable degree, it is known that relying on approximate nearest neighbors leads to some error in the solutions to the underlying data mining problems the neighbors are used to solve. However, no one has attempted to quantify this error or provide practitioners with guidance in choosing appropriate search methods for their task. In this thesis, we conduct several experiments on recommender systems with a goal to find the degree to which approximate nearest neighbor algorithms are subject to these types of error propagation problems. Additionally, we provide persuasive evidence on the trade-off between search performance and analytics effectiveness. Our experimental evaluation demonstrates that a state-of-the-art approximate nearest neighbor search method (L2KNNGApprox) is not an effective solution in most cases. When tuned to achieve high search recall (80% or higher), it provides a fairly competitive recommendation performance compared to an efficient exact search method but offers no advantage in terms of efficiency (0.1x—1.5x speedup). Low search recall (<60%) leads to poor recommendation performance. Finally, medium recall values (60%—80%) lead to reasonable recommendation performance but are hard to achieve and offer only a modest gain in efficiency (1.5x—2.3x).

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, professor David C. Anastasiu, of the Computer Engineering Department at San José State University for accepting me as his thesis student, and for being the greatest advisor in the world. Without his supervision, help and valuable suggestions, it would have been impossible for me to complete my thesis.

I would also like to thank my committee members, professor Gheorghi Guzun and professor Mahima Agumbe Suresh, for participating in my thesis defense committee and providing valuable comments on my thesis.

Last, but not least, I would like to thank my family and my husband for their patience, moral and, financial support, and encouragement.

Above all, I thank Almighty God for His grace and countless blessings.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

With increasing business competition, companies such as Amazon and Google strive to provide value to customers by trying to understand individual user behavior in order to recommend useful products to customers. The most important challenge that online recommendation systems face is to generate a list of $top-N$ recommendations for an individual user by predicting what the user will like based on his or her similarity to the other users in the system. The more similar the considered neighbors are to the user, the better the recommendation tends to be. Usually, recommendation systems are applied on sparse high-dimensional data, where some items have few ratings, or some users have rated few items.

Nearest neighbor search is a well-known method that is used to identify points in a given set that are closest (or most similar) to a given point. It is commonly believed that the exact nearest neighbor search is very expensive in high-dimensional data. Although finding exact neighbors is efficient for low-dimensional data, the increase of dimensionality results in the increased time taken to calculate similarities, and poor overall performance. This led to the development of approximate nearest neighbor (ANN) algorithms that aim to speed up the process by returning some, but not necessarily all, of the true nearest neighbors. Some methods (e.g., FLANN [1], Annoy [2]) were able to show that finding approximate neighbors saves a notable amount of computation time compared to the exact nearest neighbor search approaches. However, less computation time does not really guarantee the quality of the overall analytics results.

In this thesis, a comprehensive set of experiments is conducted to investigate how the approximation error of the search method affects results in the recommendation domain. We consider several state-of-the-art recommender system algorithms, and omit others that have been dominated by them. Our research aims to provide persuasive evidence on the trade-off between search performance and analytics effectiveness for the recommendation problem.

The research contributions of this work are:

- We conduct a comprehensive experimental evaluation of nearest neighbor methods to understand the trade-off between exact approximate-based methods in an analytics pipeline.
- Our experimental results show the correlation between the approximation error and search effectiveness and efficiency, as well as the quality of the pipeline end-point analytics.
- Based on the analysis, we provide needed practitioner guidance in choosing parameters for appropriate search methods as part of these pipelines.

This thesis does not attempt to improve any algorithm. Rather, the research question of this thesis is:

In using approximate nearest neighbor methods as part of a recommendation pipeline, what is the trade-off between the search performance and the effectiveness of the recommendation engine?

2 LITERATURE REVIEW

The popularity of e-commerce has been quickly growing in recent years. With the continuous development of e-commerce platforms, information retrieval has become necessary for e-commerce sites. Companies make suggestions on their websites to help customers find new and relevant products. One of the major problems companies face is improving the techniques used to find appropriate and relevant items for the customers. Several methods have been developed to find good recommendations. Some of the algorithms use nearest neighbor search methods to calculate similarities between users or items, which are later used to predict recommendations for a particular user.

2.1 Nearest Neighbor Search

Finding nearest neighbors for a set of objects is a common task in many fields, such as online advertising, recommender systems, image recognition, computer vision, etc. The problem of finding nearest neighbors is called nearest neighbor search (NNS). It is defined as follows: given a collection of *n* objects and an arbitrary query object, build a data structure which reports the data set object that is most similar to the query. The problem can be formulated as

$$X_* = \underset{X \in D}{argmin}\, \rho(X, Q), \tag{1}$$

where $D = \{X_1, ..., X_n\} \subset \mathbb{R}^d$ is a data set, Q is a query, and ρ is a distance measure. Nearest neighbor search methods can be divided into exact and approximate algorithms.

2.1.1 Exact Neighbor Search

Exact nearest neighbor search algorithms return the true nearest neighbors of any given query point. One of the earliest data structures that supports finding exact nearest neighbors is the KD-Tree (*k*-dimensional tree) [3]. It is a binary tree in which every leaf node is a *k*-dimensional point. KD-Tree works well for finding nearest neighbors in low-dimensional spaces. When the number of dimensions increases, its performance degrades as query time increases exponentially. Considerable progress has been made on scaling exact nearest neighbor search to high dimensions. For example, the cover tree data structure [4] can remove the exponential dependence completely. Cover trees were designed to work with high-dimensional data that have low intrinsic dimensionality. Despite much research on exact neighbor search, for higher values of dimensions almost all of the known techniques are affected with the curse of dimensionality. For objects represented in sparse vectors, there have been some methods developed to address the sparsity issue, e.g., Greedy Filtering [5] and L2KNNG [6].

Many methods have been proposed to reduce computation time by finding only some of the nearest neighbors. The complexity of the exact nearest neighbor search led researchers to construct data structures for the approximate nearest neighbor search problem as a way to avoid the curse of dimensionality [7], [8].

In an analytics pipeline, it is important to choose the algorithm that works best for each type of problem. When choosing an NNS algorithm, we must consider various factors such as search time complexity, search quality, data set dimensionality, number of samples, parameter settings, and the effort required for tuning method parameters. Unfortunately, none of the prior studies evaluate these measures completely and thoroughly. Gionis et al. [9] assume that in many cases it is not necessary to insist on exact neighbors. Rather, approximate nearest neighbors would suffice. They vaguely address the details of why ANN results are good enough. In general, there are discrepancies in experimental results reported on approximate nearest neighbors as each library takes a different approach and has different weak points and different results [10].

2.1.2 Approximate Nearest Neighbor Search

In c-approximate nearest neighbor search, instead of returning all of the closest points from the query point, the algorithm returns any point p within a radius c from the query q

with some probability. Unlike exact nearest neighbor methods, ANN do not return all of the correct neighbors, potentially missing some of the closest neighbors. The main interest of the approximate nearest neighbor search is the trade-off between query time and accuracy. It can be measured either in terms of the probability of finding the true nearest neighbors (recall) or distance ratios.

There are mainly four types of ANN methods:

1) Tree-based (hierarchical structure) methods, such as those using the randomized KD-tree or the K-means tree.
2) Hashing-based methods, such as locality sensitive hashing (LSH).
3) Graph-based methods.
4) Permutation-based methods.

Tree-based methods perform very well when the dimensionality of the data is relatively low. They often employ space-partitioning techniques to group samples that are in close proximity to each other. However, their storage utilization increases and search performance decreases as the number of dimensions grows, making these methods ill-suited for searching high-dimensional data sets.

LSH was introduced by Indyk and Motwani [8]. As the name suggests, LSH uses hash functions to produce similar hashes for items that are close to each other. Before the search, as a pre-processing step, for each data point the method uses several hash functions to form a new identifier, called a signature. These hash functions reduce the dimensionality of the data based on random projections. The data are then divided into uniform bins. If there is a large number of bins, a second hashing step may be performed to obtain a smaller signature. At a similarity search time, the algorithm returns candidates by first hashing the query point using hash functions and then finding the close data points in the same bin. Through linear search, the final nearest neighbors are selected from the candidate data points, which is a much smaller subset of objects than the full data set. Given a proper choice of hash functions, LSH is not susceptible to the curse of dimensionality and works well with larger data sets [11].

Graph-based methods have drawn considerable attention in recent years. Some of the popular graph-based methods are NN-Descent [12], K-Graph [13], and GNNS [14]. Their main idea is that a neighbor of a neighbor may also be a neighbor. These methods find candidate neighbors of a query point using random selection, and then iteratively check the neighbors of these candidate neighbors for closer points. While they offer high search efficiency, graph-based methods tend to have high computation times for building the *k*-nearest neighbor graph data structure when the data set is large.

Permutation-based methods were first introduced by Chavez et al. [15] and Amato et al. [16]. These methods use a dimensionality-reduction technique, in which each data point is represented by a ranked list of vectors, called permutations, which are then sorted based on the increasing distance to find nearest neighbor candidates [17], [18]. Amato et al. [16] proposed to index permutations using an inverted file, called MI-File. Tellez et al. [19] proposed a modification of the MI-File, called the neighborhood approximation index (NAPP). These methods do not rely on metric properties of the distance and can thus be applied to non-metric spaces.

A number of open-source packages have been developed in recent years for the nearest neighbor search task. The fast look-up of cosine and other nearest neighbors (FALCONN) method is based on an LSH variant by Andoni et al. [20]. The fast library for approximate nearest neighbors (FLANN) [21] package provides access to a collection of approximate nearest neighbor search algorithms. It automatically chooses the best of the available algorithms depending on the characteristics of the data set. Annoy [2] was built by Erik Bernhardsson to use at Spotify for music recommendations. The library recursively builds a neighborhood tree, given a set of points, which can be used at search time to quickly identify a subset of search candidates. It can be used for both user and item similarity search and have the ability to use static files as indexes.

2.2 Implications of Approximate Nearest Neighbor Methods

Good performance of approximate nearest neighbor methods has been demonstrated when data lie in a low-dimensional latent space. However, ANN algorithms provide no guarantees about their performance in many common uses of nearest neighbor search. The methods permit returning an arbitrary sample that satisfies the approximation condition. As such, ANN algorithms may be useful when the query point has a sufficiently large number of neighbors that satisfy the approximation condition. Then, the approximation algorithm is free to choose an arbitrary approximate nearest neighbor. However, when these conditions are not met, ANN methods will return unrelated far away objects that lead to low recall and may hurt further analytics performance.

An ANN algorithm provides no useful guarantees about its behavior or usefulness when it is utilized for the purposes that nearest neighbor algorithms are usually used. The algorithm fails to provide helpful predictions about its performance or any useful suggestion asymptotically (i.e., as the dimension grows) on any practical problem instance. Whether the proposed algorithms are useful depends on other properties, such as the density of neighbors or the minimum quality of a useful neighbor, which cannot easily be controlled by the ANN method. In some cases, ANN methods may return high enough quality results. However, the research community has failed to demonstrate that such algorithms have an advantage in these cases compared to traditional exact nearest neighbor methods, when considering the efficiency and effectiveness of the overall analysis.

3 RECOMMENDER SYSTEMS

Top-N recommender systems play an important role in e-commerce applications by providing personalized recommendations and content to users based on past purchase history and user feedback. The main objectives of $top-N$ recommender system are the accuracy of identifying the products or items a user will likely prefer, and the efficiency of the recommendation. Recommendation methods are classified into content-based filtering and collaborative filtering, which we will discuss in detail in the next section. Content-based filtering, also referred to as cognitive filtering, recommends items to users based on the similarities between the items and items which a user has previously rated.

3.1 Collaborative Filtering

Collaborative filtering (CF) techniques are part of the most popular and widely used real-world recommender systems. In CF, for each user, recommender systems recommend items based on how other similar users liked the item. Conceptually, they collect the past behavior of users and make rating predictions based on the similarity between user behavior patterns. The user's behavior is used to infer hidden user preferences and is usually represented by *explicit* (e.g., user ratings) and *implicit* user actions (e.g., clicks and query logs). Breese et al. [22] categorize CF systems into two subgroups, memory-based and model-based methods.

3.1.1 Memory-Based Methods

Memory-based methods operate by memorizing the rating matrix. They then recommend items based on the correlation between the queried user and the rest of the rating matrix. The memory-based CF methods are also called neighborhood-based methods, which find nearest neighbors that have rating preferences similar to those that of the target user or item. For example, if there are two users who have similar preferences and one of them gives a good rating to a product, then the other user is more likely to like that product. The methods operate over the entire user database to predict the results. The similarity is assessed by a cosine similarity-type measure, and unrated items are estimated using a k-nearest neighbor regression estimate. Memory-based methods are further classified into two categories, namely user-based k-nearest neighbor recommender (UserKNN) and item-based k-nearest neighbor recommender (ItemKNN).

In UserKNN, for a certain user, the method first identifies a set of similar users and then recommends $top-N$ items based on what items those similar users have purchased [23]. The user-based algorithms make predictions based on users' rating patterns similar to those of a target user u. Let $U(u;i)$ be a set of users who rate item i and have similar rating patterns as the target user u. Let w_{ij} denote the similarity measure between two items i and j. The predicted rating $\hat{r_{ui}}$ is calculated as

$$\hat{r_{ui}} = b_{ui} + \frac{\sum_{v \in U(u;i)} (r_{vi} - b_{vi}) w_{uv}}{\sum_{v \in U(u;i)} w_{uv}}, \tag{2}$$

where b_{vi} is a rating bias for user v's rating of item i and w_{uv} is the similarity between two users, u and v. Pearson's correlation coefficient or cosine similarity are measures often used to compute similarities between users in UserKNN methods.

Similarly, ItemKNN methods first identify a set of similar items for each of the items that the user has purchased, and then recommend $top-N$ items based on those similar items [23]. The item-based algorithms predict the rating for target item i of user u based on the rating patterns between items. Let $I(i;u)$ be a set of items that have rating patterns similar to that of i and have been rated by u. Let w_{ij} denote the similarity between two items i and j, and b_{ui} be a rating bias for user u's rating of item i. The predicted rating $\hat{r_{ui}}$ is then calculated as

$$\hat{r_{ui}} = b_{ui} + \frac{\sum_{j \in I(i;u)} (r_{uj} - b_{uj}) w_{ij}}{\sum_{j \in I(i;u)} w_{ij}}. \tag{3}$$

Cosine-based similarity measures are commonly used to calculate the similarity between items.

However, for the $top-N$ recommendation task, predicting the exact rating values is not necessary. Instead, it is more important to distinguish the importance of items that are likely to be appealing to the target user. Toward this goal, items are ranked by ignoring the denominator used for rating normalization. Since the denominator is non-negative and constant across all compared users or items, the same order is maintained while reducing the number of multiplications that must be performed to solve the problem. The predicted score in the user-based algorithms is then computed as

$$\hat{r_{ui}} = b_{ui} + \sum_{v \in U(u;i)} (r_{vi} - b_{vi}) w_{uv}, \tag{4}$$

and the predicted score in the item-based algorithms is computed as

$$\hat{r_{ui}} = b_{ui} + \sum_{j \in I(i;u)} (r_{uj} - b_{uj}) w_{ij}. \tag{5}$$

3.1.2 Model-Based Methods

Model-based methods first build a model based on the given rating matrix, and then recommend items to the users based on the fitted model. They build models by discovering patterns in the training data and use these models to make predictions on out-of-sample data. Model-based approaches, particularly latent factor models, have achieved state-of-the-art performance on large-scale recommendation tasks. The key idea of latent factor models is to factorize the user-item matrix into low-rank user and item factors that represent user preferences and item characteristics in a common latent space. The prediction for a user on an item can then be calculated as the dot product of the corresponding user factor and item factor vectors.

3.2 Matrix Factorization

Traditional CF models have been widely successful in quite a few fields. However, they all face problems such as data sparsity, scalability, and cold start. To alleviate the sparsity problem, many matrix factorization models have been developed in recent years based on popular dimensionality reduction techniques, such as singular value decomposition (SVD), principal component analysis (PCA), probabilistic matrix factorization (PMF), and non-negative matrix factorization (NMF). Matrix factorization is a powerful technique that is able to find hidden structure in the data. A key idea of SVD is to factorize an m-by-n matrix $\mathfrak{R}$ as the scaled inner product of two low-rank matrices with dimension f, i.e., one low-rank m-by-f matrix called the user-factor matrix, and another n-by-f matrix called the item-factor matrix. Each user u is thus associated with an f-dimensional vector $p_u \in \mathbb{R}^f$, and each item i is described by an f-dimensional vector $q_i \in \mathbb{R}^f$. In this case, the predicted rating $\hat{r_{ui}}$ is calculated as

$$\hat{r_{ui}} = b_{ui} + p_u q_i^T \,. \tag{6}$$

Conventional SVD is not well-defined in the presence of missing ratings (i.e., unknown values). Some earlier works have addressed this issue by filling missing ratings with a baseline estimation procedure [24]. However, this procedure has a drawback of creating a large, dense user-rating matrix, whose factorization becomes computationally not feasible. Recent works directly learn feature weights from known ratings through a suitable objective function to minimize prediction error [25]. For a $top-N$ recommendation task, we are interested only in the ranking of the items and do not care about accurate rating prediction. This grants us some flexibility. All missing ratings are considered negative user feedback, and they are imputed as close to zero. This modification can form a complete m-by-n matrix $\mathfrak{R}$, and the conventional SVD method can be applied to $\mathfrak{R}$,

$$\hat{r_{ui}} = U \sum V^T, \tag{7}$$

where U is an n-by-f orthonormal matrix, V is a m-by-f orthonormal matrix, and $\sum$ is an f-by-f diagonal matrix.

3.3 Ranking Based Collaborative Filtering

For the $top-N$ recommendation, it is important to consider the ranking of items. Learning to rank (LTR) is a machine learning technique for training the model in a ranking task. Models in the LTR paradigm can be grouped into three main categories, point-wise, pairwise, and list-wise approaches. Differences between these categories are due mainly to the form of the loss function and the type of training data used. Weimer et al. [26] proposed a ranking technique called CoFiRank that uses maximum margin matrix factorization to optimize the ranking of items. Liu et al. [27] developed EigenRank to decide the ranking of items using neighborhood-based approaches.

Hu et al. [28] proposed a weighted matrix factorization (WRMF) method for implicit feedback recommendation. WRMF includes all the unobserved user-item interactions as negative samples and uses a case weight to reduce the impact of uncertain samples. The algorithm scans through the entire data set for every iteration until convergence, which may prove computationally very expensive for data sets with a large number of users. The authors formulated a new square loss function that includes both preference and confidence metrics, which are optimized using the alternating least squares (ALS) method as follows,

$$\min_{y_* y_*} \sum_{u,i} c_{u,i}(p_{u,i} - x_u^T y_i)^2 + \lambda (\sum_u ||x||^2 + \sum_i ||y||^2), \tag{8}$$

where x_u and y_i are user and item latent vectors, respectively, $c_{u,i}$ is a confidence metric, and $p_{u,i}$ is a preference metric.

Although the approach of Hu et al. reduces the impact of missing data by relying on the confidence and preference metrics, it does not directly optimize its model parameters for ranking. Instead, it optimizes the prediction of whether an item is selected by a user or not. Rendle et al. [29] proposed a generic optimization framework called bayesian personalized ranking (BPR) that uses pairs of items (i.e., the user-specific order of two items) to discover personalized rankings for each user. It maximizes the likelihood of pairwise preferences between observed and unobserved items in implicit data sets. The objective function of BPR can be formulated as

$$\sum_{(u,i,j)\in D_S} \ln \sigma(\hat{x}_{uij}) - \lambda_\Theta \, ||\Theta||^2, \tag{9}$$

where λ_Θ are model specific regularization parameters, $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function to convert the margin to a probability, and D denotes the set of pairwise training examples,

$$D = \{(u,i,j)|i \in R_u^+ \wedge j \notin R_u^+\}, \tag{10}$$

where R_u^+ denotes the set of items that u has interacted with before. In each step, it randomly draws an observed interaction (u,i) and then selects an item j that u has not so far interacted with to constitute (u,i,j). This process of selecting j is also known as negative sampling. The optimization of BPR is usually done using stochastic gradient descent (SGD) with a small batch size, which reduces the computational time significantly.

Gantner et al. [30] extended BPR by generating negative items through random sampling of the missing entries in the training set. Shi et al. [31] combined CF with LTR methods to optimize the ranking of items. Futhermore, Shi et al. [32] combined rating and ranking-oriented algorithms with a linear combination function, and Liu et al. [33] extended probabilistic matrix factorization with list-wise preferences.

Weston et al. [34] proposed a rank-based weighting scheme, called the weighted approximate-rank pairwise loss (WARP), to penalize items at a lower rank. WARP repeatedly samples negative items until it finds one that has a higher score. Then, the number of sampling trials is used to estimate item ranks. WARP uses stochastic gradient descent and is an approximate approach that can estimate the rank function efficiently. Its main idea is to weigh pairwise violations depending on their position in the ranked list. Let N be the number of items and D_u be user u positive feedback. e can calculate users preferences for each item $j \in N$. Then, the WARP loss function is defined as

$$L_{WARP}(f_u, D_u) = \sum_{i \in D_u} \Theta \; rank_i(f_u), \tag{11}$$

where $rank_i(f_u)$ is the margin-based rank of item i, i.e.,

$$rank_i(f_u) = \sum_{j \notin D_u} I(f_u(j) \le f_u(i)), \tag{12}$$

where I$(\cdot)$ is the indicator function. Furthermore, $\Theta(\cdot)$ is a loss function which transforms the predicted rank of an item into a loss value,

$$\Theta(k) = \sum_{t=1}^{k} \alpha_t, \alpha_1 \ge \alpha_2 .... \ge 0. \tag{13}$$

Different settings of α_t allow the loss function to optimize different objectives. Minimizing Θ with $\alpha_t = \frac{1}{N}$ would optimize the N mean rank and minimizing Θ with $\alpha_t > \alpha_t + 1$ would assign higher importance to the top-ranked items. Additionally, Weston et al. [35] proposed the K-order statistic (K-OS) loss, which generalizes WARP by taking into account the set of positive examples during optimization, where K denotes the number of positive samples considered. For a given user u, let o be the vector of indices denoting the order of the positive examples in the ranked list. The K-OS loss function is defined as

$$L_{K-OS}(f_u, D_u) = \frac{1}{Z} \sum_{i=1}^{|D_u|} P\left(\frac{i}{|D_u|}\right) \Theta \, rank_{D_{u_{o_i}}}(f_u), \tag{14}$$

where Z = $\sum_i P(\frac{i}{|D_u|})$. K-OS degenerates to WARP when K = 1.

Ning et al. [36] proposed a novel sparse linear method that learns a coefficient matrix of item similarity for $top-N$ recommendation called SLIM. The basic idea in SLIM is that the user's preference over an item is modeled as the linear aggregation over the items that the user purchased before. It generates $top-N$ recommendations by aggregating user ratings. The rating for an item i is predicted as a sparse aggregation of existing ratings provided by the user,

$$\hat{r}_{ui} = r_u^T s_i, \tag{15}$$

where r_u^T is the uth row of the rating matrix R of size $m \times n$, and s_i is the sparse vector containing non-zero aggregration coefficients over all items. The sparse $n \times n$ matrix S of user ratings is learned by solving an $L1$ and $L2$ regularized optimization problem,

$$\begin{array}{ll} \underset{\mathbf{S}}{\textbf{minimize}} & \frac{1}{2}||\mathbf{R} - \mathbf{RS}||_F^2 + \frac{\beta}{2}||\mathbf{S}||_F^2 + \lambda||\mathbf{S}||_1. \\ subject\ to & S \geq 0, \\ & diag(S) = 0, \end{array} \tag{16}$$

SLIM has shown to be effective, but it can only capture relations between items that are co-purchased/co-rated. Cheng et al. [37] proposed LorSLIM, which uses the nuclear-norm to ensure the low-rank structure of the coefficient matrix. Liu et al. [38] adopted boosted regression trees to represent conditional user preferences in CF algorithms. AdaBPR [39] introduces a boosting technique to improve on BPR loss. SLIM and these algorithms have been found to outperform other methods, such as similarity or neighborhood-based methods, and are thus currently considered to be the state-of-the-art.

4 NEAREST NEIGHBOR SEARCH IN TOP-N RECOMMENDATION

The quality of recommendations produced by neighborhood-based methods depends on various factors, such as algorithm mechanism, data set sparsity, available user data, and the evaluation metric used. Hence, there has been much work focused on improving the quality of neighborhood-based models. Existing work on scaling up nearest neighbor approaches is partitioned into three categories, filtering-based approaches, approximate methods for nearest neighbor identification, and sampling-based approaches [6]. While learning their recommendation model, recommender systems often have to construct a search for all users or items in the system, a task known as the k-nearest neighbor graph construction.

4.1 K-Nearest Neighbor Graphs

K-nearest neighbor (k-NN) graphs are widely used in data mining and machine learning to solve real-world problems in collaborative filtering, information retrieval, and query search in web search engines, among others. A k-NN graph is a directed graph, $G = (D, E)$, where D is the set of nodes (i.e. data points) and E is the set of links. Node d_i is connected to node d_j if d_j is one of the k-nearest neighbors of d_i. The choice of k is crucial to achieve good performance in the analysis. A small k value makes the graph too sparse or disconnected so that hill-climbing methods frequently get stuck in local minima. Choosing a large k value gives more flexibility during the run time, but it consumes more memory and makes offline graph construction more expensive.

4.2 K-NNG Construction Methods

Exact k-NN graph construction has been extensively studied in the literature. A brute-force construction method has a time complexity of O(dn^2), where n is the number of objects and d is the dimensionality of each object vector. Several methods have been proposed to avoid this time complexity. Parades et al. [40] proposed a more efficient algorithm to construct the exact k-NN graph that has an empirical time complexity of O($n^{1.27}$) in low-dimensional data and O($n^{1.90}$) in high-dimensional data. In spite of extensive research on this topic, the time complexity of graph construction methods increases exponentially with an increase of dimensionality or linearly with increasing size of data. Recent research has therefore focused more on approximate neighborhood graph construction. Applying ANN search methods is one way of constructing an approximate k-NN graph. In the construction of approximate k-NN graph, each data point is treated as a query point, and the ANN search algorithm retrieves k-nearest neighbors for each query point by performing a search.

A popular approach for ANN search has been LSH. However, the LSH method is computationally expensive for achieving accurate approximation. Park et al. [5] developed Greedy Filtering, an approximate filtering-based approach, which filters item pairs that do not have any matching dimensions with large values. Malkov et al. [41] proposed a greedy approximate k-NN graph construction method that organizes data into a navigable small world graph structure suited for distributed approximate k-nearest neighbor search in metric spaces. Dong et al. [12] proposed K-Graph, an approximate k-NN graph construction method also called NN-Descent, which follows an iterative neighborhood improvement strategy. It is based on the idea that similar objects are likely to be found among the neighborhoods of objects in a query object's neighborhood.

Anastasiu et al. [6] developed an exact and approximate k-NN graph construction method called L2KNNG and L2KNNGApprox. L2KNNG solves the exact cosine similarity k-NN graph construction problem efficiently by effectively ignoring insignificant object pair comparisons. The L2KNNG algorithm adopts a two-phase approach. The first step uses a fast method (L2KNNGApprox) to build an initial approximate graph. The algorithm finds k similar objects for each object. These objects may not necessarily be the true nearest neighbors. In the second step, the algorithm examines all the objects again and gradually updates each object by finding k most similar objects until its true neighbors are found. Similar to the Greedy Filtering method by Park et al. [5], L2KNNGApprox first builds a set of initial neighborhoods using a value-based sorted inverted index to efficiently identify candidate objects with common high-value features with the query; then, similar to the K-Graph method by Dong et al. [12], it iteratively enhances the initial k-NN graph by looking for new candidates in each neighbor's neighborhood. Experiments have shown that L2KNNG and L2KNNGApprox outperform alternatives in both approximate and exact nearest neighbor graph construction [6]. As such, in this research, we chose L2KNNG and L2KNNGApprox as the state-of-the-art methods we employed in our error propagation experiments.

5 EXPERIMENTAL DESIGN

It is important that we choose a performant algorithm for any analytics problem. In this section, we conduct experiments on different data sets to understand the performance of current state-of-the-art recommender system algorithms. This allows us to compare different methodological approaches for the recommendation task and also to point out how ItemKNN using an exact *k*-NN graph construction method compares to state-of-the-art recommender system methods.

5.1 Data Sets

We present numerical experiments to evaluate the performance of recommendation methods on six different real data sets whose statistics are shown in Table 1. These data sets provide snapshots of real users' behavior and are widely used in the research literature for benchmarking recommender system algorithms. Some data sets are very sparse, with some of the users providing ratings for two items or less. In these cases, the recommendations for these users will likely be less precise.

Table 1: Statistics of the Data Sets

Data set	**#Users**	**#Items**	**#Trns**	**Rsize**	**Csize**	**Density**
BX	3,586	7,602	84,981	23.70	11.18	0.31%
ML100K	943	1,682	100,000	106.04	59.45	6.30%
ML1M	6,040	3,706	1,000,209	165.60	269.89	4.47%
ML10M	69,878	10,677	10,000,054	143.11	936.60	1.34%
ML20M	138,493	26,744	20,000,263	144.41	747.84	0.54%
Netflix	336,914	17,770	51,937,015	154.16	2922.74	0.87%

In each data set, the "#Users," "#Items" and "#Trns" columns are the number of users, number of items, and the number of transactions, respectively. The "Rsize" and "Csize" columns are the average number of ratings for each user and for each item (i.e., row and column density of the user-item matrix), respectively. The "Density" column is the density of each data set (i.e., Density = #Trns/(#Users x #Items)).

Movielens: Movielens is one of the popular data sets that have been widely used for offline experimental evaluation of recommender systems. The ML100K, ML1M, ML10M and ML20M data sets were obtained from the MovieLens research project. These are some of the publicly available data sets collected by GroupLens [42], a research lab at the University of Minnesota, from their active online movie recommendation system.

Netflix: In 2004, the online movie rental company Netflix announced a competition for improving its recommendation system. For the purpose of the competition, Netflix released a data set containing 480,000 user ratings of over 17,700 movies. Ratings are between 1 and 5 stars for each movie. The data set is very sparse (i.e., users mostly rated a small fraction of the available movies). For our experiments, we have extracted a subset from the Netflix Prize data set [43] such that each user has rated at least 30 movies and at most 500 movies.

Book Crossing: The BookCrossing website allows a community of book readers to share their interests in books and to review and discuss books. The system lets its users rate books on a scale of 1 to 10 stars. The specific data set that we used was collected from the BookCrossing website in a 4-week crawl during August and September 2004 by Cai-Nicolas Ziegler [44], [45]. The data set contains 105,283 users and 340,556 books, and the average number of ratings for a user is 10. This data set is even more sparse than the Netflix data set, as there are more items and fewer ratings per user. We only used a subset of the data set, in which each user has rated at least 20 items and each item has been rated by at least 5 users and at most 300 users.

As we are not interested in the rating prediction problem, we transformed the ratings for all data sets into implicit feedbacks by keeping a value of 1 if there is a transaction between the user and the item, and setting the value to 0 otherwise. In other words, we binarized the rating matrix, replacing all existing non-zero values in the matrix by ones.

5.2 The Sparsity Problem

A common problem in machine learning is sparse data. Data are considered sparse when a large number of values in a data set are missing or have a zero value, which is a common phenomenon in general large scale analytics. This problem is commonly referred to as the sparsity issue. A matrix is considered extremely sparse when there are very few elements in a matrix whose value is not non-zero. In the recommendation domain, it is also known as the cold-start recommendation problem, referring to the act of providing recommendations for new users who have rated very few items (i.e., they have very few non-zeros in their profile). Sparse data sets affect the performance of top-N recommendation algorithms by making inaccurate predictions.

Given the large size of data sets such as ML20M and Netflix, naive representation as a dense matrix would result in high computation times and likely memory overruns. To efficiently operate on the sparse data, the ratings corresponding to each user and movie are stored in a compressed sparse row (CSR) matrix. The size of the sparse matrix depends on the number of non-zero elements, rather than the dimensionality of the dataset. Fig. 1 shows the distribution of non-zero elements over rows and columns for all six data sets. For the ML10M and ML20M data sets, some of the items have very few ratings. In the BX data set, a single user has rated 2238 items. For most of the data sets, only some of the items are rated frequently, and the vast majority of items are rated rarely. These frequently rated items are referred to as popular items.

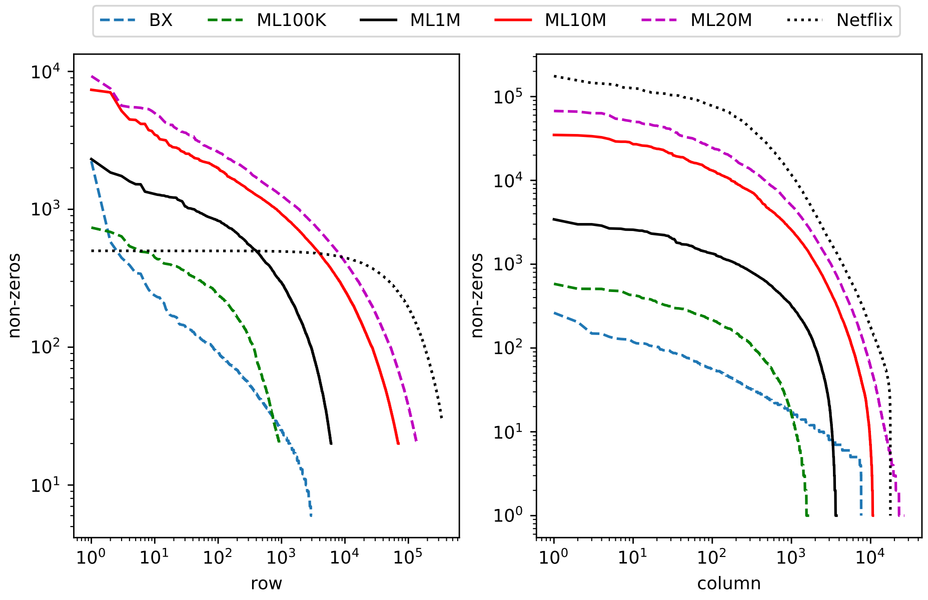
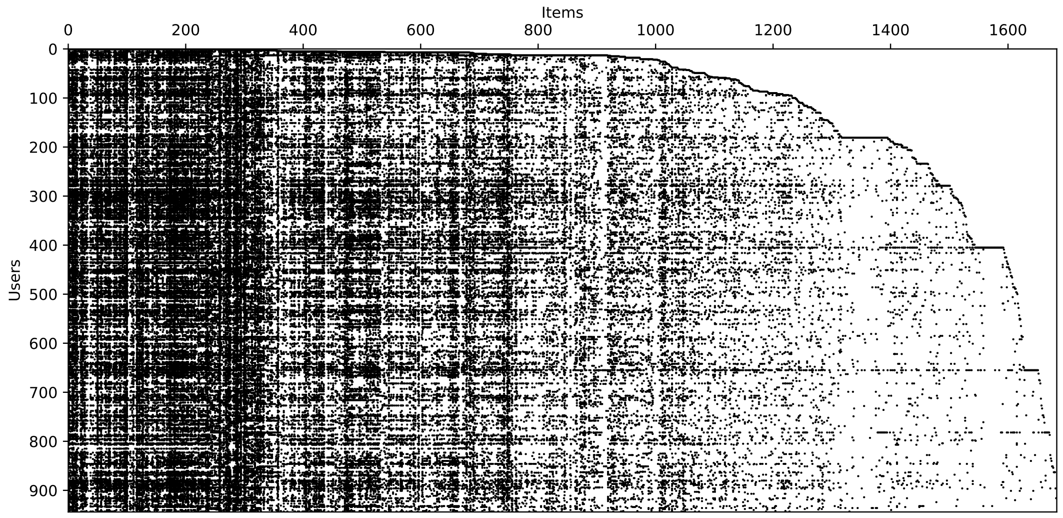
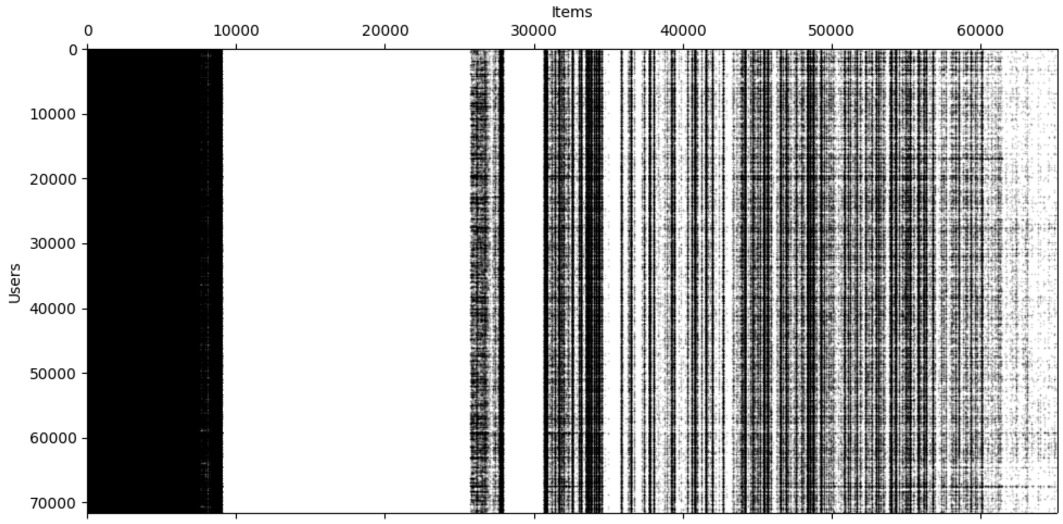


Fig. 1: Distribution of non-zero elements over rows and columns.

Fig. 2 shows the placement of non-zero elements across the rows and columns of the matrix for the ML100K and ML10M data sets (dark dots represent non-zero elements). Similar additional figures for the remaining data sets are included in Appendix A.



(a) ML100K



(b) ML10M

Fig. 2: Distribution of non-zero elements for ML100K and ML10M.

5.3 Evaluation Methodology

We applied 5-fold Leave-One-Out cross-validation (LOOCV) to evaluate the performance of the $top-N$ recommender system algorithms. In each run, each of the data sets was split into a training and test set, by using a random selection approach. We randomly selected one of the non-zero entries (i.e., item id) for each user and placed it in the test set. The rest of the data comprised the training set. The training set was used to train a model, then a ranked list of $top-N$ items for each user was generated using that model. The model was then evaluated by comparing the recommendation list of each user and the item for the particular user in the test set. Throughout this research, we used the same set of training and test sets to conduct experiments across all algorithms.

5.3.1 Recommendation System Evaluation

We measured recommendation quality using hit rate (HR), which is defined as

$$HR = \frac{\#hits}{\#users}, \tag{17}$$

where $\#users$ is the total number of users, and $\#hits$ is the number of users whose item in the testing set was recommended (i.e., hit) in the $top-N$ recommendation list. An HR value of 1.0 indicates that the algorithm is able to recommend all of the hidden items in the test set, whereas an HR value of 0.0 denotes that the algorithm is not able to recommend any of the hidden items. Furthermore, the efficiency of a recommendation engine was measured as the overall execution time needed to find recommendations for all test users, measured in seconds.

5.3.2 Nearest Neighbor Search Evaluation

Exact nearest neighbor search methods return the true nearest neighbors. ANN methods, on the other hand, return only some of the nearest neighbors. We measured the effectiveness of these methods as using recall, which is defined as

$$\text{recall} = \frac{1}{n}\sum_u \frac{\#found_u}{\#neighbors_u}, \tag{18}$$

where $\#found_u$ is the number of true neighbors of user u that were found by the search algorithm and $\#neighbors_u$ is the number of true neighbors user u has (at times $\#neighbors < k$), and n is the total number of users. By definition, exact methods have a recall of 1.0. We use the neighbors found by the exact method as the set of true neighbors needed to compute recall in approximate methods.

The efficiency of a nearest neighbor search method was measured as the overall execution time needed to find neighbors for all test objects, measured in seconds. Additionally, we consider the speedup of an approximate method over its exact counterpart, which is defined as

$$\text{speedup} = \frac{\text{time}(exact)}{\text{time}(approx)}, \tag{19}$$

where $\text{time}(exact)$ and $\text{time}(approx)$ are the execution times of the exact and approximate methods under comparison, respectively. Note that speedup of 1.0 indicates the methods finished in the same amount of time, speedup $>$ 1.0 indicates that the approximate method was faster than the exact method, and speedup $<$ 1.0, which is also called slowdown, indicates that the approximate method was slower than the exact method.

5.3.3 Hypothesis Testing

To draw a reliable conclusion from the experiments, we performed significance testing on the experimental results. A standard tool for hypothesis testing is p-value (i.e) the probability of predicting a particular score by chance. If the p-value corresponding to the difference in mean weight between two groups of participants is less than the chosen significance level (e.g, $p < \alpha$, α = 0.001), then we can reject the null hypothesis (i.e., the hypothesis that no difference exists between the comparison methods), and our result is deemed “statistically significant”. If the p-value is above the significance level, we cannot reject the null hypothesis and our result is “not statistically significant”. This procedure is called as statistical hypothesis testing.

5.4 Baseline Methods

We performed experiments with multiple baseline methods, including item neighborhood-based collaborative filtering (ItemKNN), recommender systems based on matrix factorization (MF), bayesian personalized ranking (BPR), weighted approximate-rank pairwise (WARP) loss, k-order statistic (K-OS) loss, and SLIM. ItemKNN is a classic collaborative filtering method that recommends similar items based on items previously rated by users [23]. We adapt it for implicit feedback data by predicting the rating for target item i of user u based on the rating patterns between items using Equation 5. MF [46] is a well-developed and commonly-used technique for $top-N$ recommendation. For the MF method, we used the implicit library [47], which implements MF with an ALS learning method. BPR [29] optimizes the area under the ROC curve (AUC) and is another widely-used baseline. We used WARP loss from the WSABIE [34] package and K-OS codes from their authors [35]. These codes have been integrated into the lightfm [48] library, which we used to conduct our experiments. SLIM generates recommendation results by aggregating user purchase/rating profiles. We used a SLIM software variant developed by the authors of [36]. All these methods constitute the current state-of-the-art for $top-N$ recommendation task.

6 EXPERIMENTAL RESULTS

This section is focused on answering the following questions:

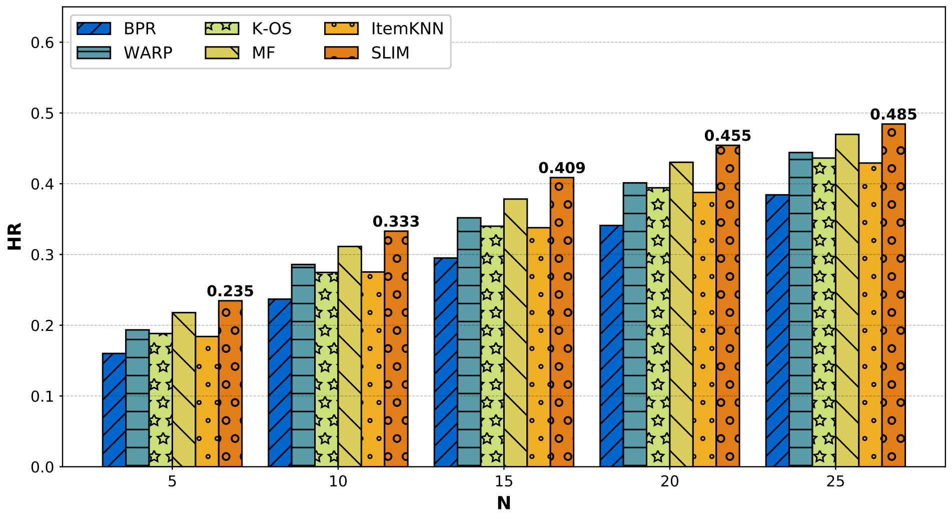
- How does the ItemKNN method perform compared to existing current state-of-the-art $top-N$ recommendation methods?
- How effective are ANN methods compared to exact nearest neighbor search?
- What is the trade-off between effectiveness (high recall) and efficiency (low execution time) of the approximate method in the $top-N$ recommendation?

All the above questions are answered by empirical evidence across the six different data sets described in Section 5.1.

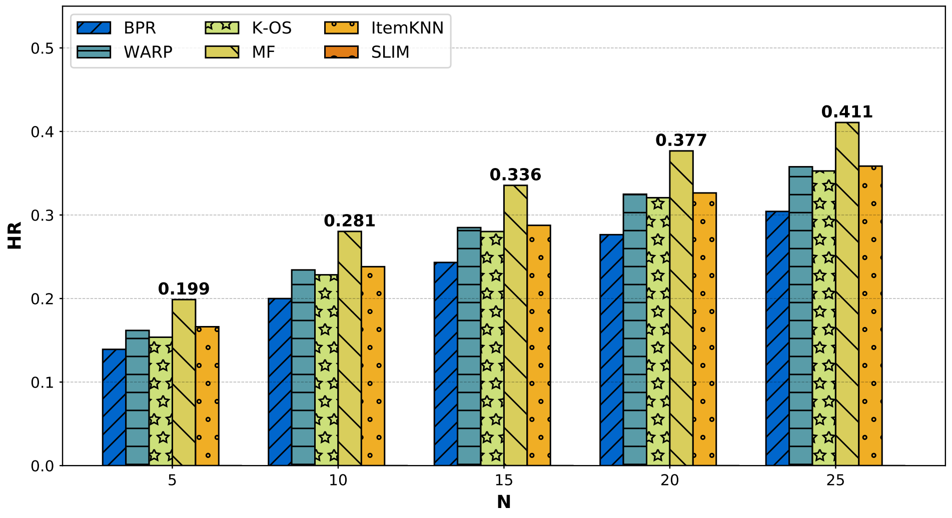
6.1 Top-N Recommendation Performance Comparison

We begin by investigating the recommendation quality of ItemKNN compared with five baselines over the six data sets. Fig. 3 depicts the best HR results of different $top-N$ recommendation algorithms for the ML100K (Fig. 3a) and ML10M (Fig. 3b) data sets. Additional figures for the remaining data sets show similar characteristics and are included in Appendix B.

All experiments were executed five times, with random seeds, and the averaged results are reported. For SLIM, we reported only the experimental results for the BX, ML100K, and ML1M data sets because it was unable to run this on a larger data set. The results showed that, SLIM outperformed all other methods for all data sets it ran on. For the Netflix data set, ItemKNN achieved a significantly better performance than other methods, and BPR, WARP, K-OS methods showed similar performance. MF performed better than ItemKNN on the ML100K, ML1M, ML10M, ML20M data sets, but worse on the BX and Netflix data sets.



(a) ML100K



(b) ML10M

Fig. 3: Performance of recommendation algorithms for different N values for ML100K and ML10M.

As shown in Fig. 3a and Fig. 3b, recommendation quality improved when more items were considered, as N was increased from 5 to 25. Among the three MF-based models, BPR and WARP had similar performance on most of the data sets. We found that the performance of ItemKNN was better than that of some of the state-of-the-art methods but worse than SLIM. We can observe the following recommendation performance trend: ItemKNN > BPR, ItemKNN > WARP, ItemKNN > K-OS for all six data sets. The values of the hyper-parameters that lead to the best resulting performance in each method are shown in Appendix C.

Table 2 presents the performance difference between ItemKNN and the best performing baseline. Columns N=5 through N=25 show the performance difference (in terms of HR) between ItemKNN and the best performing baseline for that N value. For example, 0.008 for BX in the N=5 column indicates that HR@5 (when top-5 items are recommended) was calculated and the difference between the best HR across all methods and the HR of ItemKNN was 0.008.

Table 2: Comparison of ItemKNN and the Best Performing Baseline

Data set	**N = 5**	**N = 10**	**N = 15**	**N = 20**	**N = 25**	*p-value*
BX	0.008	0.009	0.014	0.016	0.019	0.0032
ML100K	0.050	0.057	0.070	0.066	0.055	<0.0001
ML1M	0.048	0.063	0.069	0.070	0.073	<0.0001
ML10M	0.032	0.042	0.047	0.050	0.052	0.0002
ML20M	0.033	0.040	0.045	0.047	0.048	0.0001
Netflix	0.0	0.0	0.0	0.0	0.0	-

Further, to validate the effectiveness of ItemKNN, we also conducted two-sided hypothesis tests for the null hypothesis that the average of the performance difference between ItemKNN and the best performing baseline is zero. We have set the significance level to 10^{-4} instead of 0.01 because we measure the effectiveness difference between ItemKNN and the best performing baseline and not the hit rate results, and the effective range of this difference is $O(10^{-2})$. Paired t-tests were carried out on the performance difference of both methods at various levels of N = $\{5, 10, 15, 20, 25\}$.

As we can see from Table 2, given a significance level $\alpha = 0.0001$, p-values indicate that there is no significant difference between the methods for the BX, ML10M, and ML20M data sets. Based on these comparisons, we conclude that the basic ItemKNN algorithm provides a reasonably good predictions for the $top-N$ recommendation task.

6.2 ItemKNN Recommendation for Different N Values

In this section, we report the comparison of approximate and exact ItemKNN search performance. The objective is to investigate the impact of search performance (recall) on recommendation quality (HR). The ItemKNN code was implemented in Python using cosine similarity k-nearest neighbor graphs identified using the L2KNNG (exact) and L2KNNGApprox (approximate) [6] methods.

For the exact ItemKNN experiment, we first tuned the parameter of k by testing all values in the set $\{5, 10, 20, 25, 50, 75, 100, 200, 500, 1000\}$ to optimize HR. We found the best values to be $k = 10$ for BX, ML100K and ML1M and $k = 20$ for the rest of the data sets. We observed that, even with such small values for k, ItemKNN provides reasonably accurate recommendations. This is particularly important since small values of k lead to fast results (i.e., low computational cost).

The L2KNNGApprox parameters α and γ influence its effectiveness and efficiency; α indicates the number of candidates to consider for each query during the search, as a multiple of k, and γ indicates the number of iterations of the neighborhood graph enhancement step in the algorithm.

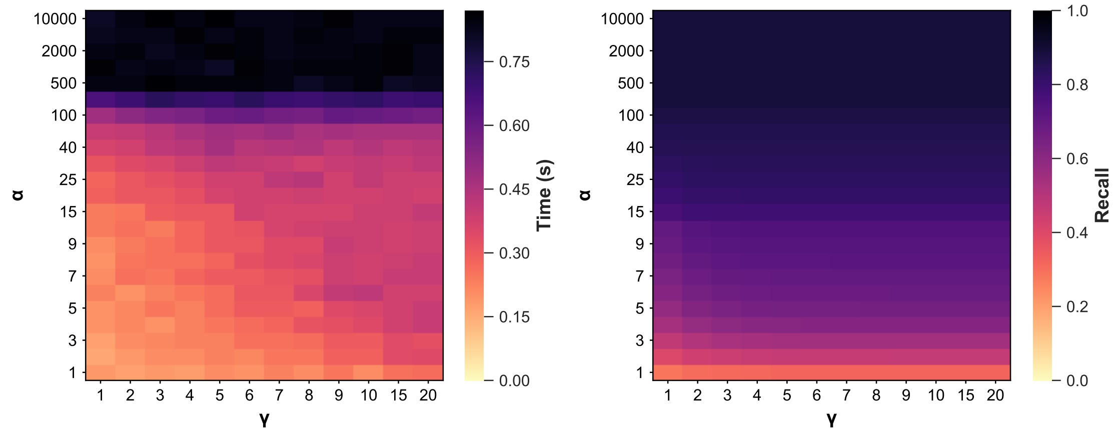
For the approximate ItemKNN experiment, we used the same setting of the parameter k as in ItemKNN and tuned α and γ by performing a grid-search through the sets $\alpha \in \{1, 2, \ldots, 10, 15, \ldots, 100\}$ and $\gamma \in \{1, 2, \ldots, 20\}$. The L2KNNGApprox search using different α and γ values produced different k-NN graphs with different recall values. Higher α values lead to more similarity evaluations and higher γ values lead to more iterations of the neighborhood enhancement step, both leading to higher recall but slower computation.

To compare the methods, we manually selected approximate k-NN graphs with recall close to $\{0.50, 0.55, \ldots, 0.95\}$. This procedure was repeated over all five data set splits, and evaluation results were averaged. For some data sets, the highest recall was near or above 0.95, while for others, the highest achieved recall was only 0.85. We denote those results in tables by a dash (-) and in figures by missing data points or bars.

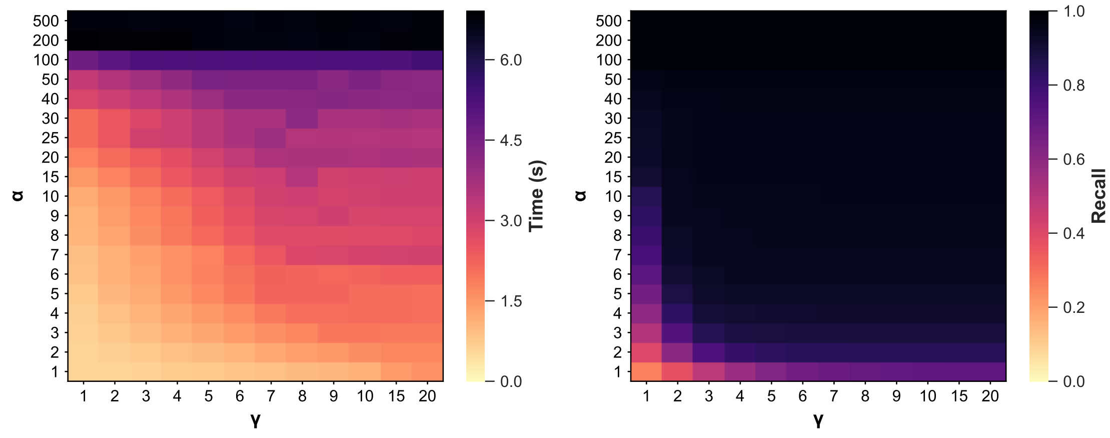
6.2.1 Parameter Sensitivity

Fig. 4 shows the effects of α and γ of the L2KNNGApprox method in terms of similarity computation time and recall for BX and ML1M data sets. Even with the low parameter setting, L2KNNGApprox was able to achieve $\geq 95\%$ recall for the ML1M dataset. By putting in more computation, we were able to boost recall for the more difficult data sets (BX, ML20M) to close or above 85%. Additional figures for the remaining data sets are included in Appendix E. When tuning parameters, the recall improves only up-to a point (highest recall). We see that beyond a critical point, recall only improves marginally. Also, there is a trade-off between achieving the highest recall and the time taken to achieve them.

An interesting aspect to note in Fig. 4 is the difficulty with which high recall can be achieved for some data sets. While most values for $\alpha > 2$ and $\gamma > 2$ give high recall for the ML1M data set (indicated by black tiles in the right sub-figure of Fig. 4b), only $\alpha > 100$ can lead to high recall for the BX dataset, which coincides with high execution times (indicated by black tiles in the left sub-figure of Fig. 4a). This further underscores the lack in reliability when employing approximate search algorithms. In general, they cannot provide quality guarantees and tend to be hard to tune to achieve high recall.



(a) BX

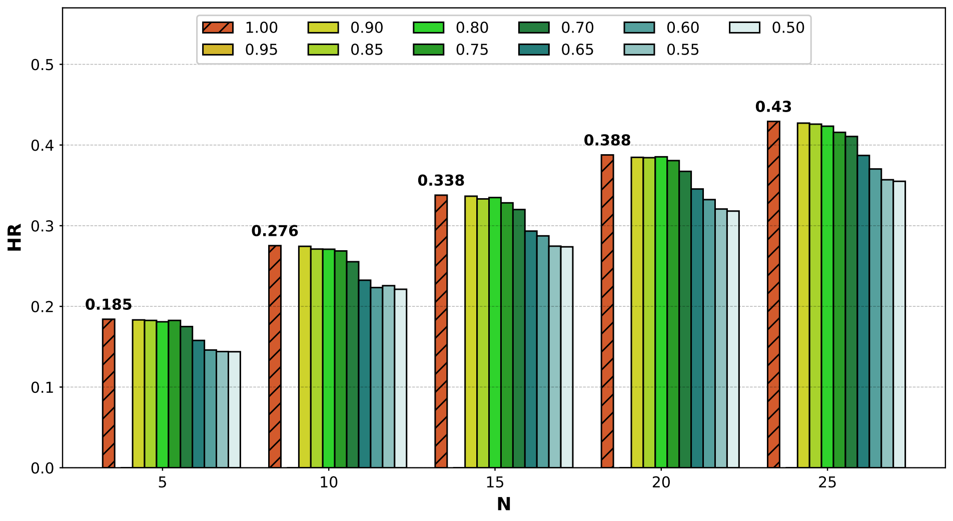


(b) ML1M

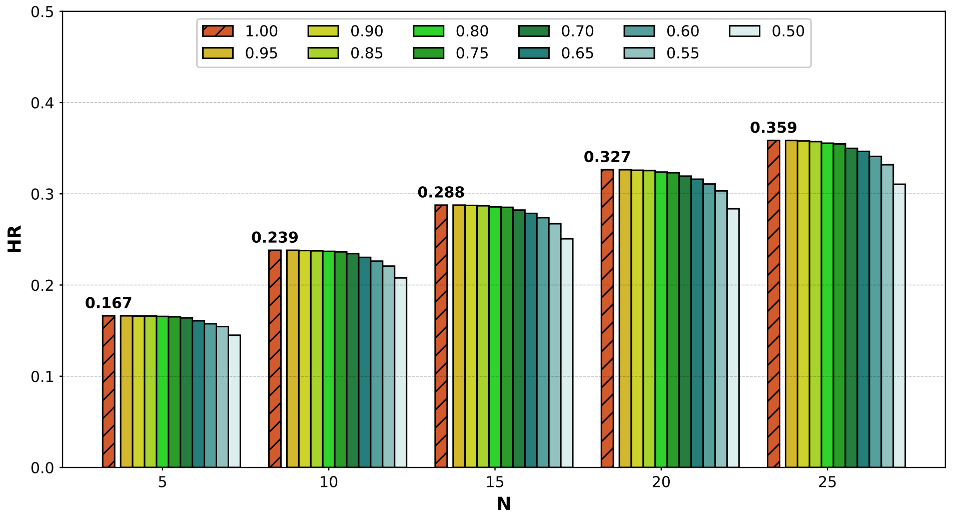
Fig. 4: Effects of α and γ on efficiency (left) and effectiveness (right) for BX and ML1M.

6.2.2 Recommendation Performance

Fig. 5 reports the best HR results for approximate and exact search-based ItemKNN for different values of N $\in \{5, 10, 15, 20, 25\}$ for the ML100K (Fig. 5a) and ML10M (Fig. 5b) data sets. Additional figures for the remaining data sets show similar results and are included in Appendix B. As we can see from the figure, the performance of the approximate method decreases as recall decreases.



(a) ML100K



(b) ML10M

Fig. 5: ItemKNN recommendation for different N values for ML100K and ML10M.

Recall has a significant impact on the recommendation quality for both data sets, as different values of recall lead to substantially different values of HR. Despite this variability, if the recall is 0.80 or higher, then the approximate method provides good overall recommendation performance.

6.2.3 *Relationship Between Recall and HR*

During our experiments, we found that the performance of the approximate search-based ItemKNN method depends on the recall value of the search result. Here we discuss how recall affects the HR of the recommender system. Fig. 6 plots the relationship between recall and normalized HR on N = {5, 10, 15, 20} for the six data sets, with recall varying from 0.50 to 1.00. HR values have been normalized using min-max normalization with the minimum value of 0 and the maximum value set to the HR obtained using the exact search-based ItemKNN. The normalization allows us to compare the relative decrease in performance for different recall values across the six data sets.

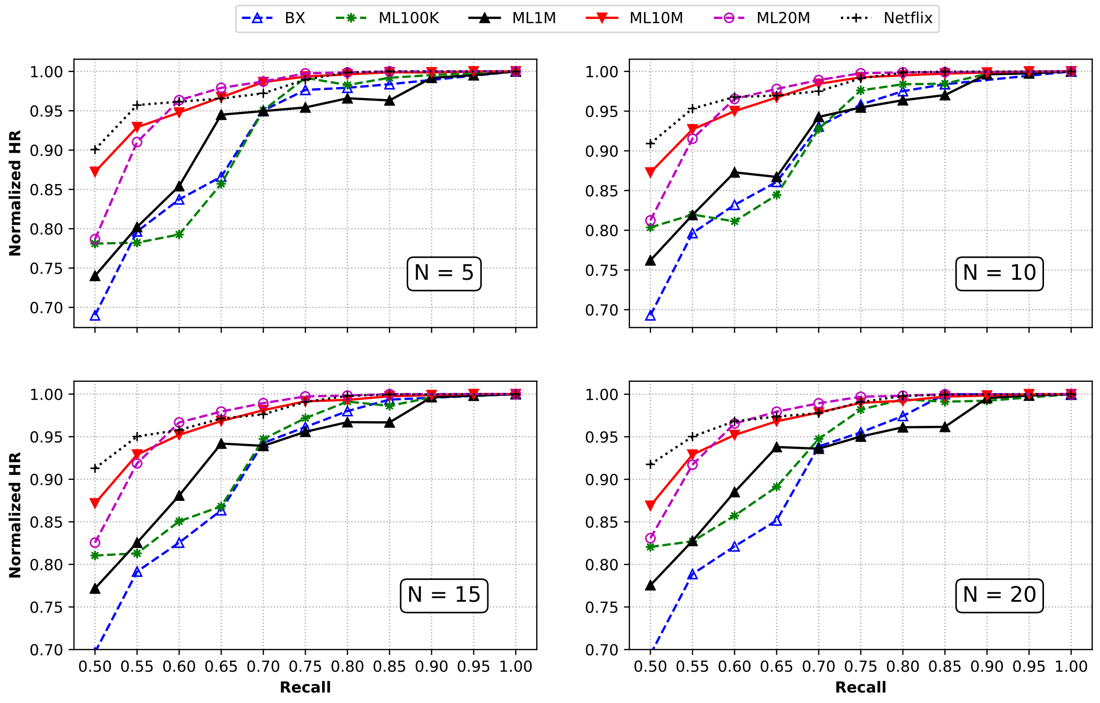


Fig. 6: Recall versus normalized hit rate (HR) for various N values.

We can see from Fig. 6 that, for most datasets, the recall value has to be at least 80% for an approximate method to be able to achieve a similar performance as its exact counterpart (normalized HR = 0.97). For recall levels between 70% and 80%, there is a decrease in the performance, but it is not significant. Performance decreases rapidly for recall levels below 70%.

6.2.4 Relationship Between Recall and Speedup

In this section, we evaluated the trade-off between search quality and efficiency. Fig. 7 shows the relationship between recall and speedup for the six data sets, with recall varying from 0.50 to 0.95. We can see from the figure that, as recall increases, speedup decreases. For recall levels above 80%, speedup is at times lower than 1.0, indicating the approximate search took longer than the exact one. By losing little recommendation performance, better speedup (0.5x - 2.0x) was achieved for recall levels between 70% and 80%. Though speedup values were higher for recall levels below 70%, the associated recommendation performance was significantly worse for most data sets.

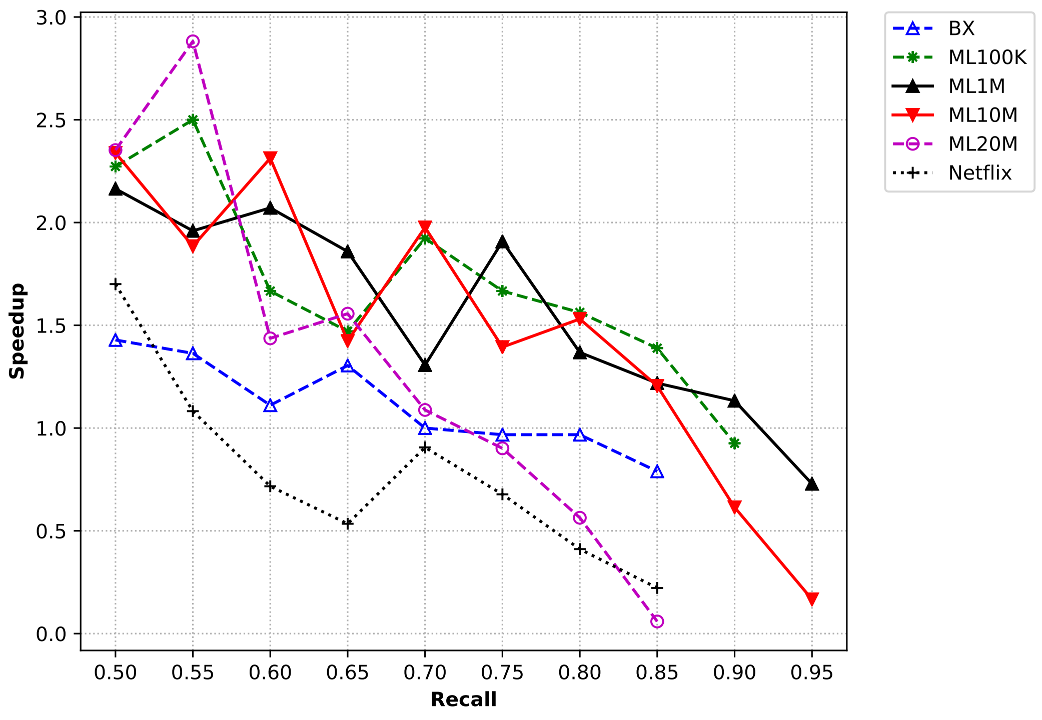


Fig. 7: Recall versus speedup for different data sets.

6.3 Statistical Analysis of the Approximation Effect on Recommendations

As a way to identify the significance of the approximation effect on the recommendation results, we computed Welch's t-test, or the unequal variances t-test, between recommendations based on the exact search results and those based on approximate ones. Welch's t-test is a two-sample location-based statistical test that tests whether two populations are equal in the expectation, i.e., they have equal means. The method is an adaptation of Student's t-test designed to account for differences in the variance of the two samples being tested. In our experiment, samples are the HR values of recommendations obtained with the item-based collaborative filtering recommender when finding neighbors either with an exact nearest search method or an approximate one. For the approximate method, we restricted neighbor search results to those near a given recall value and set the number of recommendations, N, to 10. For each data set, we executed t-tests at each recall value in the range $[0.50, 0.95]$, in increments of 0.05.

The null hypothesis in our test is that the recommendation performance when relying on approximate search methods is as good as that when relying on exact methods. Let r_a and r_e be the hit rates of the approximate and the exact experiments, respectively. Welch's t-test defines the statistic t as,

$$t = \frac{\overline{r_a} - \overline{r_e}}{\sqrt{\frac{s_a^2}{n_a} + \frac{s_b^2}{n_b}}}, \tag{20}$$

where $\overline{r_a}$, s_a^2, and n_a are the approximate sample mean, variance, and size, respectively, and $\overline{r_e}$, s_e^2, and n_e are similarly defined for the exact sample. In each experiment, we executed a two-tailed test and analyzed the probability value (p-value) of the null hypothesis being true, i.e., the probability that the absolute value of the sample mean difference between the results obtained with approximate search results and those obtained with exact search results is greater or equal than that of the observed results. When setting a confidence level, e.g., $C = 95\%$, p-values smaller than $(1.0 - C = 0.05)$

imply that the null hypothesis may be rejected and the alternative hypothesis accepted. Alternatively, if the p-value is above the significance level ($\alpha = 0.05$), we fail to reject the null hypothesis and cannot accept the alternative hypothesis. In our case, the alternate hypothesis states that the recommendation performance when relying on approximate search methods is significantly worse than that when relying on exact search methods.

Table 3 shows the results of our statistical analysis across all data sets. Bold values indicate p-values lower than our significance level, i.e., recall values for which we can reject the null hypothesis. Approximate algorithms provide significantly worse performance for recall values with bold p-values in each data set. Specifically, recall levels below 70% have a p-value lower than the cut-off threshold α.

Table 3: Statistical Analysis of L2KNNGApprox for Top-N Recommendation

	BX	**ML100K**	**ML1M**	**ML10M**	**ML20M**	**Netflix**
recall			*p-value*			
0.95	-	-	0.8979	0.9788	-	-
0.90	-	0.9173	0.8546	0.6033	-	-
0.85	0.6824	0.5443	0.1221	0.3538	0.6058	0.9575
0.80	0.5160	0.6013	0.0557	0.1281	0.4225	0.8001
0.75	0.2688	0.4699	**0.0462**	**0.0464**	0.1781	0.1137
0.70	0.0937	**0.0226**	**0.0091**	**0.0008**	**0.0014**	**0.0011**
0.65	**0.0041**	**0.0015**	$\mathbf{6.36 \times 10^{-5}}$	$\mathbf{2.04 \times 10^{-6}}$	$\mathbf{2.79 \times 10^{-6}}$	**0.0002**
0.60	**0.0018**	**0.0003**	$\mathbf{4.41 \times 10^{-5}}$	**0.0011**	$\mathbf{3.09 \times 10^{-6}}$	**0.0042**
0.55	**0.0006**	**0.0002**	$\mathbf{3.12 \times 10^{-6}}$	$\mathbf{3.01 \times 10^{-8}}$	$\mathbf{1.16 \times 10^{-7}}$	$\mathbf{5.64 \times 10^{-5}}$
0.50	$\mathbf{4.79 \times 10^{-5}}$	**0.0015**	$\mathbf{3.39 \times 10^{-6}}$	$\mathbf{1.63 \times 10^{-6}}$	$\mathbf{3.99 \times 10^{-10}}$	$\mathbf{5.55 \times 10^{-6}}$

6.3.1 Discussion

For p-values of 0.05 and above, our analysis cannot conclude that the performance of the approximate and exact method is the same, or even similar. For p-values between 0.05 and 0.001, the evidence is not strong enough to conclusively state that the performance of the approximate method is worse than that of the exact method, but the result acts as a suggestion, providing a middle-ground between acceptance and rejection. For p-values of 0.001 and below, we can state with high confidence that equal performance of the approximate and exact methods is extremely unlikely and reject the null hypothesis in favor of the alternative hypothesis. We would have to conclude here that approximate method effectiveness remains unproven. Analysis results show that the approximate methods yield better performance for low-dimensional data sets. As the recall value ranges between 0.95 - 0.50, there is a linear drop in *p*-values for small data sets and an exponential drop for large data sets. For example, ML20M shows a significant difference in *p*-values across decreasing recall levels, especially below 70%, while the decrease is much slower for ML100K.

As shown in Fig. 8, building approximate *k*-NN graphs with high recall generally incurs high computational costs, at times much higher than even the exact search method (denoted by recall of 1.0). The ML20M and Netflix data sets are intrinsically more difficult than the other data sets to search. Approximate methods sacrifice some search accuracy to lower the execution cost. In order to increase accuracy, one must set the method parameters high, which will result in high execution cost. The parameter analysis of the approximate methods is shown in Appendix E.

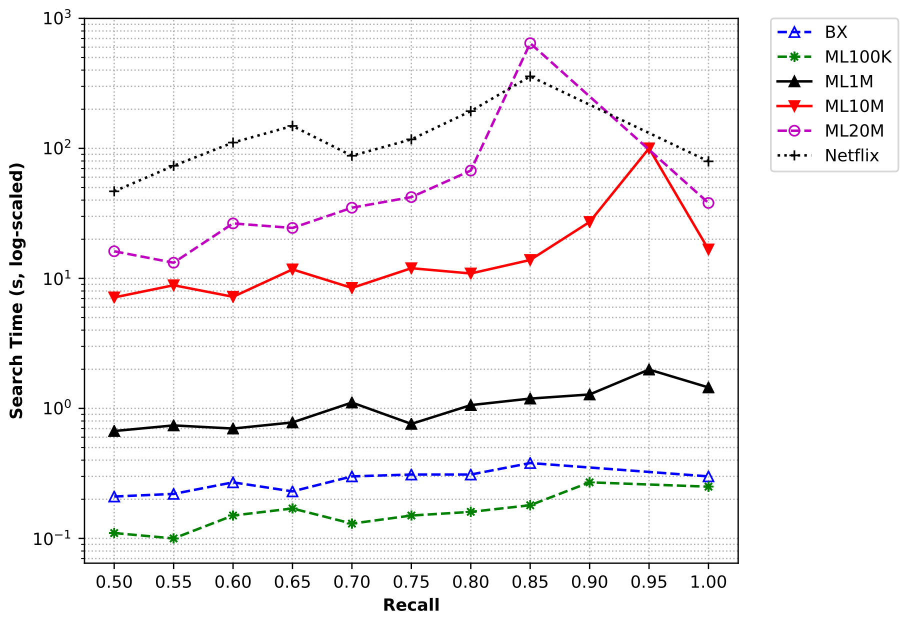


Fig. 8: Recall versus search time (log-scaled) for different data sets.

In real-world scenarios, due to the massive size of the data sets being analyzed, recommender systems present a few items to each user and the set of items to be recommended are chosen among the top-most items often identified off-line. We believe this study to be promising for industrial applications for the following reasons:

- For applications requiring interactivity, where response time is important, the α and γ parameters in the approximate method allow adjusting the trade-off between accuracy (effectiveness) and response time (efficiency).
- We have shown that approximate approach is not efficient in the context of big data, as it incurs a high computational cost to provide high recall values.

Based on our comprehensive evaluation, we provide the following recommendations for researchers or practitioners.

- When high effectiveness is required, it is best to use an efficient exact search method, such as L2KKNG, as the search time of the approximate method (L2KNNGApprox) tends to be higher than that of the exact method (L2KNNG) for high recall levels (above 80%).
- When aiming to achieve high efficiency, approximate methods provide "good enough" performance for recall levels between 70% and 80%.
- When using approximate methods for larger data sets, it is important to tune parameters to achieve a recall level of at least 70%. Failure to do so may result in significantly worse recommendation performance.
- Not all the data sets achieve near-perfect recall using approximate methods. For some datasets, it is impractical to tune parameters of approximate methods to reach the desired recall as the execution cost may be higher than that of the exact method.

7 CONCLUSIONS AND FUTURE WORK

Our research was mainly focused on the nearest neighbor search, and the experiments we conducted were limited to the recommendation domain. In this thesis, we discussed how the quality of approximate nearest neighbor search affects the effectiveness and efficiency of the $top-N$ recommendation task. Our experimental results across six real-world data sets demonstrated that the approximate method achieves a similar recommendation performance as the exact method for high-quality search results (recall values above 80%) and only slightly lower performance for medium quality search results (recall values between 70% and 80%). The approximate approach is not necessarily an effective solution for reasonably sized data sets, as approximate search methods may take longer to finish the search than exact methods when desiring high-quality results for these data sets.

We used L2KNNGApprox to construct the k-nearest neighbor graph for the approximate approach. While L2KNNGApprox has been shown to outperform other approximate search methods in general, it would be interesting to conduct a similar study with state-of-the-art locality-based hashing or permutation-based approximate search methods, which would help generalize the result.

In future work, we plan to expand the approximate search error propagation analysis we conducted in this thesis to other domains, such as classification, regression, or clustering. Our understanding of real-world data sets is inadequate. We plan to study what properties of a data set affect the quality of the approximate nearest neighbor search and how the error incurred in the search of these data sets propagates in the underlying analysis. We hope that this research opens up more questions on the usage of approximate methods while providing helpful guidance to data mining and machine learning practitioners and researchers when working with nearest neighbor search algorithms.

Literature Cited

[1] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pp. 331–340, INSTICC Press, 2009.

[2] B. Frederickson, "Annoy." https://github.com/spotify/annoy.

[3] J. L. Bentley, "Multidimensional binary search trees in database applications," *IEEE Trans. Softw. Eng.*, vol. 5, pp. 333–340, July 1979.

[4] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, (New York, NY, USA), pp. 97–104, ACM, 2006.

[5] Y. Park, H. Hwang, and S.-g. Lee, "A novel algorithm for scalable k-nearest neighbour graph construction," *Journal of Information Science*, vol. 42, 07 2015.

[6] D. C. Anastasiu and G. Karypis, "L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, (New York, NY, USA), pp. 791–800, ACM, 2015.

[7] E. Kushilevitz, R. Ostrovsky, and Y. Rabani, "Efficient search for approximate nearest neighbor in high dimensional spaces," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, (New York, NY, USA), pp. 614–623, ACM, 1998.

[8] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, (New York, NY, USA), pp. 604–613, ACM, 1998.

[9] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 518–529, Morgan Kaufmann Publishers Inc., 1999.

[10] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *CoRR*, vol. abs/1807.05614, 2018.

[11] A. S. Das, M. Datar, A. Garg, and S. Rajaram, “Google news personalization: Scalable online collaborative filtering,” in *Proceedings of the 16th International Conference on World Wide Web*, WWW ’07, (New York, NY, USA), pp. 271–280, ACM, 2007.

[12] W. Dong, C. Moses, and K. Li, “Efficient k-nearest neighbor graph construction for generic similarity measures,” in *Proceedings of the 20th International Conference on World Wide Web*, WWW ’11, (New York, NY, USA), pp. 577–586, ACM, 2011.

[13] W. Dong, “Kgraph: A library for approximate nearest neighbor search.” https://github.com/aaalgo/kgraph.

[14] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, “Fast approximate nearest-neighbor search with k-nearest neighbor graph,” in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI’11, pp. 1312–1317, AAAI Press, 2011.

[15] E. Chavez Gonzalez, K. Figueroa, and G. Navarro, “Effective proximity retrieval by ordering permutations,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, pp. 1647–1658, Sept. 2008.

[16] G. Amato and P. Savino, “Approximate similarity search in metric spaces using inverted files,” in *Proceedings of the 3rd International Conference on Scalable Information Systems*, InfoScale ’08, (ICST, Brussels, Belgium, Belgium), pp. 28:1–28:10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[17] B. Naidan, L. Boytsov, and E. Nyberg, “Permutation search methods are efficient, yet faster search is possible,” *Proc. VLDB Endow.*, vol. 8, pp. 1618–1629, Aug. 2015.

[18] A. Ponomarenko, N. Avrelin, B. Naidan, and L. Boytsov, “Comparative analysis of data structures for approximate nearest neighbor search,” in *Proceedings of the Sixth International Conference on Emerging Network Intelligence*, p. 6, 12 2013.

[19] E. S. Tellez, E. Chávez, and G. Navarro, “Succinct nearest neighbor search,” in *Proceedings of the Fourth International Conference on SImilarity Search and APplications*, SISAP ’11, (New York, NY, USA), pp. 33–40, ACM, 2011.

[20] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, “Practical and optimal lsh for angular distance,” in *Proceedings of the 28th International*

Conference on Neural Information Processing Systems - Volume 1, NIPS'15, (Cambridge, MA, USA), pp. 1225–1233, MIT Press, 2015.

[21] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, 2014.

[22] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, (San Francisco, CA, USA), pp. 43–52, Morgan Kaufmann Publishers Inc., 1998.

[23] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, (New York, NY, USA), pp. 285–295, ACM, 2001.

[24] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. T. Riedl, "Application of dimensionality reduction in recommender system – a case study," in *IN ACM WEBKDD WORKSHOP*, 2000.

[25] A. Paterek, "Improving regularized singular value decomposition for collaborative filtering," *Proceedings of KDD Cup and Workshop*, 01 2007.

[26] M. Weimer, A. Karatzoglou, Q. V. Le, and A. Smola, "Cofirank maximum margin matrix factorization for collaborative ranking," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, (USA), pp. 1593–1600, Curran Associates Inc., 2007.

[27] N. N. Liu and Q. Yang, "Eigenrank: A ranking-oriented approach to collaborative filtering," in *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, (New York, NY, USA), pp. 83–90, ACM, 2008.

[28] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *2008 Eighth IEEE International Conference on Data Mining*, pp. 263–272, Dec 2008.

[29] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "Bpr: Bayesian personalized ranking from implicit feedback," in *Proceedings of the Twenty-Fifth*

Conference on Uncertainty in Artificial Intelligence, UAI '09, (Arlington, Virginia, United States), pp. 452–461, AUAI Press, 2009.

[30] Z. Gantner, L. Drumond, C. Freudenthaler, and L. Schmidt-Thieme, "Personalized ranking for non-uniformly sampled items," in *Proceedings of the 2011 International Conference on KDD Cup 2011 - Volume 18*, KDDCUP'11, pp. 231–247, JMLR.org, 2011.

[31] Y. Shi, M. Larson, and A. Hanjalic, "List-wise learning to rank with matrix factorization for collaborative filtering," in *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, (New York, NY, USA), pp. 269–272, ACM, 2010.

[32] Y. Shi, M. Larson, and A. Hanjalic, "Unifying rating-oriented and ranking-oriented collaborative filtering for improved recommendation," *Inf. Sci.*, vol. 229, pp. 29–39, Apr. 2013.

[33] J. Liu, C. wu, Y. Xiong, and W. Liu, "List-wise probabilistic matrix factorization for recommendation," *Information Sciences*, vol. 278, 04 2014.

[34] J. Weston, S. Bengio, and N. Usunier, "Wsabie: Scaling up to large vocabulary image annotation," in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pp. 2764–2770, AAAI Press, 2011.

[35] J. Weston, H. Yee, and R. J. Weiss, "Learning to rank recommendations with the k-order statistic loss," in *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, (New York, NY, USA), pp. 245–248, ACM, 2013.

[36] X. Ning and G. Karypis, "Slim: Sparse linear methods for top-n recommender systems," in *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, ICDM '11, (Washington, DC, USA), pp. 497–506, IEEE Computer Society, 2011.

[37] Y. Cheng, L. Yin, and Y. Yu, "Lorslim: Low rank sparse linear methods for top-n recommendations," in *Proceedings of the 2014 IEEE International Conference on Data Mining*, ICDM '14, (Washington, DC, USA), pp. 90–99, IEEE Computer Society, 2014.

[38] J. Liu, C. Sui, D. Deng, J. Wang, B. Feng, W. Liu, and C. Wu, “Representing conditional preference by boosted regression trees for recommendation,” *Information Sciences*, vol. 327, 08 2015.

[39] Y. Liu, P. Zhao, A. Sun, and C. Miao, “A boosting algorithm for item recommendation with implicit feedback,” in *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI’15, pp. 1792–1798, AAAI Press, 2015.

[40] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro, “Practical construction of k-nearest neighbor graphs in metric spaces,” in *Proceedings of the 5th International Conference on Experimental Algorithms*, WEA’06, (Berlin, Heidelberg), pp. 85–97, Springer-Verlag, 2006.

[41] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, “Approximate nearest neighbor algorithm based on navigable small world graphs,” *Information Systems*, vol. 45, pp. 61–68, 01 2013.

[42] “Movielens.” https://grouplens.org/datasets/movielens/. (Accessed on 11/09/2018).

[43] “Uci machine learning repository: Netflix prize data set.” https://web.archive.org/web/20090925184737/http://archive.ics.uci.edu/ml/datasets/Netflix+Prize. (Accessed on 11/09/2018).

[44] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen, “Improving recommendation lists through topic diversification,” in *Proceedings of the 14th International Conference on World Wide Web*, WWW ’05, (New York, NY, USA), pp. 22–32, ACM, 2005.

[45] “Book-crossing dataset.” http://www2.informatik.uni-freiburg.de/~cziegler/BX/. (Accessed on 11/09/2018).

[46] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” *Computer*, vol. 42, pp. 30–37, Aug. 2009.

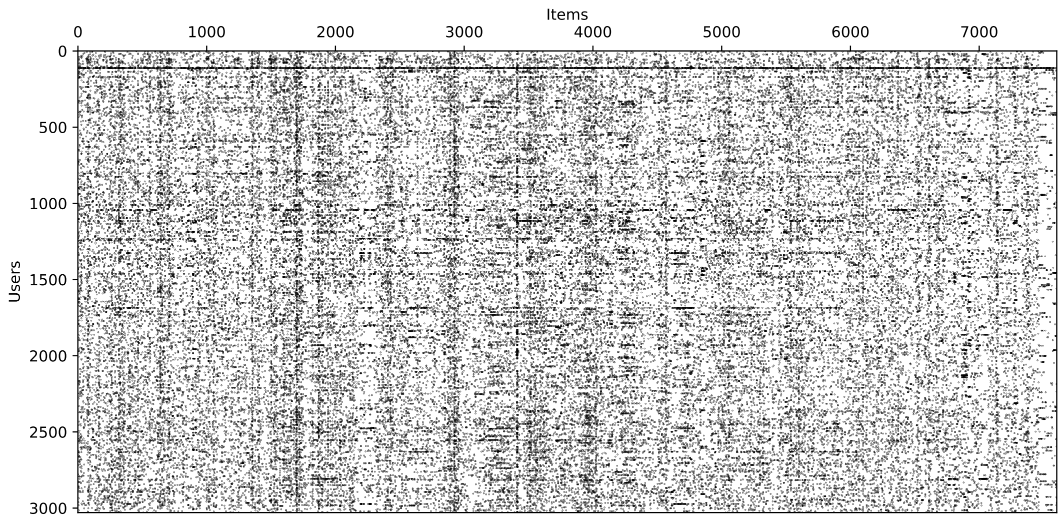
[47] B. Frederickson, “Fast python collaborative filtering for implicit datasets.” https://github.com/benfred/implicit, 2016.

[48] M. Kula, “Metadata embeddings for user and item cold-start recommendations,” *CoRR*, vol. abs/1507.08439, 2015.

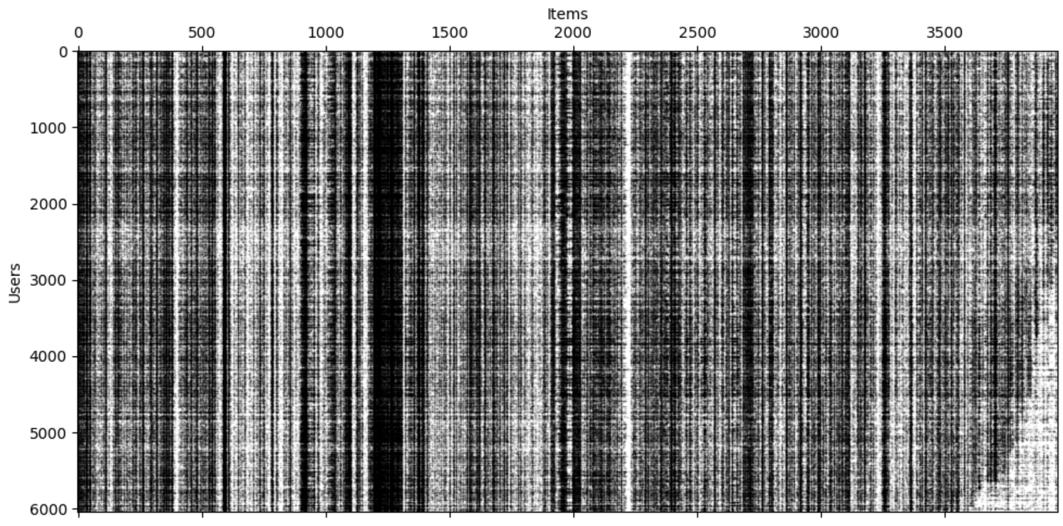
Appendix A

DATA SET CHARACTERISTICS

Fig. 9 and 10 show the distribution of non-zero elements over the matrix for the BX, ML1M, ML20M, and Netflix data sets (dark dots represent non-zero elements). Additional figures for the remaining data sets are included in Section 5.2.

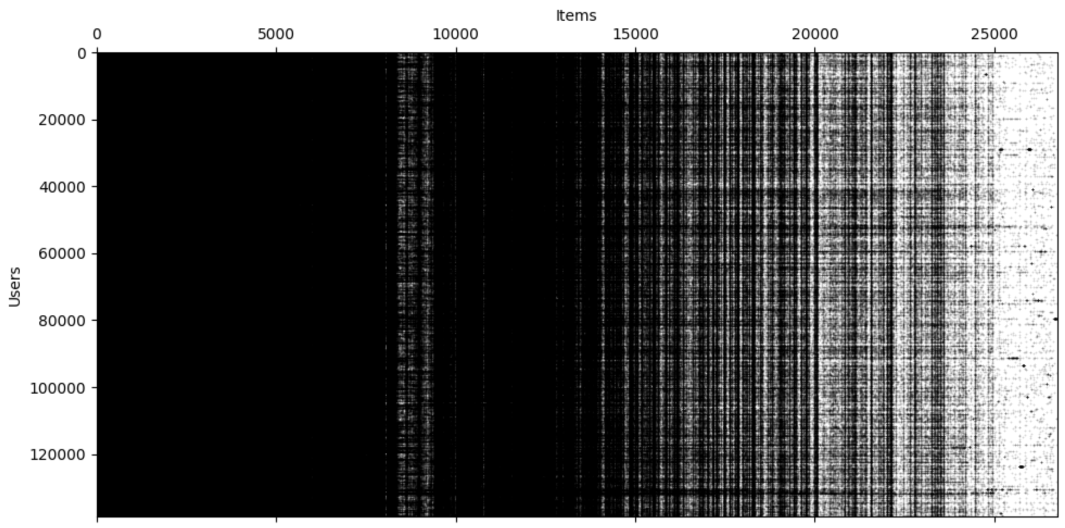


(a) BX

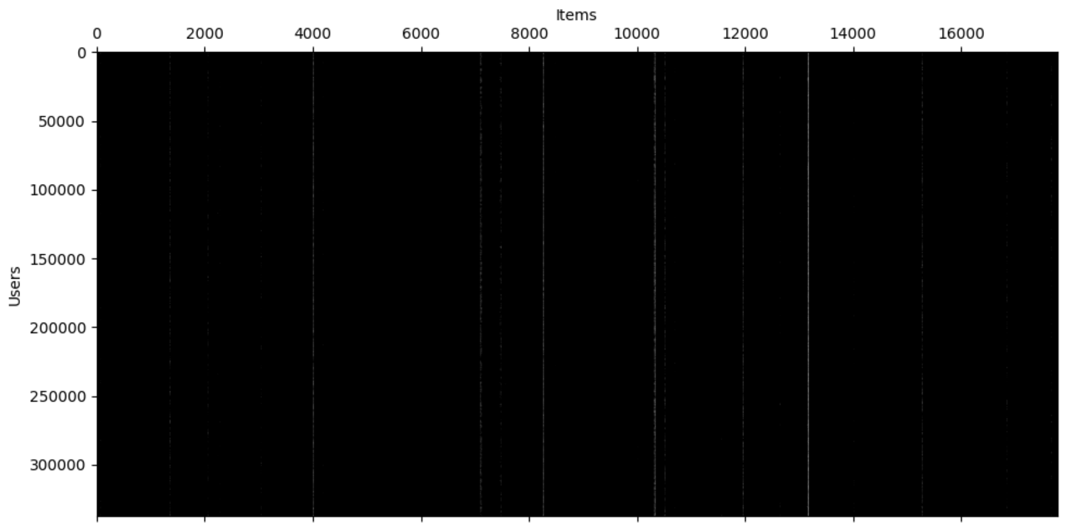


(b) ML1M

Fig. 9: Distribution of non-zero elements for BX and ML1M.



(a) ML20M



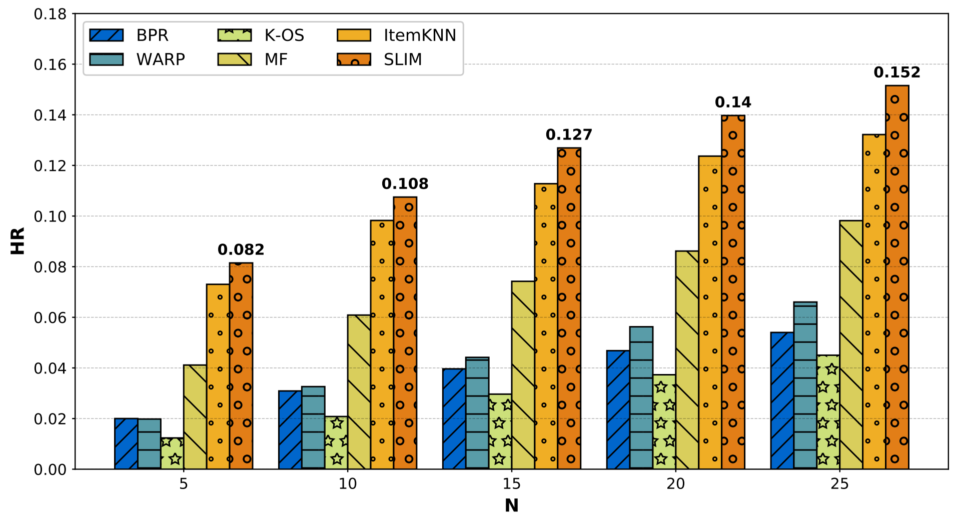
(b) Netflix

Fig. 10: Distribution of non-zero elements for ML20M and Netflix.

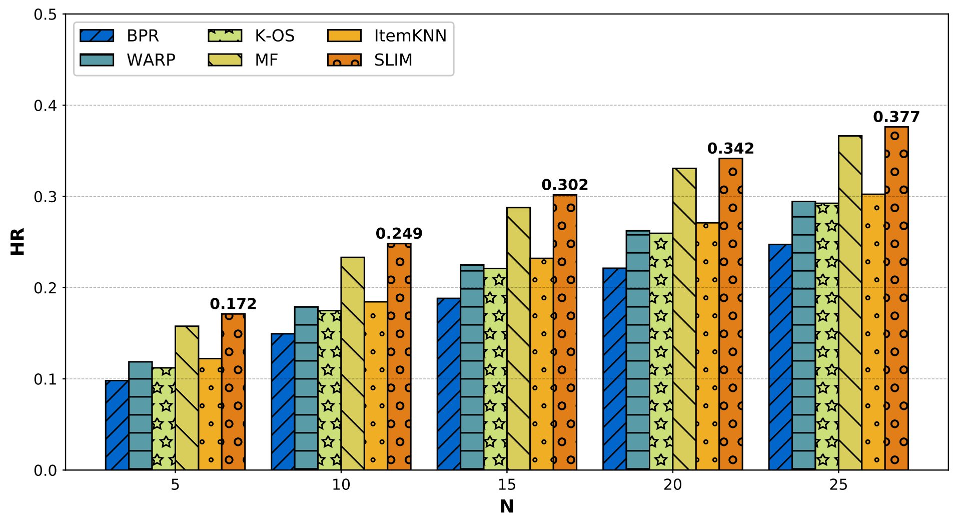
Appendix B

RECOMMENDATION FOR DIFFERENT TOP-N VALUES

Fig. 11 and 12 show the performance of different $top-N$ recommendation algorithms on the BX, ML1M, ML20M, and Netflix data sets. Additional figures for the remaining data sets are included in Section 6.1.

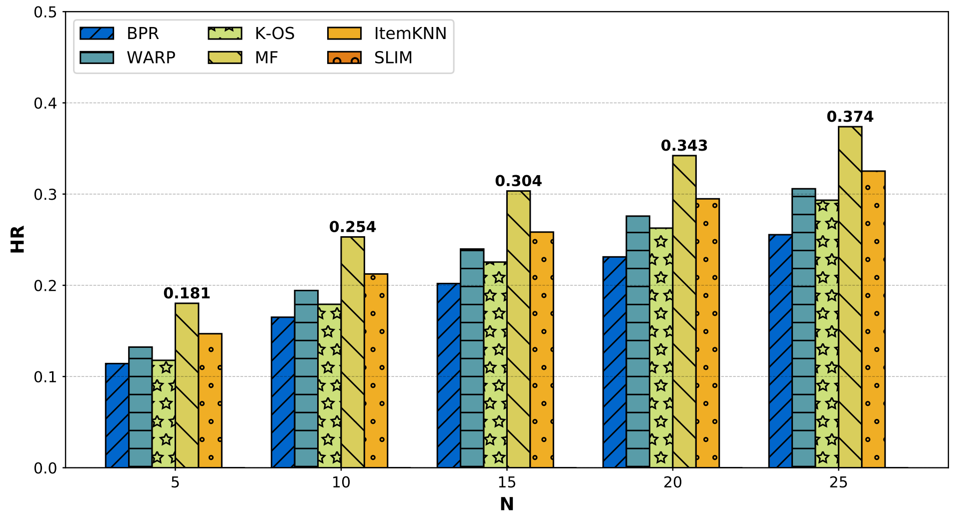


(a) BX

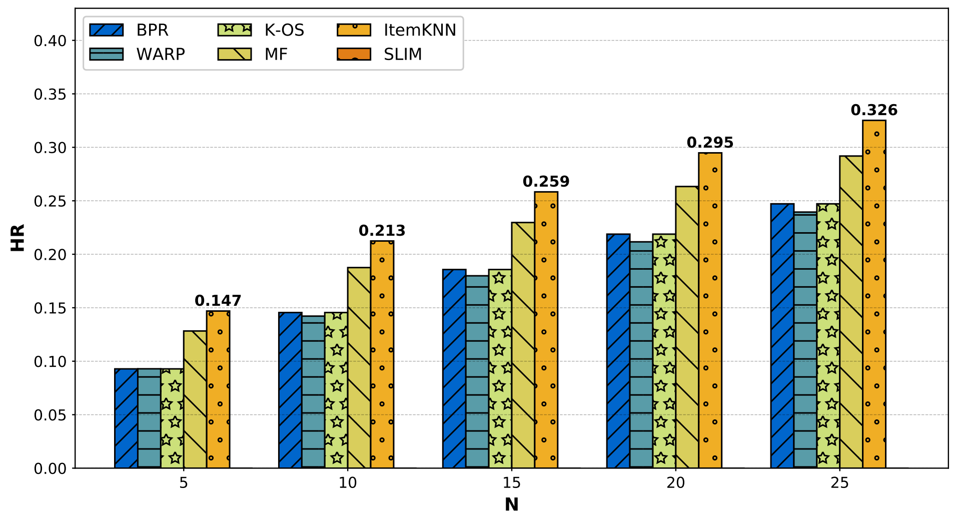


(b) ML1M

Fig. 11: Performance of recommendation algorithms for different N values for BX and ML1M.



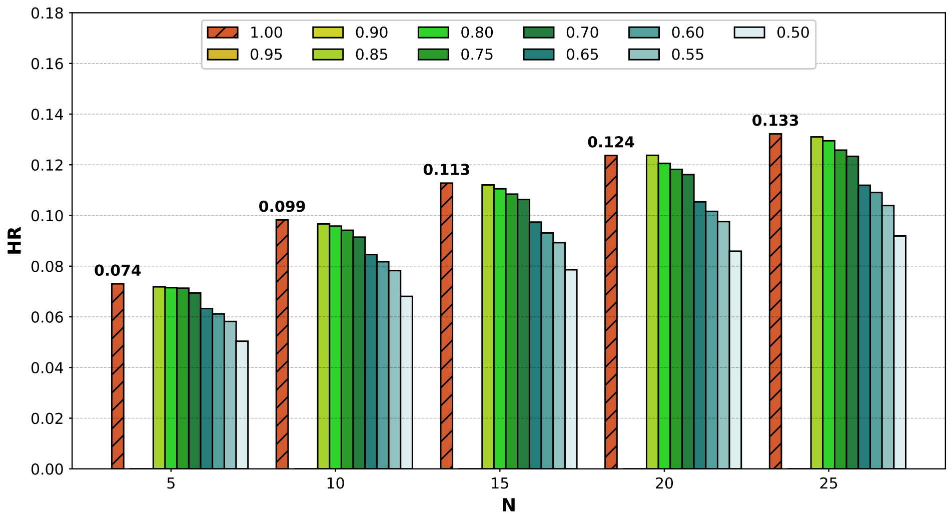
(a) ML20M



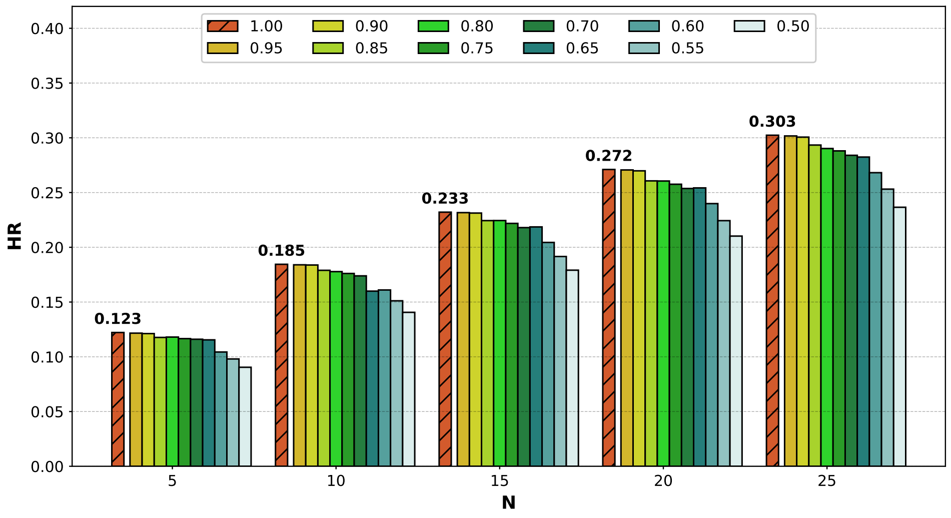
(b) Netflix

Fig. 12: Performance of recommendation algorithms for different N values for ML20M and Netflix.

Fig. 13 and 14 show the best approximate HR results compared with the exact HR for different values of N $\in \{5, 10, 15, 20, 25\}$ for the BX, ML1M, ML20M, and Netflix data sets. Additional figures for the remaining data sets are included in Section 6.2.2.

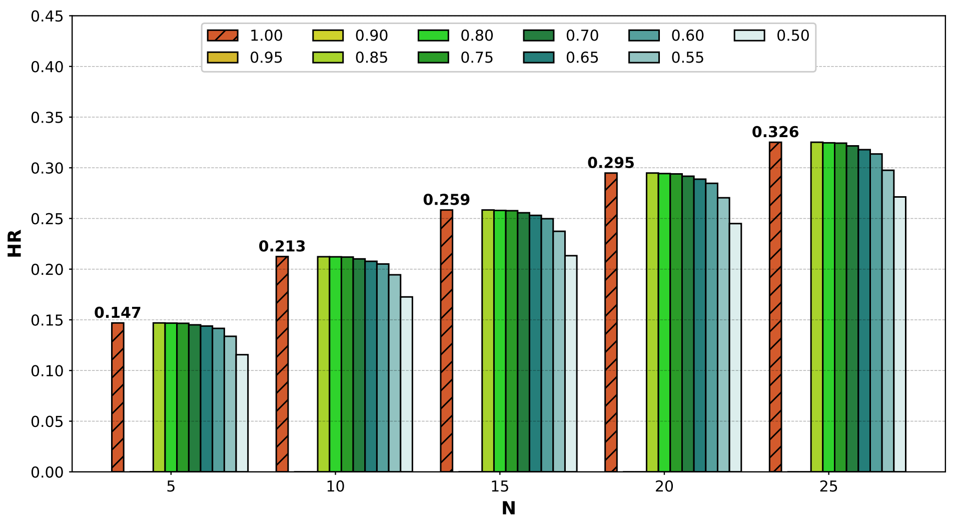


(a) BX

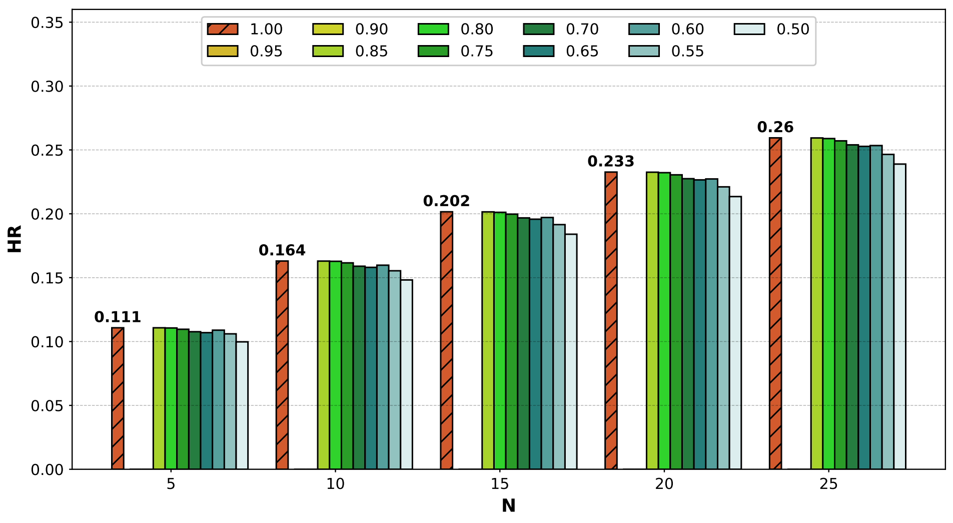


(b) ML1M

Fig. 13: ItemKNN Recommendation for different N values for BX and ML1M.



(a) ML20M



(b) Netflix

Fig. 14: ItemKNN Recommendation for different N values for ML20M and Netflix.

Appendix C

HYPER-PARAMETER CHOICES FOR TOP-N RECOMMENDATION ALGORITHMS

Table 4 represents the hyper-parameter choices for $top-N$ recommendation algorithms. Best and second-best HR values are shown in bold and italic, respectively. These hyper-parameters for each model were tuned using a grid search. We use the symbol '-' to denote that the corresponding algorithm failed to process the specific data set.

Table 4: Hyper-Parameter Choices for Top-N Recommendation Algorithms

Method	BX			ML100K			ML1M		
	Params		HR	Params		HR	Params		HR
ItemKNN	10	-	*0.099*	10	-	0.275	10	-	0.184
MF	10	-	0.061	10	-	*0.311*	20	-	*0.281*
BPR	0.01	10	0.031	0.01	10	0.237	0.05	10	0.149
WARP	0.01	10	0.033	0.01	10	0.286	0.05	10	0.179
K-OS	0.01	10	0.021	0.01	10	0.274	0.05	10	0.175
SLIM	3	0.5	**0.108**	2.0	2.0	**0.332**	1.0	2.0	**0.248**
Method	**ML10M**			**ML20M**			**Netflix**		
	Params		**HR**	**Params**		**HR**	**Params**		**HR**
ItemKNN	20	-	*0.239*	20	-	*0.259*	20	-	**0.213**
MF	50	-	**0.281**	100	-	**0.304**	100	-	*0.188*
BPR	0.01	20	0.2	0.5	30	0.165	1.0	30	0.165
WARP	0.05	20	0.235	0.5	30	0.198	1.0	30	0.185
K-OS	0.01	30	0.229	0.5	30	0.181	1.0	30	0.172
SLIM	-	-	-	-	-	-	-	-	-

The column "Params" in Table 4 represents the parameters for the corresponding method. For the ItemKNN method, the only required parameter is the number of neighbors. The MF method requires the number of latent factors. For the BPR, WARP, and K-OS methods, the parameters are the learning rate and the number of epochs to run. For SLIM, the parameters are $L2$-norm regularization parameter β and the $L1$-norm regularization parameter λ. The number of recommendations, N for results in this table is 10. For ItemKNN, k was selected from $\{5,10,20,25,50,75,100,200,500,1000\}$. For MF, the latent factor was selected from $\{10,20,50,100\}$. For SLIM, $L2$ regularization parameters were selected from $\{1,2,3,5\}$ and $L1$ regularization parameters were selected from $\{0.5,1,2\}$. For the BPR, WARP, and K-OS methods, the learning rate and number of epochs were selected from the range $\{0.01,0.05,0.5,1\}$ and $\{10,20,30\}$, respectively.

Appendix D

COMPUTATION TIME FOR TOP-N RECOMMENDATION ALGORITHMS

Table 5 shows the average recommendation computation time across the 5 folds of the cross validation. The number of recommendations, N, for results in this table is 10. We use the symbol '-' to denote that the corresponding algorithm failed to process the specific data set.

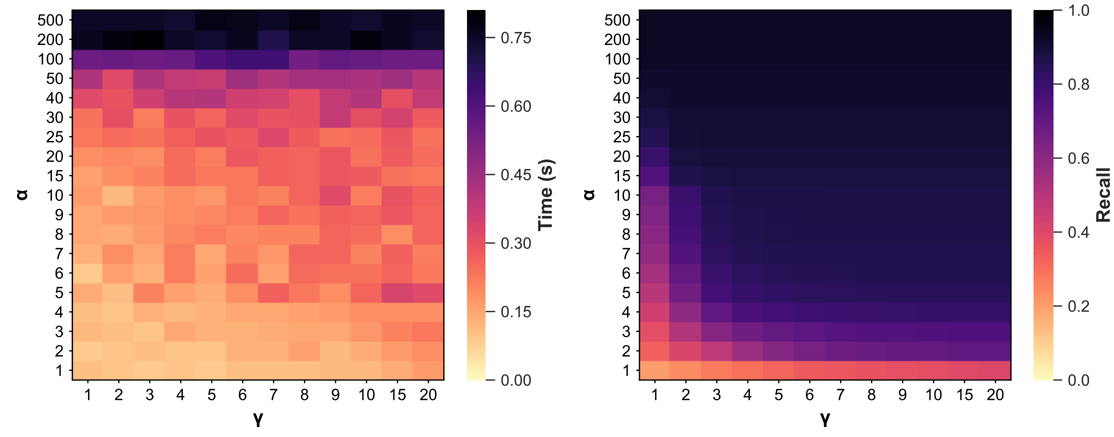
Table 5: Computation Efficiency Results

Dataset	**BPR**	**WARP**	**K-OS**	**MF**	**ItemKNN**	**SLIM**
BX	46	37	41	129	19	355
ML100K	54	59	48	368	41	956
ML1M	1445	1312	1248	545	253	8294
ML10M	5863	5794	5712	1736	329	-
ML20M	13242	14891	13145	4221	574	-
Netflix	38531	38604	39452	9873	872	-

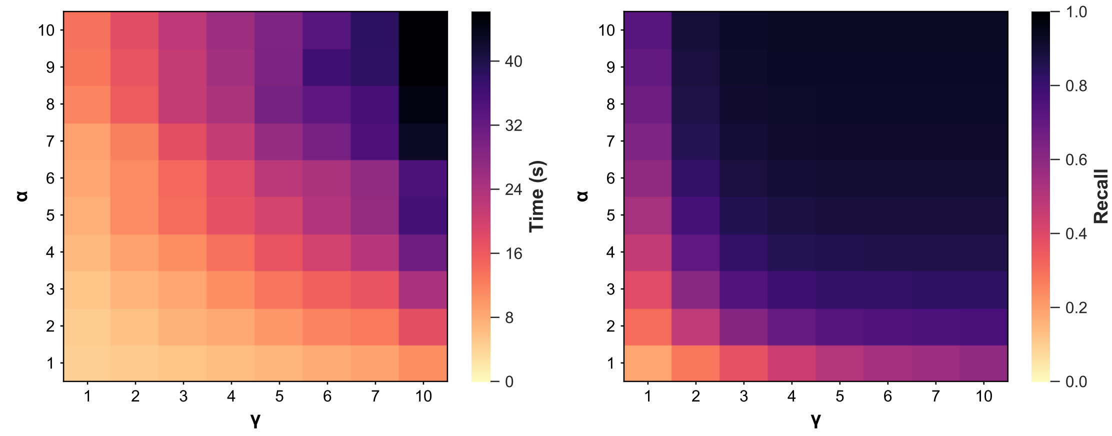
Appendix E

PARAMETER ANALYSIS OF APPROXIMATE METHODS

Fig. 15 and 16 show the effects of the α and γ parameters of the L2KNNGApprox method in terms of similarity computation time and recall for the ML100K, ML10M, ML20M, and Netflix data sets. Additional figures for the remaining data sets are included in Section 6.2.1.

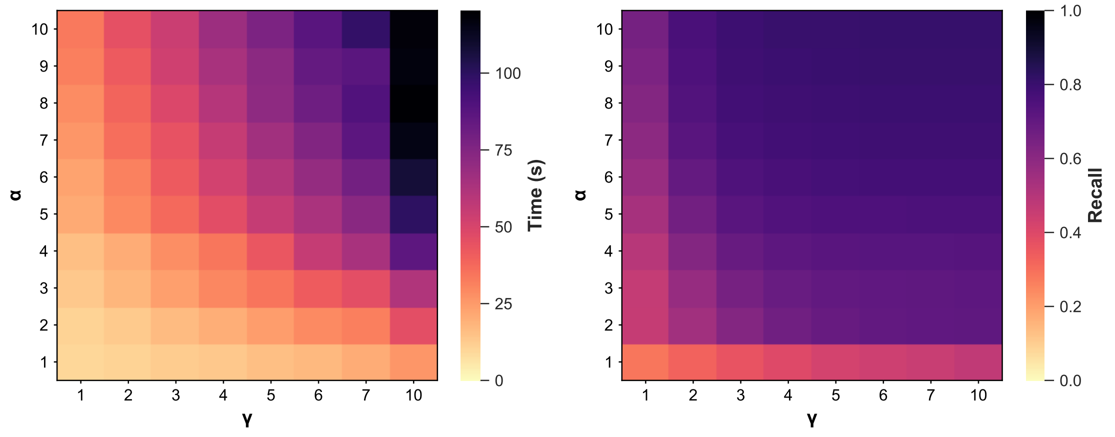


(a) ML100K

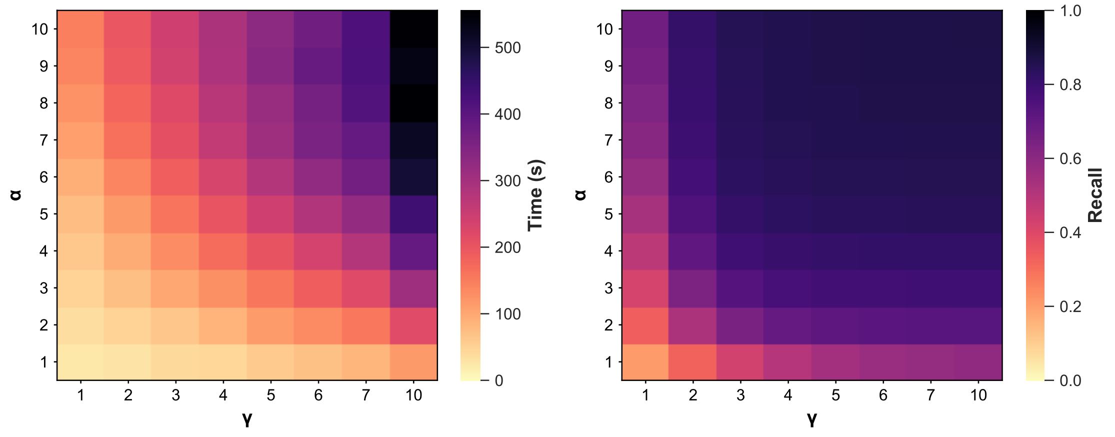


(b) ML10M

Fig. 15: Effects of α and γ on efficiency (left) and effectiveness (right) for ML100K and ML10M.



(a) ML20M



(b) Netflix

Fig. 16: Effects of α and γ on efficiency (left) and effectiveness (right) for ML20M and Netflix.

Table 6 shows the parameter choices for the ItemKNN approximate methods. The symbol '-' denotes that the corresponding data set was unable to achieve the desired recall in the L2KNNGApprox method.

Table 6: Parameter Choices for the ItemKNN Approximate Methods

Recall	BX		ML100K		ML1M		ML10M		ML20M		Netflix	
	α	γ	α	γ	α	γ	α	γ	α	γ	α	γ
0.95	-	-	-	-	9	4	100	1	-	-	-	-
0.90	-	-	25	4	4	4	6	7	-	-	-	-
0.85	40	2	9	3	10	1	4	4	200	5	7	6
0.80	20	2	4	8	8	1	3	4	10	4	9	2
0.75	10	5	3	8	2	3	2	6	6	3	5	2
0.70	7	5	4	3	1	10	2	4	3	5	2	4
0.65	5	4	10	1	5	1	8	1	3	3	10	1
0.60	4	6	8	1	2	2	3	2	7	1	7	1
0.55	4	2	2	4	1	4	1	7	2	2	1	6
0.50	3	2	3	2	3	1	1	5	4	1	1	4