공학박사학위논문

Fast Approximate Algorithms for k-NN Search and k-NN Graph Construction

k-NN 검색 및 k-NN 그래프 생성을 위한 고속 근사 알고리즘

2015 년 2 월

서울대학교 대학원

전기컴퓨터공학부

박 영 기

공학박사학위논문

Fast Approximate Algorithms for k-NN Search and k-NN Graph Construction

k-NN 검색 및 k-NN 그래프 생성을 위한 고속 근사 알고리즘

2015 년 2 월

서울대학교 대학원

전기컴퓨터공학부

박 영 기

Fast Approximate Algorithms for k-NN Search and k-NN Graph Construction

k-NN 검색 및 k-NN 그래프 생성을 위한 고속 근사 알고리즘

지도교수 이 상 구

이 논문을 공학박사학위논문으로 제출함

2015 년 1 월

서울대학교 대학원

전기컴퓨터공학부

박 영 기

박영기의 박사학위논문을 인준함

2014 년 12 월

위 원 장 : 김 형 주 (인)

부위원장 : 이 상 구 (인)

위 원 : 문 봉 기 (인)

위 원 : 심 규 석 (인)

위 원 : 황 혜 수 (인)

Abstract

Fast Approximate Algorithms for k-NN Search and k-NN Graph Construction

Youngki Park
School of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

Finding k-nearest neighbors (k-NN) is an essential part of recommeder systems, information retrieval, and many data mining and machine learning algorithms. However, there are two main problems in finding k-nearest neighbors: 1) Existing approaches require a huge amount of time when the number of objects or dimensions is scale up. 2) The k-NN computation methods do not show the consistent performance over different search tasks and types of data. In this dissertation, we present fast and versatile algorithms for finding k-nearest neighbors in order to cope with these problems. The main contributions are summarized as follows: first, we present an efficient and scalable algorithm for finding an approximate k-NN graph by filtering node pairs whose large value dimensions do not match at all. Second, a fast collaborative filtering algorithm that utilizes k-NN graph is presented. The main idea of this approach is to reverse the process of finding k-nearest neighbors in item-based collaborative filtering. Last, we propose a fast approximate algorithm for k-NN search by selecting query-specific signatures from a signature pool to pick high-quality k-NN candidates. The experimental results show that the proposed algorithms

guarantee a high level of accuracy while also being much faster than the other algorithms over different types of search tasks and datasets.

Keywords: k-nearest neighbor search, k-nearest neighbor graph construction, collaborative filtering, locality-sensitive hashing

Student Number: 2010-30219

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Finding k-nearest neighbors (k-NN) is one of the most important base operations in the field of recommender systems, information retrieval, data mining and machine learning. With the exponentially increasing amount of data, approximate k-NN computations rather than exact computations have become the predominant methods due to their lower computational cost. The aim is to reduce the search space as much as possible while retaining an acceptable level of accuracy. There are two main k-NN computation tasks: one is the *k-NN search*, which finds the (approximate) k-nearest neighbors for a given query; the other is k-NN graph construction, which finds the (approximate) k-nearest neighbors of all of the objects. In this dessertation, we propose a set of fast approximate algorithms for k-NN search and k-NN graph construction.

In this chapter, we begin by introducing our motivation and research problems in Section 1.1. In Section 1.2, we give an overview of our solutions to efficiently find approximate k-nearest neighbors. In Section 1.3, we highlight the contributions of the dissertation. We give an outline of the remaining chapters in Section 1.4.

1.1 Motivation and Challenges

Finding k-nearest neighbors is an essential part of recommender systems: for example, user-based [1] and item-based collaborative filtering [2] are two of the most widely used techniques in recommender systems and the k-NN computations play a central role in their methods. Assuming that we want to recommend items to a user, we recommend the preferred items of his k-nearest users in user-based collaborative filtering. In item-based collaborative filtering, we recommend the k-nearest items of his preferred items. It is known that the Amazon and YouTube recommender systems [3, 4] utilize CF-based algorithms, and many modified versions of CF algorithms continue to be proposed for the purpose of building context-aware recommender systems [5, 6, 7].

The k-NN computations are also very widely used in information retrieval: for a given query, one powerful way of finding search results is to find its k-nearest neighbors. For example, if we enter a short sequence of keywords as a query, then a search engine can show the k documents closest the query [8]. Likewise, if we provide our own image as a query, a search engine can show the k images closest to the query image [9]. This process is what we call *search*. Note after obtaining search results, we can now browse the datasets in popular search engines. If we select the "similar" button of the document in Google, then we can see its k-nearest documents [10]. Likewise, if we select any images of the search results in Google Images, then we can see its k-nearest images in the database. This process is what we call *similarity browsing*.

Many data mining and machine learning algorithms also exploit k-nearest neighbors. For example: 1) *k-NN classifier* determines the category of a query object based on the categories of its k-NN. 2) It is known that we can simulate *agglomerative clustering* using all of the k-NN relationships between objects in the database. The main idea of this approach is that instead of directly finding the closest pairs among the clusters, selecting the closest pairs among

the k-NN relationships is enough for clustering. 3) It is also known that we can detect the outliers based on k-NN [11]. The basic idea of this algorithm is that if there is an object o such that all of the other objects do not select o as a k-NN, then we select o as an outlier. 4) *Locally linear embedding* (LLE) uses all of the k-nearest neighbor relationships to reduce the dimensionality of data [12]. Unlike principal component analysis (PCA), LLE well preserves the k-NN relationships in reduced vectors.

Although there are many advantages and applications in finding k-nearest neighbors, there are two main problems related to k-NN computations: the first problem is that the exact k-NN computations require a hugh amount of time when the number of objects or dimensions is scale up. Thus approximate k-NN computations have become the predominant methods rather than exact computations recently. In Section 1.1.1, we will discuss this issue in detail. The second problem is that the existing approaches do not show the consistent performance over different search tasks and types of data. Thus for some cases, even the state-of-the-art approximate algorithms are not faster than the exact calculation method while keeping the acceptable level of accuracy. In Section 1.1.2, we will discuss this issue in more detail.

1.1.1 Fast Approximation

The first problem in finding k-nearest neighbors is that the exact k-NN computation method requires too much time. Let n be a number of objects and d be a number of dimensions. Assuming that we use the cosine similarity as our similarity measure, the exact k-NN computation for every object requires the time complexity of $O(n^2d)$. Thus as the number of objects increases, the computation time increases significantly. For example, Figure 1.1(a) shows the amount of time required for the exact k-NN computation of every object using the New York Times dataset. Because it takes more than 12 hours for 120,000 news articles in a single machine, it is impractical to use the exact calculation

method for larger datasets, such as Wikipedia pages or YouTube videos.

Our aim is to develop fast and scalable algorithms for k-NN computation while keeping the *acceptable* level of accuracy. In this dissertation, we set the acceptable level to the accuracy of 90% defined in Section 3.5.1, because it is known that at this level of accuracy the results of the collaborative filtering and agglomerative clustering algorithms are nearly the same as the original results [12, 13]: when we implement a recommender system based on the MovieLens dataset, the k-NN graph accuracy of more than 70% yields the similar recall to the 100% case with a difference of at most 0.1 on average; when we implement an agglomerative clustering algorithm based on the PIE face database, an 89% accurate graph is much similar to the 100% graph. Obviously, it could be varied depending on applications so that the k-NN computation method should also be able to adjust the accuracy using its parameters. Figure 1.1 shows that NN-Descent, which is one of the most efficient state-of-the-art algorithms, cannot achieve the 90% accuracy although the algorithm outperforms the brute-force approach significantly.

1.1.2 Versatility

The second problem in finding k-nearest neighbors is that the existing approaches do not show the consistent performance over different k-NN computation tasks and types of data. Thus we do not know in advance which algorithm is best for given application. Furthermore, it is observed that for some cases, none of the existing approaches is not faster than the exact computation method while keeping the acceptable level of accuracy. Examples of those cases are described in the following sections.

Typically, there are three types of k-NN computation tasks: k-NN computation for a single query (k-NN search), for every object in the database (k-NN graph construction), for some of the objects in the database (partial k-NN graph construction). An interesting thing is that although the k-NN graph construc-

tion can be implemented by the iterative executions of k-NN search, the k-NN search algorithms do not perform as well for k-NN graph construction, and vice versa. It is because they do not reuse the information that can be obtained from the k-NN computations of the other objects. Therefore, in cases where we need two or three types of k-NN computation tasks (e.g., a search engine that supports both search and similarity browsing), we have to find or develop an effective algorithm for each task.

The existing algorithms are also significantly affected by the types of datasets being used. For example, the existing approaches do not efficiently find the k-nearest neighbors for text or log data, because they are usually represented by very high-dimensional sparse vectors. Figure 1.2 shows that there is a significant difference in dimensionality between text/log datasets and multimedia datasets, more than an order of magnitude. As another example, even using the same raw multimedia data, the performance of existing approaches significantly varies depending on the types of feature extraction methods being used. Thus in cases where we need two or more types of feature extraction methods (e.g., a search engine that uses facial features for facial images and global features for the other images [14]), we have to find or develop an effective algorithm for each feature extraction method.

1.2 Our Solutions

In this section, we briefly introduce the fast approximate algorithms through our k-NN computation framework as shown in Figure 1.3. In this framework, there are 6 layers: *data*, *weighting scheme*, *similariy measure*, *preprocessing*, *algorithm* and *application*. For each layer, the components can be divided into two groups: one group is for sparse datasets (text and log data), and the other group is for dense datasets (multimedia data). For implementing a speicifc application, we can scan from the bottom to the top of this figure while selecting the components we want to use in the application. For example, in order to construct a real-

time recommender system, we can select log data as our data, TF-IDF as our weighting scheme, cosine similarity as our similarity measure, L_2-normalization and sorting by value/dimension as our preprocessing step, greedy filtering as our k-NN computation method, and reversed CF as our recommendation algorithm. Note as shown in Figure 1.4, greedy filtering does not support k-NN search operation so that we have to use signature selection LSH instead when our application needs k-NN search.

Among the various components described in Figure 1.3, our contribution consists of three parts: greedy filtering (GF), signature selection LSH (S2LSH), and reversed CF (RCF). Each part is described in the following subsections.

1.2.1 Greedy Filtering

GF can be used for both the k-NN graph construction and partial k-NN graph construction. The algorithms takes either text data or log data. This assumes that we use TF-IDF as our weighing scheme, which is de facto standard in text information retrieval and is also suitable for recommender systems. When we weight vectors with TF-IDF, we can use many variants described in [15]. Although GF is originally developed for cosine similarity, this also supports at least five representative similarity measures: cosine similarity, pearson correlation coefficient, adjust cosine similarity, normalized Euclidean distance, and Euclidean distance. We can use the first three similarity measures, because they are based on dot products. We can also use the fourth similarity measure because the k-nearest neighbors based on normalized Euclidean distance are same as those based on cosine similarity by the following formula: for normalized vectors a and b, $|a - b|^2 = 2(1 - cos(a, b))$. Finally, GF supports the fifth similarity measure because the k-nearest neighbors of based on Eucliean distance are similar to those based on cosine similarity when the dimension is high [16]. As a pre-processing step, GF basically uses L_2-normalization and sorting by value and dimension in order to enhance its performance.

The limitations of greedy filtering is that it is specialized for high dimensional sparse datasets and k-NN graph construction. However, as shown in Figure 1.4, S2LSH can replace this algorithm when appropriate.

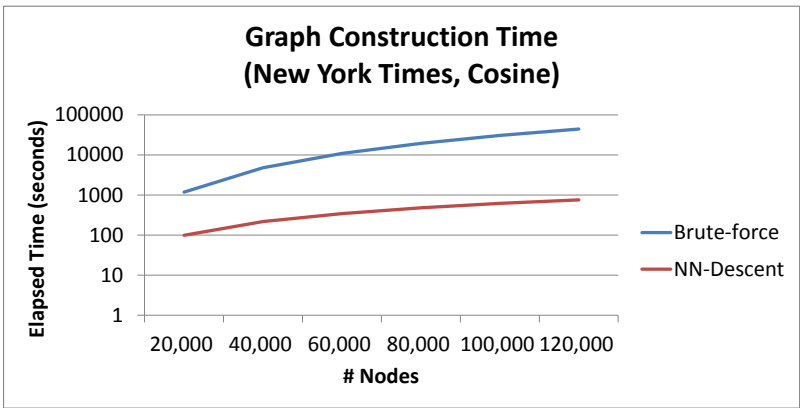
1.2.2 Signature Selection LSH

S2LSH is a generalized algorithm that can perform all of the three k-NN computation tasks. Although the algorithm basically takes multimedia data, it can also applied to text or log data if there is an efficient locality-sensitive hashing algorithm for those data. It can use various types of feature extraction methods, such as color features, texture features, and shape features. One main characteristics of S2LSH is that it can support even the non-metric similarity measures, such as Kullback-Leibler (KL) divergence, chamfer distance, dynamic time warping (DTW) distance, and edit distance, because there are locality-sensitive hashing algorithms for those similarity measures [17]. The locality sensitive hashing algorithm is one of the most important component for S2LSH, which affects the accuracy significantly. Thus we recommend to use the state-of-the-art LSH algorithms, such as anchor graph hashing (AGH) [18] or spherical hashing (SH) [9].

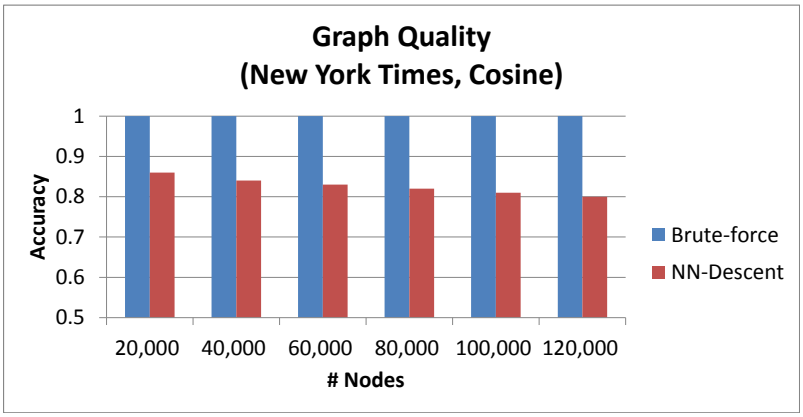
1.2.3 Reversed CF

RCF exploits log data to recommend items to users. The log data should be weighted by the TF-IDF weighting scheme, because it decreases the k-NN graph construction time significantly. Although all of the five similarity measures GF supports are widely used in recommender systems, RCF uses the cosine similarity measure in order to implement non-normalized cosine neighborhood. This algorithm uses L_2-normalization and sorting by value/dimension as a preprocessing step. RCF can select either GF and S2LSH, because it only requires k-NN graph construction. However, we recommend to use GF rather than S2LSH because the existing LSH algorithms for text/log datasets, such as random

hyperplanes [19] and MinHash [38, 39] are not good as those for multimedia datasets.



(a) Graph Construction Time



(b) Graph Quality

Figure 1.1: Experimental results of the existing approaches. Brute-force approach takes much time to construct a k-NN graph, and NN-Descent cannot construct a graph with a high level of quality for the New York Times dataset.

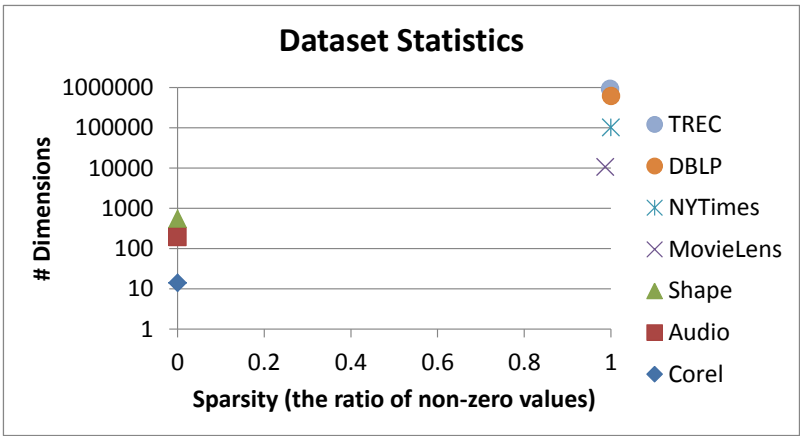


Figure 1.2: Statistics of representative datasets used in Chapter 3 and 5. Text/log datasets (TREC, DBLP, NYTimes and MovieLens) usually have very high dimensions, whereas multimedia datasets (Shape, Audio and Corel) usually have high dimensions.

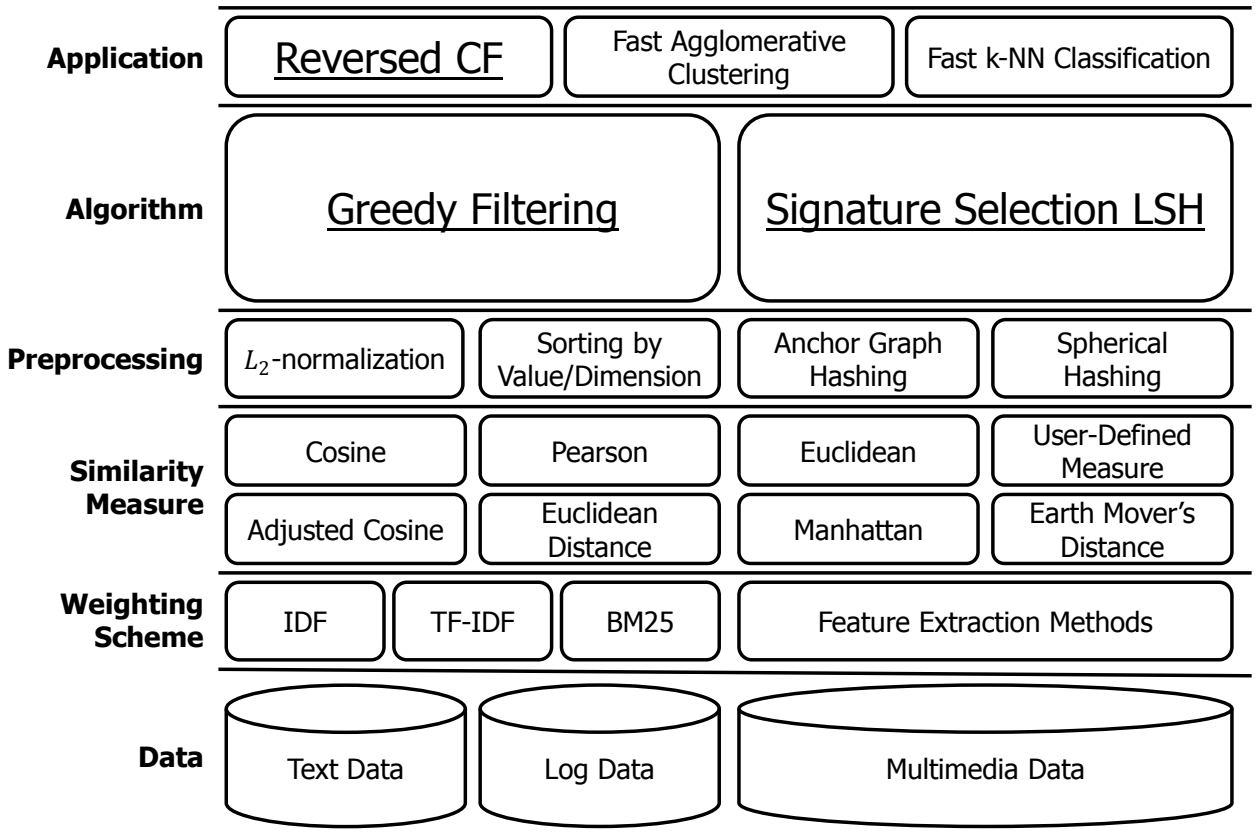


Figure 1.3: Our k-NN computation framework

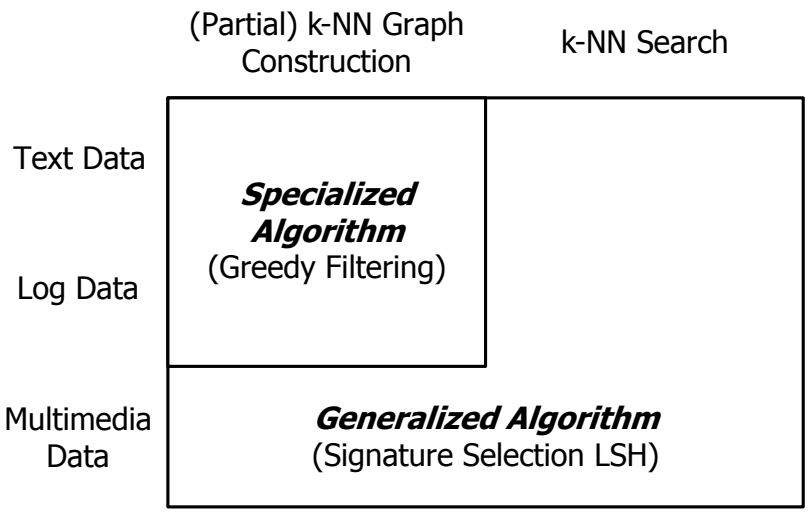


Figure 1.4: Our k-NN computation methods

1.3 Contributions

The main contributions of this dissertation can be summarized as follows:

- We present *greedy filtering (GF),* an efficient and scalable algorithm for finding an approximate k-nearest neighbor graph by filtering node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a linear time complexity, our algorithm chooses essentially a fixed number of node pairs as candidates for every node. We also present an optimized version of greedy filtering based on the use of inverted indices for the node prefixes.

- We propose *signature selection LSH (S2LSH),* a novel algorithm for approximate k-NN search. We select query-specific signatures from a signature pool to pick high-quality k-NN candidates. The signatures are generated based on a data-dependent LSH algorithm to capture the global topological features specific to the given dataset. We also incorporate three additional optimization techniques to further improve the performance of S2LSH in a bulk execution setting such as k-NN graph con-

struction.

- A fast collaborative filtering algorithm based on a k-NN graph is introduced. We call this algorithm as reversed CF (RCF), because the main idea of this approach is to reverse the process of finding k neighbors; instead of finding k similar neighbors of unrated items as in conventional collaborative filtering, RCF finds the k-nearest neighbors of rated items. Not only does this algorithm perform fewer predictions while filtering out inaccurate results, but it also supports the rapid retrieval of similar items.

1.4 Outline

The rest of this dissertation is structured as follows. Chapter 2 discusses background and related work of k-NN search and k-NN graph construction. Chapter 3 describes greedy filtering, a fast approximate k-NN graph construction algorithm. Chapter 4 describes signature selection LSH (S2LSH), a fast approximate k-NN search algorithm. Its extension for k-NN graph construction and partial k-NN graph construction is also presented. Chapter 5 describes reversed CF (RCF), a fast collborative filtering algorithm based on GF. Chapter 6 concludes the dissertation.

Chapter 2

Background and Related Work

This chapter describes the background knowledge by providing an overview of the existing k-NN search and k-NN graph construction algorithms. In Section 2.1, the k-NN search algorithms are introduced. In Section 2.2, the algorithms for k-NN graph construction are described while comparing with the algorithms in Section 2.1. Section 2.3 summarizes this chapter.

2.1 k-NN Search

As shown in Figure 2.1, *k-NN search* finds the k-nearest neighbors of query q. Formally, it is defined as follows:

Definition 1 (k-NN Search) Given a set of *data vectors* V, a parameter k and a query vector q, the similarity search returns $argmax^k_{v \in V}\left(sim(v, q)\right)$, where $argmax^k$ returns the k arguments that give the highest values.

One of the main barriers to achieving fast approximate k-NN search is that the data are usually represented by high-dimensional vectors: for example, 1) documents and logs are usually represented by a huge number of words or items,

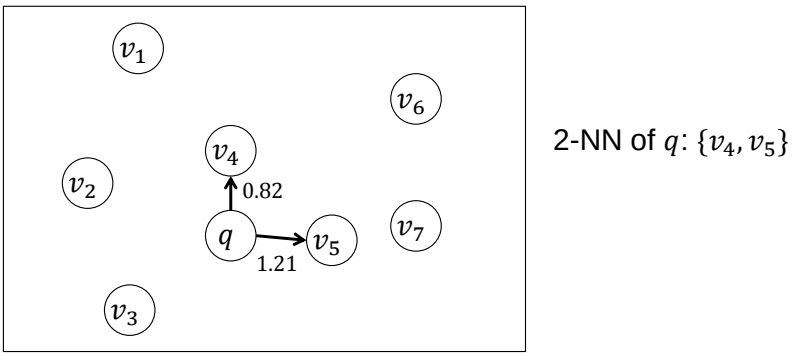


Figure 2.1: An example of k-NN search

respectively; 2) we usually represent images and videos by a huge number of extracted features. Although there have been proposed tree-like space partitioning approaches such as kd-tree, quadtree and R-tree in order to speed up this process, they all suffer from the curse of dimensionality [20].

Locality-sensitive hashing is one of the most effective techniques for finding k-nearest neighbors in a high-dimensional space [21]. It converts high-dimensional vectors into *signatures* while preserving the relative distances between them. Formally, a signature of a vector v consists of an ordered set of hash values, each calculated by the corresponding LSH hash function $h(v) : \mathbb{R}^d \to \mathbb{N}$. Because the signatures are usually low-dimensional vectors, we can find similar vectors with lower computational cost.

There are two types of challenges for LSH-based k-NN search: 1) improving the quality of LSH itself, and 2) finding the k-nearest neighbors using a given LSH algorithm.

2.1.1 Locality Sensitive Hashing

In order to improve the quality of LSH, there have been proposed various types of LSH functions. As shown in Figure 2.2(a), one of the most popular LSH functions is based on *random projections* [36, 21], which project a high-dimensional

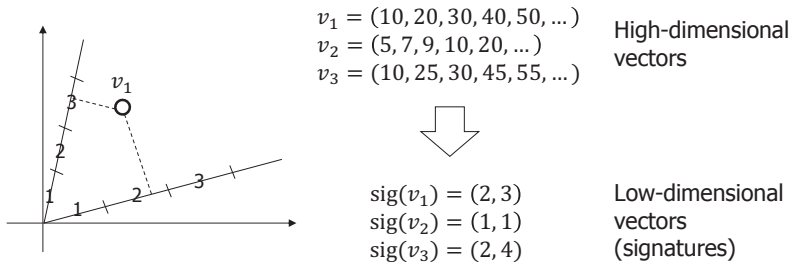
vector onto a small line segment as follows:

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{r} \right\rfloor \tag{2.1}$$

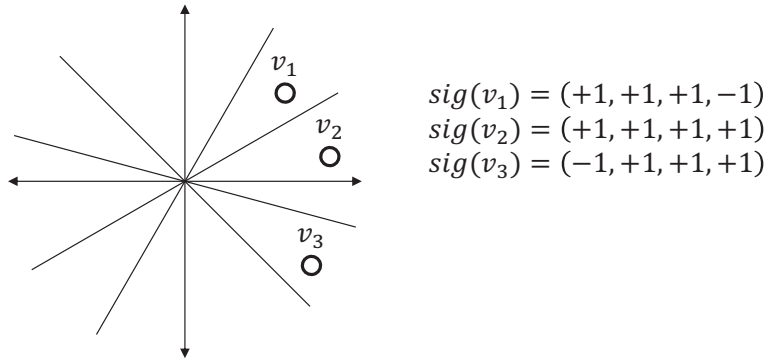
Here, v denotes a query vector (or a data vector); a is a vector where each component is drawn independently from a p-stable distribution; r is a constant; b is a constant chosen uniformly from the range $[0, r]$. By randomly selecting a and b the H number of times, we can obtain H hash functions h_1, h_2, ..., h_H and corresponding signatures of length H. There are also many popular variants of the random projections, such as *random hyperplanes* for cosine similarity [19] shown in Figure 2.2(b) or random-permutations based *MinHash* for jaccard similarity [20].

2.1.2 LSH-based k-NN Search

If we have a signature for every data vector v and a query q, then we need a way to find candidates for a given query q. One simplest way is to calculate the distances between the signatures of q and that of every data vector v, and select the k clostest data vectors as the k-nearest neighbors of q. However, this approach is not sufficient for many applications because of the following reasons: 1) if we have short signatures (for example, a length of 100), even the state-of-the-art data-dependent LSH algorithms do not achieve the level of MAP@100 higher than 15% for large amounts of datasets [9]; 2) if we have long signatures, it takes much time to calculate the similarities between the signatures. To cope with this problem, *Exact Euclidean LSH* (E2LSH) [21, 20] constructs compound hash functions $g_1(v)$, $g_2(v)$, ..., $g_L(v)$, each consisting of an equal number of hash functions. Then if a query vector q and a data vector v have a same compound hash value, i.e., $g_i(q) = g_i(v)$ for some i, we consider v as a candidate for the k-nearest neighbors of q. *LSB-tree* and *LSB-forest* [22] further convert every low-dimensional vector $g(v)$ into one-dimensional value using z-order curve. Given a query vector q, they select a vector that has a



(a) Random projections



(b) Random hyperplanes

Figure 2.2: Illustrative examples of two of the most popular locality-sensitive hashing schemes

z-order value with the greatest LLCP (length of longest common prefix) as a candidate for q. *Collision Counting LSH* (C2LSH) [23] counts the number of collisions between a query vector q and data vectors v using the hash functions h_1, h_2, ..., h_H, and if the collision counts is equal to or greater than pre-specified threshold l, then it selects v as a candidate for q.

2.2 k-NN Graph Construction

As shown in Figure 2.2, *k-NN graph construction* finds the k-nearest neighbors for every node. Formally, it is defined as follows:

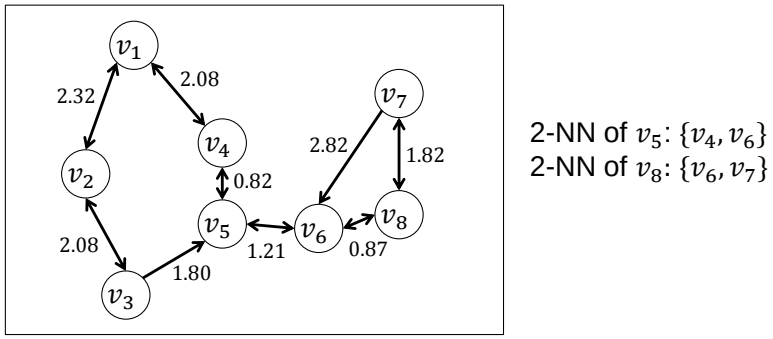


Figure 2.3: An example of k-NN graph construction

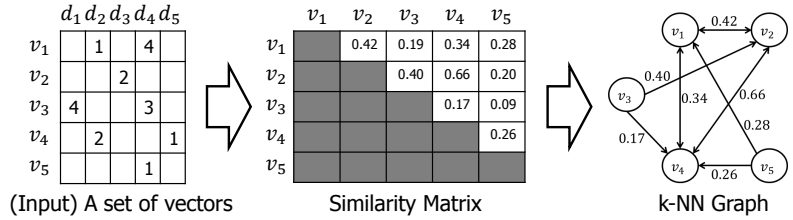


Figure 2.4: An example of brute-force approach for k-NN graph construction

Definition 2 (k-NN Graph Construction) Given a set of vectors V, the k-NN graph construction returns for every vector $v_i \in V$, $argmax^k_{v_j \in V \wedge v_i \neq v_j}(sim(v_i, v_j))$, where $argmax^k$ returns the k arguments that give the highest values.

k-NN graph construction is the process of finding (approximate) k-nearest neighbors among the vectors in V for every vector in V. It is also one of the primitives operations in data mining, information retrieval, recommender systems and machine learning [24, 12, 25, 4, 26, 11, 13, 27, 28, 29, 30]. Figure 2.3 shows the simplest approach for constructing a k-NN graph: first, given a set of vectors, we calculate all of the similarities between vectors and store them in a matrix (or maintain a set of k-NN lists for reducing the memory requirement of this process). Then we identify the k-nearest neighbors for every vector based on the matrix. Since this brute-force approach requires the significant amount of time, there have been proposed four types of algorithms for fast approximate

k-NN graph construction: 1) LSH-based Approach, 2) clustering or hyperplane-based algorithms, 3) heuristic-based algorithms, and 4) similarity join or top-k similarity join algorithms.

2.2.1 LSH-based Approach

One naive way to construct a k-NN graph is to execute a LSH-based k-NN search algorithm for every vector in V. However, the algorithms for k-NN search in the bulk execution setting is usually outperformed by the existing k-NN graph construction algorithms [24, 25] in that they do not reuse the information that can be obtained through the search task of the other vectors. In order to cope with this problem, the method of Zhang et al. [29] exploit the heuristic that the 2-hop neighbors of a vector v could be similar to v. It also applies the random projections to the compound hash functions $g_1(v)$, $g_2(v)$, ..., $g_L(v)$, which is one variation of LSB-tree and LSB-forest.

2.2.2 Clustering-based Approach

Clustering-based algorithms are the most simplest methods. The intuition behind these algorithms is that the vectors in the same clusters have a high probability that they are the k-nearest neighbors of each other. It is known that k-means clustering and canopy clustering based algorithms [31] are a little bit faster than brute-force search while keeping the high level of accuracy. Wang et al. [28] found that the iterative execution of 2-means clustering is more effective in finding k-NN graph. Recursive Lanczos bisection [12] uses a hyperplane in order to make clusters: first, it draws a hyperplane that splits the set of vectors V such that it maximizes the sum of squared distances between $v \in V$ to the hyperplane that passes through the centroid. Second, it (recursively) divide the vectors in V into two overlapping clusters using the hyperplane. Figure 2.5 shows an example of recursive Lanczos bisection. Similarly, the method of Wang et al. [27] uses random hyperplanes to divide the set of vectors V. Note that

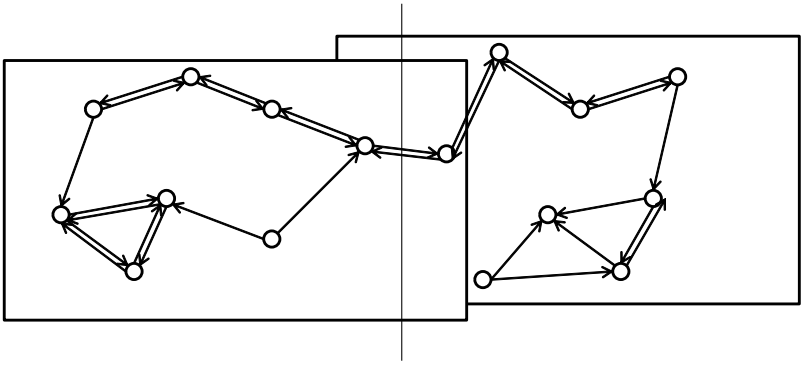


Figure 2.5: An illustrative example of recursive Lanczos bisection

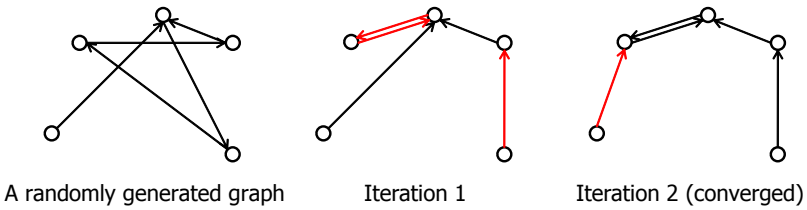


Figure 2.6: An illustrative example of NN-Descent

the clustering algorithms exploit the abovementioned 2-hop neighbors heuristic to improve the accuracy.

2.2.3 Heuristic-based Approach

NN-Descent [24] exploit the 2-hop neighbors heuristic but in a more efficient way. It first randomly selects the k-nearest neighbors for every vector. Then for each vector, it checks whether a neighbor of a neighbor of the vector is similar to the vector and repeats until convergence. Figure 2.6 shows an example of NN-Descent. It also uses a technique for avoiding many redundant distance computations without the use of much memory, and a technique for slowing down the convergence of the algorithm. The experimental results show that although NN-Descent outperforms the other algorithms in terms of accuracy and execution time, it does not perform well as the number of dimensions scales up [24].

2.2.4 Similarity Join

The k-NN graph construction is closely related to other fields, such as the *similarity join* and *top-k similarity join* fields. First, we introduce the similarity join problem as follows:

Definition 3 (Similarity Join) Given a set of vectors V and a similarity threshold ϵ, a similarity join algorithm returns all possible pairs $\langle v_i \in V, v_j \in V \rangle$ such that $sim(v_i, v_j) \geq \epsilon$.

Assume that we use cosine similarity as our similarity measure. The inverted index join algorithm [32] for similarity join builds inverted indices for all dimensions and then exploits them to calculate the similarities. While it performs much faster than the brute-force search algorithm for sparse datasets, it still has to calculate all of the similarities between the vectors. On the other hand, prefix filtering techniques [32, 33, 34] effectively reduce the search space. They sort the elements of all vectors by their dimensions and set the prefixes such that the similarity between two vectors is below a threshold when their prefixes do not have a common dimension. As a result, we can easily prune many vector pairs by only looking at their prefixes.

The top-k similarity join is identical to the similarity join with regards to finding the most similar pairs. The difference is that it is based on a parameter k instead of ϵ. The top-k similarity join is defined as follows:

Definition 4 (Top-k Similarity Join) Given a set of vectors V and a parameter k, a top-k similarity join algorithm returns $argmax^k_{\langle x \in V, y \in V \rangle \wedge x \neq y} \left(sim(x, y) \right)$, where $argmax^k$ returns the k arguments that give the highest values.

The most common strategy is to calculate the similarities of the most *probable* vector pairs first and then to iterate this step until a stop condition occurs. For example, Kim *et al.* [35] estimates a similarity value ϵ corresponding to

the parameter k, selects the most probable candidates, and continues to select candidates until it can be guaranteed that all vector pairs excluding those that were already selected as the candidates have similarity values of less than ϵ. Similarly, Xiao *et al.* [10] stops its iteration when it can be guaranteed that the similarity value of the next probable vector pair is not greater than that of any candidate that has been selected.

Note that the problem definitions in related work are analogous to that of k-NN graph construction such that the abovementioned solutions can also be applied to constructing k-NN graphs. For example, if we know all of the similarities between vectors by the inverted index join algorithm, we can obtain the k-NN graph by taking the most similar k vectors for each vector and throwing the rest away. However, these types of approaches do not perform well as the number of nodes or dimensions is scaled up. In Chapter 3, we discuss these issues in detail, present several ways to construct a k-NN graph based on the algorithms of these fields, and analyze their performance results.

2.3 Summary

The limitations of the existing k-NN search algorithms are twofold: first, they use data-independent LSH techniques (such as random projections) at the early stage of the algorithms so that we could lose too much information about relative distances according to the types of datasets being used. For example, the existing approaches do not achieve a high level of accuracy for the 500-dimensional NUS-WIDE dataset using a small amount of time. Second, they use data-independent candidate selection techniques. For example, they do not consider which hash function is the best for selecting candidates in certain types of datasets although some hash functions could play a more important role in achieving the high level of accuracy. Our approach alleviates the above problems based on data-dependent LSH functions and our data-dependent signature selection algorithms.

The limitations of the existing k-NN graph construction algorithms are as follows: first, the algorithms do not consider the data distributions in order to speed up the elapsed time. Second, the algorithms do not effectively support different types of k-NN computation tasks, such as *partial* k-NN graph construction, which is defined as finding k-nearest neighbors among the vectors in V for every vector $v \in V'$ such that $V' \subset V$. Note this task can be used for incremental k-NN graph construction.

Table 2.1 shows the summary of the all k-NN computation methods. Our approaches are highlighted in boldface and italic. Note that the brute-force approach is used as a baseline for dense datasets, and that the inverted index join-based algorithm is used as a baseline for sparse datasets. In practice, inverted index join-based algorithm performs more than 10 times faster than brute-force approach in sparse datasets. Figure 2.7 shows the comparison of every k-NN computation process: 1) the baseline approaches directly compute the k-nearest neighbors. 2) Most of the k-NN graph construction algorithms try to efficiently find the candidate pairs, and then construct a k-NN graph by calculating their similarities. 3) k-NN search algorithms reduce the dimensionality using locality-sensitive hashing, and find the candidate pairs based on the generated signatures. 4) Finally, some approaches further convert the reduced vectors into one-dimensional values and then find the candidate pairs. In the following chapters, we will compare and analyze their performance in detail.

Algorithms	**Tasks**	**Data Types**	**Similarity Measures**	**Main Methods**
Brute-force approach	Search / graph	Sparse / dense	All	Exhaustive search
Inverted index join	Graph	Sparse	Cosine similarity	Inverted index
k-means clustering	Graph	Sparse / dense	Euclidean distance	k-means clustering
Canopy clustering	Graph	Sparse / dense	All	Canopy clustering
Recursive Lanczos bisection	Graph	Dense	Euclidean distance	Spectral bisection
Zhang's approach	Graph	Sparse / dense	Popular	LSH, projection
NN-Descent	Graph	Sparse / dense	All	Heuristic-based
Greedy Filtering	***Graph***	***Sparse, TF-IDF***	***Cosine similarity***	***Data distribution-based***
E2LSH	Search	Sparse / dense	Euclidean distance	LSH
C2LSH	Search	Sparse / dense	Euclidean distance	LSH
LSB-tree	Search	Sparse / dense	Euclidean distance	LSH, z-order curve
S2LSH	***Search / graph***	***Sparse / dense***	***Popular***	***LSH***

Table 2.1: Summary of the all k-NN computation methods. Our algorithms are highlighted in underlined boldface.

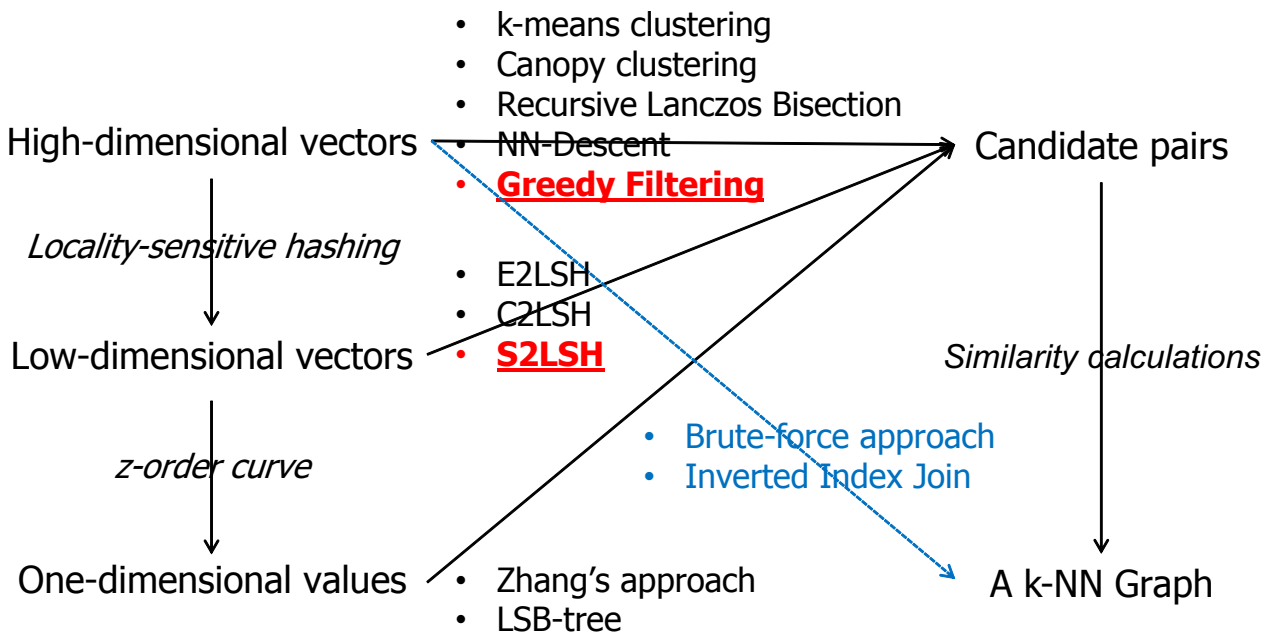


Figure 2.7: Comparison of every k-NN computation process. Our algorithms are highlighted in boldface and italic.

Chapter 3

Fast Approximate k-NN Graph Construction

Finding the k-nearest neighbors for every node is one of the most important data mining tasks as a primitive operation in the field of information retrieval and recommender systems. However, existing approaches to this problem do not perform as well when the number of nodes or dimensions is scaled up. In this chapter, we present *greedy filtering*, an efficient and scalable algorithm for finding an approximate k-nearest neighbor graph by filtering node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a time complexity of $O(n)$, our algorithm chooses essentially a fixed number of node pairs as candidates for every node. We also present a faster version of greedy filtering based on the use of inverted indices for the node prefixes. We conduct extensive experiments in which we (i) compare our approaches to the state-of-the-art algorithms in seven different types of datasets, and (ii) adopt other algorithms in related fields (similarity join, top-k similarity join and similarity search fields) to solve this problem and evaluate them. The experimental results show that greedy filtering guarantees a high level of accuracy while also being much faster than other algorithms for large

amounts of high-dimensional data.

3.1 Introduction

Constructing a k-Nearest Neighbor (k-NN) graph is an important data mining task which returns a list of the most similar k nodes for every node [30]. For example, assuming that we constructed a k-NN graph whose nodes represent users, we can quickly recommend items to user u by examining the purchase lists of u's nearest neighbors. Furthermore, if we implement an enterprise search system, we can easily provide an additional feature that finds k documents most similar to recently viewed documents.

We can calculate the similarities of all possible pairs of k-NN graph nodes by a brute-force search, for a total of $n(n-1)/2$. However, because there are many nodes and dimensions (features) in the general datasets, not only does calculating the similarity between a node pair require a relatively long execution time, but the total execution time will be very large. The inverted index join algorithm [32] is much faster than a brute-force search in sparse datasets. It is one of the fastest algorithms among those producing exact k-NN graphs, but it also requires $O(n^2)$ asymptotic time complexity and its actual execution time grows exponentially.

Another way to construct a k-NN graph is to execute a k-nearest neighbor algorithm such as locality sensitive hashing (LSH) iteratively. LSH algorithms [36, 37, 19, 38] first generate a certain number of signatures for every node. When a query node is given, the LSH compares its signatures to those of the other nodes. Because we have to execute the algorithm for every node, the graph construction time will be long unless one query can be executed in a short time.

As far as we know, NN-Descent [24] is the most efficient approach for constructing k-NN graphs. It randomly selects k-NN lists first before exploiting the heuristic in which a neighbor of a neighbor of a node is also be a neighbor of the node. This dramatically reduces the number of comparisons while retain-

ing a reasonably high level of accuracy. Although the performance is adequate as the number of nodes grows, it does not perform well when the number of dimensions is scaled up.

In this chapter, we present greedy filtering, an efficient, scalable algorithm for k-NN graph construction. This finds an approximate k-NN graph by filtering node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a time complexity of $O(n)$, our algorithm selects essentially a fixed number of node pairs as candidates for every node. We also present a faster version of greedy filtering based on the use of inverted indices for the prefixes of nodes. We demonstrate the effectiveness of these algorithms through extensive experiments where we compare various types of algorithms and datasets. More specifically, our contributions are as follows:

- We propose a novel algorithm to construct a k-NN graph. Unlike existing algorithms, the proposed algorithm performs well as the number of nodes or dimensions is scaled up. We also present a faster version of the algorithm based on inverted indices (Section 3.3).

- We present several ways to construct a k-NN graph based on the top-k similarity join, similarity join, and similarity search algorithms (Section 3.5.1). Additionally, we show their weaknesses by analyzing their experimental results (Section 3.5.2).

- We conduct extensive experiments in which we compare our approaches to existing algorithms in seven different types of datasets. The experimental results show that greedy filtering guarantees a high level of accuracy while also being much faster than the other algorithms for large amounts of high dimensional data. We also analyze the properties of the algorithms with the TF-IDF weighting scheme (Section 3.4 and 3.5.2).

3.2 Problem Formulation

In this section, we redefine the k-NN graph construction as follows: let G be a graph with n nodes and no edges, V be the set of nodes of the graph, and D be the set of dimensions of the nodes. Each node $v \in V$ is represented by a vector, which is an ordered set of elements $e_1, e_2, ..., e_{|v|-1}, e_{|v|}$ such that each has a pair consisting of a dimension and a value, $\langle d_i, r_j \rangle$, where $d \in D$ and $0 \leq r_j \in \mathbb{R} \leq 1$. The values are normalized by L_2-norm such that the following equation holds:

$$\sum_{\langle d_i \in D, r_j \in \mathbb{R} \rangle \in V} r_j^2 = 1. \tag{3.1}$$

Definition 1 (k-NN Graph Construction) Given a set of vectors V, the k-NN graph construction returns for each vector $x \in V$, $argmax^k_{y \in V \wedge x \neq y} (sim(x, y))$, where $argmax^k$ returns the k arguments that give the highest values.

We use the cosine similarity as the similarity measure for k-NN graph construction. The cosine similarity is defined as follows:

$$sim(v_i \in V, v_j \in V) = \frac{v_i \cdot v_j}{\|v_i\| \|v_j\|} = v_i \cdot v_j. \tag{3.2}$$

Example 1. In Figure 3.1, if we assume that the hidden elements (as described by the ellipses) have a value of 0 and $k = 2$, the k-nearest neighbors of v_1 are v_2 and v_4, because $sim(v_1, v_2)$, $sim(v_1, v_3)$, $sim(v_1, v_4)$, and $sim(v_1, v_5)$ are 0.42, 0.13, 0.34, 0.28, respectively. The k-NN graph is obtained by finding k-nearest neighbors for every vector: $\{v_2, v_4\}, \{v_4, v_1\}, \{v_2, v_4\}, \{v_2, v_1\}, \{v_1, v_4\}$.

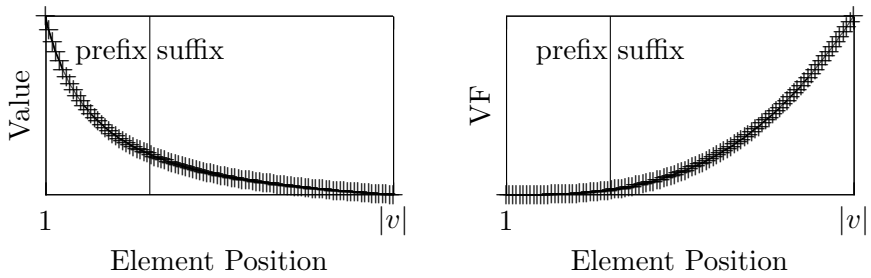
3.3 Constructing a k-Nearest Neighbor Graph

3.3.1 Greedy Filtering

Before presenting our algorithms, we introduce several distributions of datasets, as follows: Figure 3.2(a) shows the value of each element of a vector $v \in V$, where the value of the i^{th} element is larger than that of $(i+1)^{th}$ element. Figure 3.2(b)

v_1	d_1 0.5	d_3 0.37	d_8 0.33	d_4 0.31	d_9 0.23	...
v_2	d_1 0.73	d_2 0.55	d_5 0.37	d_3 0.1	d_8 0.05	...
v_3	d_2 0.4	d_7 0.29	d_6 0.27	d_1 0.25	d_{10} 0.1	...
v_4	d_5 0.8	d_4 0.35	d_3 0.3	d_2 0.27	d_1 0.25	...
v_5	d_3 0.48	d_5 0.37	d_7 0.34	d_4 0.32	d_{10} 0.2	...

Figure 3.1: Example of greedy filtering: the prefixes of vectors are colored. We assume that the hidden elements (as described by the ellipses) have a value of 0 and $k = 2$.



(a) Value Distribution

(b) Vector Frequency Distribution

Figure 3.2: Typical distributions after performing our preprocessing steps

shows the *vector frequency* of the i^{th} element of a vector $v \in V$, where the vector frequency of the i^{th} element is smaller than that of $(i+1)^{th}$ element. Let $dim(e)$ be the dimension of element e. Then the vector frequency of the element e is defined as the number of vectors that have the element of dimension $dim(e)$.

An interesting finding is that regardless of the dataset used, the dataset often follows distributions similar to those shown in Figure 3.2(a) and Figure 3.2(b) by performing some of the most common pre-processing steps. If we sort the elements of each vector in descending order according to their values, their value distributions will be similar to the distribution shown in Figure 3.2(a). Note that this pre-processing step does not change the similarity values

between vectors. Furthermore, if we weigh the value of each element according to a scheme that adds weights to the values corresponding to sparse dimensions, such as IDF, TF-IDF, or BM25, then the vector frequency distributions will be similar to the distribution shown in Figure 3.2(b). These weighting schemes are widely used in information retrieval and recommender systems along with popular similarity measures [40].

Let v' and v'' be the prefix and suffix of vector $v \in V$, respectively. The prefix v' consists of the first n elements and the suffix v'' consists of the last m elements such that $|v| = n + m$. Then $sim(v_i \in V, v_j \in V) = sim(v'_i, v'_j) + sim(v'_i, v''_j) + sim(v''_i, v'_j) + sim(v''_i, v''_j)$. Our intuition is as follows: assuming that the elements of each vector follows the distributions shown in Figure 2 and that the prefix and suffix of each vector is determined beforehand, $sim(v_i, v_j)$ would not have a high value when $sim(v'_i, v'_j) = 0$ because, first, $sim(v'_i, v''_j)$ would not have a high value; the vector frequencies of the elements in v'_i are so small that there is a low probability that v'_i and v''_j have a common dimension. Although there are some common dimensions in their elements, the values of the elements in v''_j are so small that they do not increase the similarity value significantly. For a similar reason, $sim(v''_i, v'_j)$ and $sim(v''_i, v''_j)$ would not have a high value: The elements of high values have low vector frequencies and the elements of high vector frequencies have low values. When $sim(v'_i, v'_j) \neq 0$, on the other hand, $sim(v_i, v_j)$ would have a relatively high value because v'_i and v'_j have the highest values.

If we generalize this intuition, we can assert that two vectors are not one of the k-nearest neighbors of each other if their prefixes do not have a single common dimension. That is to say, we would obtain an approximate k-NN graph by calculating the similarities between vectors that have at least one common dimension in their prefixes. Note that the vector frequencies of the prefixes are so small that they usually do not have a common dimension. Thus we can prune many vector pairs without computing the actual similarities. Because this ap-

proach initially checks whether the dimensions of large values match, we call it *greedy filtering*.

Definition 2 (Match) Let v_i and v_j be the vectors in V, and let e_i and e_j be any of the elements of v_i and v_j, respectively. We hold that e_i and e_j match if $dim(e_i) = dim(e_j)$. We also say that v_i and v_j match if any $e_i \in v_i$ and $e_j \in v_j$ match.

Definition 3 (Greedy Filtering) Greedy filtering returns for each vector $x \in V$, $argmax^k_{y \in V \wedge x \neq y \wedge match(x,y)} (sim(x, y))$, where $argmax^k$ returns the k arguments that give the highest values, and $match(x, y)$ is true if and only if x and y match.

Example 2. Figure 3.1 shows an example of greedy filtering, where the prefixes are colored. If we assume that the hidden elements (described by ellipses) have a value of 0 and $k = 2$, greedy filtering calculates the similarities of $\langle v_1, v_2 \rangle$, $\langle v_1, v_3 \rangle$, $\langle v_1, v_4 \rangle$, $\langle v_1, v_5 \rangle$, $\langle v_2, v_3 \rangle$, $\langle v_4, v_5 \rangle$, and $\langle v_1, v_4 \rangle$, filters out $\langle v_2, v_4 \rangle$, $\langle v_2, v_5 \rangle$, $\langle v_3, v_4 \rangle$ and $\langle v_3, v_5 \rangle$, and returns k-nearest neighbors for every vector: $\{v_2, v_4\}$, $\{v_3, v_1\}$, $\{v_2, v_1\}$, $\{v_5, v_1\}$, $\{v_4, v_1\}$.

Note that the result of Example 2 is slightly different from that of Example 1, because greedy filtering is an approximate algorithm. If the dataset follows the distributions similar to those of Figure 3.2, the algorithm will be more accurate. In Section 3.4 and 3.5, we will justify our intuition in more detail.

3.3.2 Prefix Selection Scheme

If we set the prefix such that $|v'_i| = |v_i|, \forall v_i \in V$, then greedy filtering generates the exact k-NN graph though its execution time will be very long. On the other hand, if we set the prefix such that $|v'_i| = 0, \forall v_i \in V$, then greedy filtering returns a graph with no edges while the algorithm will terminate immediately. Note that the elapsed time of greedy filtering and the quality of the constructed

graph depend on the prefix selection scheme. In general, there is a tradeoff between time and quality.

Assume that greedy filtering can find the approximate k-nearest neighbors for $v_i \in V$ if the number of matched vectors of v_i is equal to or greater than a small value μ. Then if for each vector v_i we find v'_i such that $|v'_i|$ is minimized and the number of matched vectors of v_i is at least μ, then we can expect a rapid execution of the algorithm and a graph of good quality.

Algorithm 1 describes our prefix selection scheme, where $e^j_{v_i}$ denotes the j^{th} element of vector v_i and $dim(e)$ denotes the dimension of element e. In line 2, we initially prepare an empty list for each dimension. Because one list $L[d_i]$ contains vectors that have the dimension of d_i in their prefixes, if any list has the two different vectors v_i and v_j, then greedy filtering will calculate the similarity between them. In lines 7-8, we insert the vectors in R into the lists, meaning that we increase the prefixes of the vectors in R by 1. In lines 10-13, we estimate the number of matched vectors, denoted by M, for each $v_i \in R$. In lines 14-16, we check the stop conditions for each vector and determine which vectors will increase their prefixes again.

Note that Algorithm 1 sacrifices two factors for the performance and ease of implementation. First, it allows the duplicate execution of the brute-force search (lines 19-20 of Algorithm 1 and lines 3-5 of Algorithm 2). If the two vectors v'_i and v'_j have the d number of dimensions that match, we will calculate the d number of calculations of $sim(v_i, v_j)$. Although we can avoid these redundant computations by exploiting a hash table, this is not good for scalability in general. Second, we overestimate the value μ for a similar reason: if the two vectors v'_i and v'_j have d number of dimensions that match, then M increases by d instead of 1. Also, we calculate the value of M only once per iteration; this makes M slightly larger.

Example 3. Figure 3.1 shows the result of our prefix selection scheme when

$\mu = 2$. Let $M(v)$ be the value M of the vector v. Initially, the prefix size of each vector is 1: $M(v_1) = M(v_2) = 1$ and $M(v_3) = M(v_4) = M(v_5) = 0$, because only $\langle v_1, v_2 \rangle$ match. As the next step, we increase the prefix sizes of all vectors by 1, as $M(v_i) < \mu, \forall v_i \in V$. Then $\langle v_1, v_5 \rangle$, $\langle v_2, v_3 \rangle$ and $\langle v_4, v_5 \rangle$ match. At this point, $M(v_1) = M(v_2) = M(v_5) = 2$ and $M(v_3) = M(v_4) = 1$. Thus we increase the prefix sizes of v_3 and v_4. As we continue until the stop condition is satisfied, our prefix selection scheme selects the colored elements shown in Figure 3.1.

Our prefix selection scheme has $O(|V|\,|D|^2)$ time complexity, and the brute-force search has to compare each vector v to M number of other vectors. However, our preliminary results show that the prefix sizes are so small that we can regard D as a constant. Furthermore, we set the variable M close to μ; empirically, M is not twice as large as μ. Assuming that D is a constant and $M = 2\mu$, the total complexity of greedy filtering is $O(|V| + 2\mu\,|V|) = O(|V|)$.

3.3.3 Optimization

Our algorithm uses a brute-force search a constant number of times for each vector. Because the execution times of the brute-force search highly dependent on the sizes of the vectors, it will take a relatively long time when a dataset contains very large vectors. For instance, experimental results show that the execution time of datasets whose vector sizes are relatively large, such as TREC 4-gram, is longer than that of other datasets.

We present one variation of greedy filtering, called *fast greedy filtering.* The main idea of this approach is that if $sim(v'_i, v'_j)$ is relatively high, then $sim(v_i, v_j)$ will also be relatively high. Then we can formulate an approximate k-NN graph by calculating the similarities between prefixes. Algorithm 1 and Algorithm 3 describe the process of this algorithm in detail: $e^j_{v_i}$ denotes the j^{th} element of vector v_i, and $dim(e)$ and $value(e)$ denote the dimension and value of element e, respectively. In Algorithm 1, we set the prefix of each vector according to the abovementioned prefix selection scheme and invoke Algorithm

Algorithm 1: Greedy-Filtering (V, μ)

Input: a set of vectors V, a parameter μ

Output: k-NN queues Q

```
1  begin
2      L[d_i] ⟵ φ, ∀d_i ∈ D /* candidates */
3      C ⟵ 1 /* an iteration counter */
4      R ⟵ V /* vectors to be processed */
5      repeat
6          /* find candidates */
7          for v_i ∈ R do
8              add v_i to L[dim(e^C_{v_i})]
9          /* check stop conditions */
10         for v_i ∈ R do
11             M ⟵ 0
12             for j ← 1 to C do
13                 M ⟵ M + |L[dim(e^j_{v_i})]|
14             if M ≥ μ or C ≥ |v_i| then
15                 P[v_i] ⟵ C
16                 remove v_i from R
17         C ⟵ C + 1
18     until |R| > 0
19     if default algorithm then
20         return Brute-force-search(L)
21     else
22         return Inverted-index-join(V, P)
```

Algorithm 2: Brute-force-search (L)

Input: lists for dimensions L

Output: k-NN queues Q

```
1 begin
2     Q[v_i] ⟵ φ, ∀v_i ∈ V /* empty queues */
3     for d_i ∈ D do
4         compare all vector pairs ⟨v_x, v_y⟩ in L[d_i]
5         update the priority queues, Q[v_x] and Q[v_y]
6     return Q
```

3. Then in lines 6-12 of Algorithm 3, we calculate the similarities between the current vector $v_i \in V$ and the other vectors already indexed and update the k-nearest neighbors of v_i and the indexed vectors. In lines 14-15, we put the current vector v_i into the inverted indices. Unlike greedy filtering, the execution time of fast greedy filtering is highly dependent on the number of dimensions and the vector frequencies of the datasets rather than the vector sizes.

Definition 4 (Fast Greedy Filtering) For each vector $x \in V$, fast greedy filtering returns $argmax^k_{y \in V \wedge x \neq y \wedge match(x,y)} (sim(x', y'))$, where $argmax^k$ returns the k arguments that give the highest values, and where $match(x, y)$ is true if and only if x and y match.

Example 4. If we apply fast greedy filtering to the example in Figure 1, the algorithm returns slightly different results: $\{v_2, v_5\}$, $\{v_3, v_2\}$, $\{v_2, v_1\}$, $\{v_5, v_1\}$, $\{v_4, v_5\}$ when $k = 2$.

3.4 Theoretical Analysis

Greedy filtering constructs an approximate k-NN graph by calculating the similarities between vectors that have at least one common "rare" dimension. In

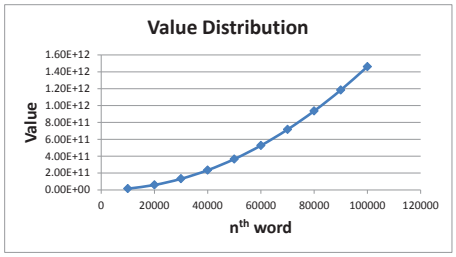
Algorithm 3: Inverted-index-join (L, P)

Input: a set of vectors V, prefix sizes P

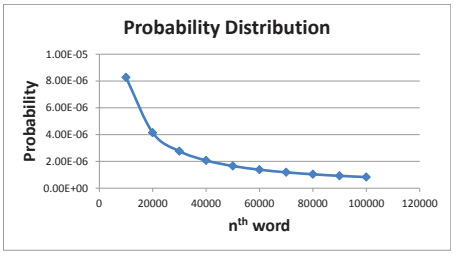
Output: k-NN queues Q

```
1  begin
2      Q[v_i] ⟵ φ, ∀v_i ∈ V /* empty queues */
3      I[d_i] ⟵ φ, ∀d_i ∈ D /* empty indices */
4      for v_i ∈ V do
5          /* verification phase */
6          C[v_j] ⟵ 0, ∀v_j ∈ V /* sim(v_i, v_j) = 0 */
7          for l ← 1 to P[v_i] do
8              for ⟨v_j, r_j⟩ ∈ I[dim(e^l_{v_i})] do
9                  C[v_j] ⟵ C[v_j] + r_j * value(e^l_{v_i})
10         for v_j ∈ V do
11             if C[v_j] > 0 then
12                 update the queues, Q[v_x] and Q[v_y]
13         /* indexing phase */
14         for l ← 1 to P[v_i] do
15             add ⟨v_i, value(e^l_{v_i})⟩ to I[dim(e^l_{v_i})]
16     return Q
```

this section, we will show that this approach is effective for sparse datasets where each vector component follows zipfian distribution and is weighted by a TF-IDF weighting scheme, assuming that we use *dot product* as a similarity measure. Because cosine similarity belong to a family of dot product similarity measures, we can extend the following lemmas and theorem to those of cosine similarity.



(a) Value Distribution



(b) Probability Distribution

Figure 3.3: The value and probability distributions of words

3.4.1 Preliminaries

Zipfian distribution. Let V be a set of N-dimensional vectors. Let w_n be the n^{th} word in our vocabulary. We assume that vectors represent documents. Then N indicates the number of words in our vocabulary, and the n^{th} component of vector $v \in V$ represents the frequency of w_n in v. We also assume that frequency of words follows zipfian distribution. That is to say, for vectors in V, the normalized frequency of the n^{th} word is as follows:

$$p_n = \frac{1/n^s}{\sum_{i=1}^{N}(1/i^s)} \tag{3.3}$$

Here, s is the parameter of the zipfian distribution. We set $s = 1$ in this dissertation.

TF-IDF weighting scheme. For vectors $v \in V$, we assume that the n^{th} component is weighted by the following TF-IDF weighting scheme:

$$TF_{v,n} = \begin{cases} 1, \text{ the } n^{th} \text{ word frequency in } v > 0 \\ 0, \text{ otherwise} \end{cases} \tag{3.4}$$

$$IDF_n = \frac{1}{(p_n)^2} \tag{3.5}$$

Figure 3.3 shows the distributions of words assuming that the words follow the zipfian distribution and are weighted by TF-IDF. Note if we sort the elements in descending order of value, then we will see the distributions similar to

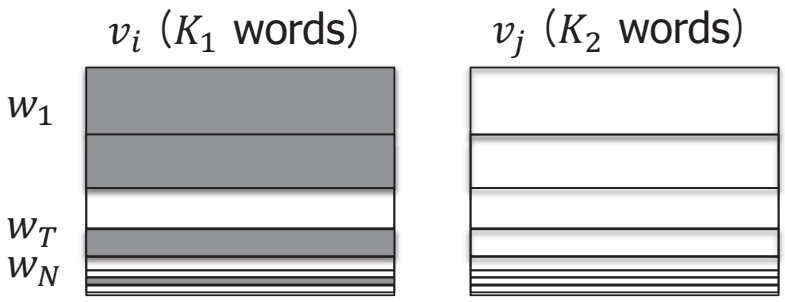


Figure 3.4: An illustrative example of the analysis of graph construction time

those of Figure 3.2.

Dot product similarity measure. For vectors $v_i \in V$ and $v_j \in V$, the dot product similarity measure is defined as follows:

$$sim(v_i, v_j) = v_i \cdot v_j \tag{3.6}$$

3.4.2 Graph Construction Time

In this subsection, we prove that the greedy filtering significantly reduces the number of candidates by showing that the probability of two random vectors having any rare word in common is very low (Lemma 1 and Example 5). Figure 3.4 shows a conceptual example of two random vectors, where a shaded region indicates that the corresponding word occurs at least once, and the vertical length of each region indicates the probability that the corresponding word occurs at least once.

Lemma 1. Let K_1 and K_2 be positive non-zero integers, and v_i and v_j be vectors constructed by randomly generated K_1 and K_2 words, respectively. Let T be an integer such that $1 \leq T \leq N$. If $w_T, ..., w_N$ are rare words and the other words are not rare, then the probability that v_i and v_j have at least one common rare word is

$$p_{match} = 1 - \left(1 - \left(\sum_{n=T}^{N} \left(1 - (1 - p_n)^{K_1}\right) p_n\right)\right)^{K_2} \tag{3.7}$$

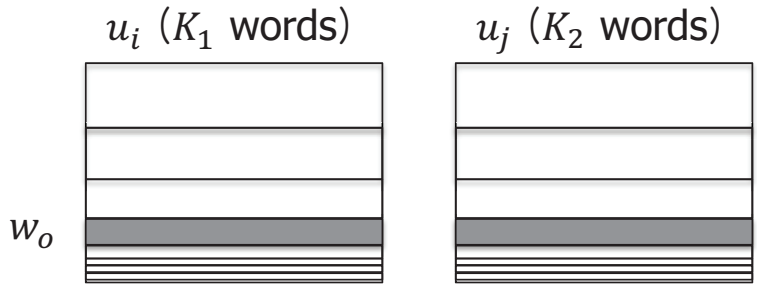


Figure 3.5: An illustrative example of the analysis of graph accuracy

Proof: The probability that w_n is at least once appeared in v_i is $1-(1-p_n)^{K_1}$. Hence the expected value of the sum of the probability of rare words at least once appeared in v_i is $\sum_{n=T}^{N}\left(1-(1-p_n)^{K_1}\right)p_n$. Since the probability that v_j does not have any rare word in common with v_i is $\left(1-\left(\sum_{n=T}^{N}\left(1-(1-p_n)^{K_1}\right)p_n\right)\right)^{K_2}$, it follows that

$$p_{match}=1-\left(1-\left(\sum_{n=T}^{N}\left(1-(1-p_n)^{K_1}\right)p_n\right)\right)^{K_2} \tag{3.8}$$

Example 5. Given N, T, K_1, and K_2, we can calculate p_{match} as follows:

- If $N=100,000, T=90,000, K_1=K_2=1,000$, then $p_{match}=0.00757009$.
- If $N=100,000, T=80,000, K_1=K_2=1,000$, then $p_{match}=0.0169509$.
- If $N=100,000, T=90,000, K_1=K_2=2,000$, then $p_{match}=0.0299255$.
- If $N=110,000, T=100,000, K_1=K_2=1,000$, then $p_{match}=0.00610197$.

3.4.3 Graph Accuracy

In this subsection, we show that if two random vectors have a rare word w_o in common as described in Figure 3.5, then the expected value of the similarity between the vectors is much higher than that of another two random vectors (Lemma 2, 3, Theorem 1 and Example 6).

Lemma 2. Define $c_n = (IDF_n)^2$. Then the expected value of the similarity between two random vectors consisting of K_1 and K_2 words, respectively, is

$$E_{K_1,K_2} = \sum_{n=1}^{N} \left(1 - (1-p_n)^{K_1}\right)\left(1 - (1-p_n)^{K_2}\right) \cdot c_n \tag{3.9}$$

Proof. Let v_i and v_j be vectors constructed by randomly generated K_1 and K_2 words. That is to say, $E_{K_1,K_2} = E[v_i \cdot v_j]$. Since each component of a vector has its corresponding TF-IDF value, the expected value of the similarity between v_i and v_j is

$$E[v_i \cdot v_j] = E\left[\sum_{n=1}^{N} TF_{v_i,n} \cdot TF_{v_j,n} \cdot c_n\right] \tag{3.10}$$

By linearity of expectation,

$$E\left[\sum_{n=1}^{N} TF_{v_i,n} \cdot TF_{v_j,n} \cdot c_n\right] = E[TF_{v_i,1} \cdot TF_{v_j,1}]c_1 + ... + E[TF_{v_i,N} \cdot TF_{v_j,N}]c_N \tag{3.11}$$

Since the probability that $TF_{v_i,n} = 1$ is $(1 - (1-p_n)^{K_1})$ and the probability that $TF_{v_j,n} = 1$ is $(1 - (1-p_n)^{K_2})$, it follows that

$$E[TF_{v_i,n} \cdot TF_{v_j,n}] = \left(1 - (1-p_n)^{K_1}\right)\left(1 - (1-p_n)^{K_2}\right) \tag{3.12}$$

Hence,

$$E[v_i \cdot v_j] = \sum_{n=1}^{N} \left(1 - (1-p_n)^{K_1}\right)\left(1 - (1-p_n)^{K_2}\right) \cdot c_n \tag{3.13}$$

Lemma 3. Define Δ_1 and Δ_2 as follows:

$$\Delta_1 = \sum_{n=1}^{N} p_n(1-p_n)^{K_1-1} \cdot \left(1 - (1-p_n)^{K_2-1}\right) \cdot c_n \tag{3.14}$$

$$\Delta_2 = \sum_{n=1}^{N} p_n(1-p_n)^{K_2-1} \cdot \left(1 - (1-p_n)^{K_1}\right) \cdot c_n \tag{3.15}$$

Then,

$$E_{K_1,K_2} = E_{K_1-1,K_2-1} + \Delta_1 + \Delta_2 \tag{3.16}$$

Proof. Let $v_i^{'}$ and $v_j^{'}$ be vectors constructed by randomly generated K_1-1 words and K_2-1 words, respectively. Note $E[v_i \cdot v_j] = E_{K_1-1,K_2-1}$. If we insert one random word into $v_i^{'}$, then the expected value is increased by

$$\Delta_1 = \sum_{n=1}^{N} p_n(1-p_n)^{K_1-1} \cdot \left(1-(1-p_n)^{K_2-1}\right) \cdot c_n \tag{3.17}$$

Then since $v_i^{'}$ and $v_j^{'}$ has K_1 and K_2-1 words, respectively, if we insert one random word into $v_j^{'}$, then the expected value is increased by

$$\Delta_2 = \sum_{n=1}^{N} p_n(1-p_n)^{K_2-1} \cdot \left(1-(1-p_n)^{K_1}\right) \cdot c_n \tag{3.18}$$

Since the expected value of $v_i^{'} \cdot v_j^{'}$ before and after inserting the two words are E_{K_1-1,K_2-1} and E_{K_1,K_2}, respectively, it follows that

$$E_{K_1,K_2} = E_{K_1-1,K_2-1} + \Delta_1 + \Delta_2 \tag{3.19}$$

Theorem 1. For a given integer o such that $1 \neq o \neq N$, we assume that w_o is a rare word. Let u_i and u_j be vectors constructed by randomly generated K_1 and K_2 words, respectively, such that they have non-zero o^{th} components. Then,

$$E[u_i \cdot u_j] = E[v_i \cdot v_j] - (\Delta_1 + \Delta_2) - \epsilon + c_o \tag{3.20}$$

Proof. Let $v_i^{'}$ and $v_j^{'}$ be vectors constructed by randomly generated K_1-1 words and K_2-1 words, respectively. Since $E[u_i \cdot u_j]$ is the sum of c_o and the expected value that can be additionally obtained by $v_i^{'} \cdot v_j^{'}$, it follows that

$$E[u_i \cdot u_j] = c_o + E[v_i^{'} \cdot v_j^{'}] - (1-(1-p_o)^{K_1-1})(1-(1-p_o)^{K_2-1}) \cdot c_o \tag{3.21}$$

Since w_o is a rare word, $1-(1-p_o)^{K_1-1})(1-(1-p_o)^{K_2-1}) \cdot c_o \ll c_o$. Since $E[v_i^{'} \cdot v_j^{'}] = E_{K_1-1,K_2-1}$, by Lemma 3, $E[v_i^{'} \cdot v_j^{'}] = E[v_i \cdot v_j] - (\Delta_1 + \Delta_2)$. Hence,

$$E[u_i \cdot u_j] = E[v_i \cdot v_j] - (\Delta_1 + \Delta_2) - \epsilon + c_o \tag{3.22}$$

Example 6. Given N, K_1, K_2, and o, the ratio of the two expected values is calculated as follows:

- If $N = 100,000, K_1 = 1,000, K_2 = 1,000$, and $o = 100,000$, then $\frac{E[u_i \cdot u_j]}{E[v_i \cdot v_j]} = 15.71$.

- If $N = 100,000, K_1 = 1,000, K_2 = 1,000$, and $o = 90,000$, then $\frac{E[u_i \cdot u_j]}{E[v_i \cdot v_j]} = 12.91$.

- If $N = 110,000, K_1 = 1,000, K_2 = 1,000$, and $o = 100,000$, then $\frac{E[u_i \cdot u_j]}{E[v_i \cdot v_j]} = 14.57$.

- If $N = 100,000, K_1 = 2,000, K_2 = 2,000$, and $o = 100,000$, then $\frac{E[u_i \cdot u_j]}{E[v_i \cdot v_j]} = 4.69$.

Note Example 6 shows that $E[u_i \cdot u_j]$ is much higher than $E[v_i \cdot v_j]$, because $\Delta_1 + \Delta_2$ is much smaller than c_o. Lemma 4 and Example 7 show the relationship between $\Delta_1 + \Delta_2$ and c_o more clearly.

Lemma 4. If $K = K_1 = K_2$ and $p_1 \leq 0.5$, then

$$\Delta_1 + \Delta_2 \leq 0.75 \cdot ln(N+1) \cdot \frac{N(N+1)}{2} \tag{3.23}$$

Proof. Since $x(1-x) \leq 0.25$, $1/(1-p_n) \leq 2$, and the N^{th} harmonic number can be interpreted as a Riemann sum of the integral, $\int_1^{N+1} \frac{1}{x} dx = ln(N+1)$,

the following inequalities holds:

$$\begin{aligned}
\Delta_1 &= \sum_{n=1}^{N} p_n(1-p_n)^{K-1} \cdot (1-(1-p_n)^{K-1}) \cdot c_n \\
&= \sum_{n=1}^{N} (1-p_n)^{K-1} \cdot (1-(1-p_n)^{K-1}) \cdot \frac{1}{p_n} \\
&\leq 0.25 \cdot \sum_{n=1}^{N} \frac{1}{p_n} \\
&= 0.25 \cdot \sum_{n=1}^{N} \frac{\sum_{i=1}^{N} \frac{1}{i}}{\frac{1}{n}} \\
&= 0.25 \cdot \sum_{i=1}^{N} \frac{1}{i} \cdot \sum_{n=1}^{N} n \\
&= 0.25 \cdot ln(N+1) \cdot \frac{N(N+1)}{2}
\end{aligned} \tag{3.24}$$

$$\begin{aligned}
\Delta_2 &= \sum_{n=1}^{N} p_n(1-p_n)^{K-1} \cdot (1-(1-p_n)^{K}) \cdot c_n \\
&= \sum_{n=1}^{N} (1-p_n)^{K} \cdot (1-(1-p_n)^{K}) \cdot \frac{1}{1-p_n} \cdot \left(\frac{\sum_{i=1}^{N} \frac{1}{i}}{\frac{1}{n}} \right) \\
&\leq 0.25 \cdot ln(N+1) \cdot \frac{N(N+1))}{2} \cdot \frac{1}{1-p_1} \\
&\leq 0.5 \cdot ln(N+1) \cdot \frac{N(N+1)}{2}
\end{aligned} \tag{3.25}$$

Hence,

$$\Delta_1 + \Delta_2 \leq 0.75 \cdot ln(N+1) \cdot \frac{N(N+1)}{2} \tag{3.26}$$

Example 7. If $N = 100,000$ and $o = 100,000$, then $\Delta_1 + \Delta_2 \leq 4.31740 \cdot 10^{10}$ and $c_o = 1.46172 \cdot 10^{12}$. Thus $\Delta_1 + \Delta_2 \ll c_o$.

3.5 Experiments

3.5.1 Experimental Setup

Algorithms. We considered eight types of algorithms for a comparison. Three algorithms among them adopt the similarity join (abbreviated by SIM) [32], the

top-k similarity join (TOP) [41], and similarity search (LSH) [19] approaches. Two algorithms among them are NN-Descent (DE1) and Fast NN-Descent (DE2) [24], originally developed for the purpose of constructing k-NN graphs. The other two algorithms are greedy filtering (GF1) and fast greedy filtering (GF2) algorithms as proposed in this paper. Finally, we use the inverted index join (IDX) [32], which calculates all similarities with inverted indices, as a baseline algorithm. In all experiments, we set the number of neighbors to 10 (k=10).

We adopted the similarity join algorithm for k-NN graph construction. First, we implement the vector similarity join algorithm, *MM-join* [32], which outperforms the All-pairs algorithm [33] in various datasets. Then, we iterate the execution of the algorithm while decreasing the threshold ϵ by δ until either at least s% of vectors find k-nearest neighbors or until the elapsed time is higher than that of inverted index join. We used the following values in the experiments: $\epsilon = 1.00$ (the initial value), $\delta = 0.05$ and $s = 30$.

Adapting the top-k similarity join algorithm [41] for the k-NN graph construction process is along the same lines as that of the similarity join algorithm, except (1) we increase the parameter k at each iteration instead of decreasing δ, and (2) because the top-k similarity join algorithm uses *sets* as data structures, we need to transform the data structures into vectors and set new upper bounds for the suffixes of vectors using the prefix filtering and length filtering conditions. We set $s = 70$ for the top-k similarity join algorithm.

We also adopted the similarity search algorithm for k-NN graph construction by executing the algorithm N times. We used random hyperplane-based locality sensitive hashing for cosine similarity [19]. We cannot adopt other LSH algorithms, such as those in Broder et al. [38] or Gionis et al. [36], as they were originally developed for other similarity measures. We set the number of signatures for each vector to 100.

Table 3.1: Datasets and statistics

Dataset Statistics	$\|V\|$	$\|D\|$	Avg. Size	Avg. VF
DBLP	**250,000**	163,841	5.14	7.85
TREC	125,000	484,733	79.83	20.59
Last.fm	125,000	56,362	4.78	10.60
DBLP 4-gram	150,000	279,380	27.97	15.02
TREC 4-gram	50,000	**731,199**	**509.20**	34.82
Last.fm 4-gram	100,000	194,519	20.77	10.68
MovieLens	60,000	10,653	141.23	**795.44**

Datasets. We considered seven types of datasets for a comparison. There are two document datasets (DBLP[1] and TREC[2]), one text dataset that consists of music metadata (Last.fm[3]), three artificial text datasets (DBLP 4-gram, TREC 4-gram and Last.fm 4-gram), and one log dataset that consists of the movie ratings of users (MovieLens[4]). Note DBLP 4-gram, TREC 4-gram, and Last.fm 4-gram are derived from DBLP, TREC and Last.fm, respectively. We remove whitespace characters in the original vectors and extracted the 4-gram sequences from them. Table 3.1 shows their major statistics, where $|V|$ denotes the number of vectors and $|D|$ is the number of dimensions, *Avg. Size* denotes the average size of all vectors, and *Avg. VF* is defined as the average vector frequencies of all dimensions.

Evaluation Measures. We use the execution time and the scan rate as the measures of performance. The execution time is measured in seconds; it does not include the data preprocessing time, which accounts for only a minor portion as indicated in Figure 3.6. Since the preprocessing time comprises of the time for

[1] `http://dblp.uni-trier.de/xml/`
[2] `http://trec.nist.gov/data/t9_filtering.html/`
[3] `http://www.last.fm/`
[4] `http://grouplens.org/datasets/movielens/`

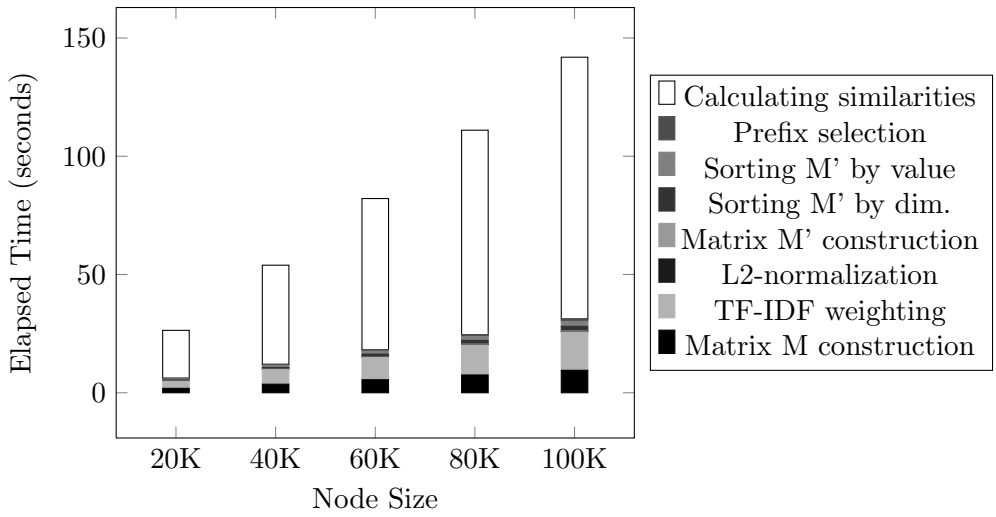


Figure 3.6: Elapsed time for each task (greedy filtering, k-NN graph construction, and the New York Times dataset)

1) constructing a sparse matrix M, 2) TF-IDF weighting, 3) L_2-normalization, 4) creating a copy, matrix $M^{'}$, of matrix M, 5) sorting M by dimension and 6) sorting $M^{'}$ by value, the execution time only consists of the time for prefix selection (finding candidates) and calculating similarities. The scan rate is defined as follows:

$$Scan\ Rate = \frac{\#\ similarity\ calculations}{|V|\,(|V|-1)/2} \tag{3.27}$$

The *similarity calculation* expresses the exact calculation of the similarity between a pair. Thus, the brute-force search and the inverted index join always have a scan rate of 1, as they calculate all of the similarities between vectors. On the other hand, fast greedy filtering has a scan rate of 0 because this algorithm only estimates the degrees of similarity.

We use the level of accuracy as the measure of quality. Assuming that an algorithm returns k neighbors for each vector, the accuracy of the algorithm is defined as follows:

$$Accuracy = \frac{\#\ correct\ k\text{-}nearest\ neighbors}{k\,|V|} \tag{3.28}$$

Weighting Schemes. The value of each element can be weighted by the popular weighting scheme, such as *TF-IDF*. Let v be a vector in V and e_i be an element in v. Then, we define the TF-IDF as follows:

$$tf\text{-}idf(e_i, v) = \left(0.5 + \frac{0.5 * value(e_i)}{max\left\{value(e_j) : e_j \in v\right\}}\right) * \left(log\frac{|V|}{VF(e_i)}\right), \quad (3.29)$$

Here, $value(e)$ is the initial value of e. In the text datasets, the initial values are the term frequencies; in the MovieLens dataset, the values are the ratings.

3.5.2 Performance Comparison

Comparison of All Algorithms. Figure 3.7 and Table 3.2 show the execution time, accuracy, and scan rate of all algorithms with a small number of TREC nodes. We do not specify the accuracy and scan rate of inverted index join in Table 3.2, as its accuracy is always 1 and its scan rate is always 0. By the same token, the scan rates of LSH and GF2 are left blank. We set $\mu = 300$ for our greedy filtering algorithms.

The experimental results show that the greedy filtering approaches (GF1 and GF2) outperform all other approximate algorithms in terms of the execution time, accuracy and scan rate. The second best algorithms behind GF1 and GF2 are the NN-Descent algorithms (DE1 and DE2). However, as already descrbed in work by Dong et al., the accuracy of the algorithms significantly decreases as the number of dimensions scales up. The other algorithms require either a long execution time or return results that are not highly accurate. The top-k similarity join and similarity join algorithms require a considerable amount of time to construct k-NN graphs, and locality sensitive hashing based on random hyperplanes requires many signatures (more than 1,000 signatures in our experimental settings) to ensure a high level of accuracy.

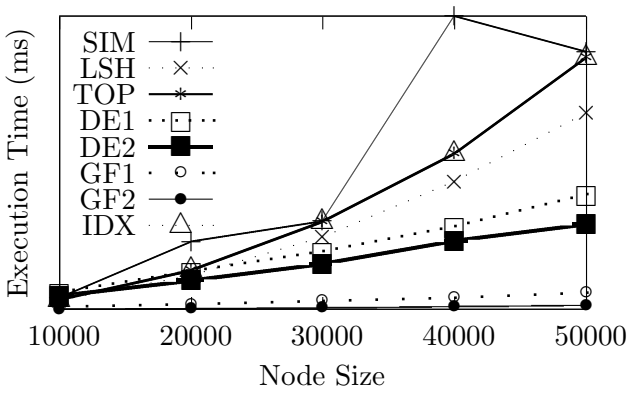


Figure 3.7: Execution time of all algorithms (TREC)

Table 3.2: Accuracy and scan rate of all algorithms

Node	Accuracy (TREC)							Scan Rate (TREC)						
	SIM	LSH	TOP	DE1	DE2	GF1	GF2	SIM	LSH	TOP	DE1	DE2	GF1	GF2
10K	0.00	0.01	0.68	0.54	0.43	**0.96**	0.65	0.05	-	0.06	0.27	0.19	**0.05**	-
20K	0.00	0.01	0.76	0.50	0.38	**0.95**	0.63	0.07	-	0.08	0.15	0.11	**0.03**	-
30K	0.00	0.01	0.76	0.48	0.36	**0.94**	0.62	0.05	-	0.08	0.10	0.08	**0.03**	-
40K	0.00	0.01	0.78	0.47	0.34	**0.93**	0.61	0.08	-	0.08	0.08	0.06	**0.02**	-
50K	0.00	0.01	0.79	0.46	0.33	**0.93**	0.59	0.05	-	0.08	0.07	0.05	**0.02**	-

Comparison of All Datasets. Table 3.3 shows the comparison results of the two outperformers, greedy filtering and NN-Descent, over the seven types of datasets with the TF-IDF weighting scheme. The results of their optimized versions are specified within the parentheses. In this table, we define a new measure, *time*, as the execution time divided by the execution time of inverted index join. We set the parameters μ such that the accuracy of GF1 is at least 90%. The experimental results show that GF1 outperforms the NN-Descent algorithms in all of the datasets except for DBLP and MovieLens. Although the execution time of GF1 is slower than the times required by the the NN-Descent algorithms for the two datasets, its accuracy is much higher.

Note that while fast greedy filtering exploits inverted index join instead of brute-force searches, it is not always faster than greedy filtering. Fast greedy filtering can be more effective in a dataset for which the vector sizes are relatively large and the number of dimensions and the vector frequencies are relatively small. For example, fast greedy filtering outperforms the other algorithms in the TREC 4-gram datasets, which have the largest average size.

Performance Analysis. Recall that before executing the greedy filtering algorithm, we utilize some of the most common pre-processing steps, as described in Section 3.3. First, we weigh the value of each element according to a weighting scheme, and then we sort the elements of each vector in descending order according to their values. Figure 3.8 shows the distributions of all of our datasets after performing these pre-processing steps. Note that the distributions after pre-processing are similar to those in Figure 3.2. Note also that Figure 3.8 and the experimental results are in accord with our intuition as presented in Section 3.3: for example, the distributions of DBLP 4-gram, Last.fm, and TREC in Figure 3.8 are very similar to those shown in Figure 3.2; moreover, the experimental results in Table 3.3 show that their execution time is better than those of the other datasets. As another example, the distributions of Movie-

Lens with TF-IDF are relatively less similar to those shown in Figure 3.2 in that the element position does not greatly affect the vector frequency. Thus, their performance is slightly worse than the performance levels of the other datasets.

3.6 Summary

In this chapter, we present *greedy filtering,* an efficient and scalable algorithm for finding an approximate k-nearest neighbor graph by filtering node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a linear time complexity, our algorithm chooses essentially a fixed number of node pairs as candidates for every node. We also present *fast greedy filtering* based on the use of inverted indices for the node prefixes. We demonstrate the effectiveness of these algorithms through extensive experiments in which we compare various types of algorithms and datasets.

The limitation of our approaches is that they are specialized for high dimensional sparse datasets, weighting schemes that add weight to the values corresponding to sparse dimensions, and cosine similarity measure. In chapter 5, we present more generalized algorithms using locality-sensitive hashing.

Table 3.3: Comparison of all datasets

Datasets (TF-IDF)	DE1 (DE2)			GF1 (GF2)		
	Time	Accuracy	Scan Rate	Time	Accuracy	Scan Rate
DBLP	0.015 **(0.013)**	0.14 (0.11)	0.005 **(0.004)**	0.242 (0.076)	**0.98** (0.90)	0.102 (-)
TREC	0.190 (0.140)	0.43 (0.27)	0.030 (0.021)	0.030 **(0.009)**	**0.90** (0.56)	**0.007** (-)
Last.fm	0.322 (0.189)	0.69 (0.68)	0.014 (0.008)	**0.063** (0.149)	**0.98** (0.80)	**0.003** (-)
DBLP 4-gram	0.066 (0.046)	0.52 (0.34)	0.019 (0.011)	**0.004** (0.006)	**0.93** (0.59)	**0.001** (-)
TREC 4-gram	0.228 (0.163)	0.60 (0.42)	0.066 (0.047)	0.106 **(0.003)**	**0.90** (0.48)	**0.035** (-)
Last.fm 4-gram	1.207 (0.800)	0.65 (0.65)	0.013 (0.008)	**0.139** (0.204)	**0.90** (0.59)	**0.001** (-)
MovieLens	0.244 (0.161)	0.55 (0.38)	0.046 (0.028)	0.302 **(0.013)**	**0.90** (0.19)	**0.073** (-)

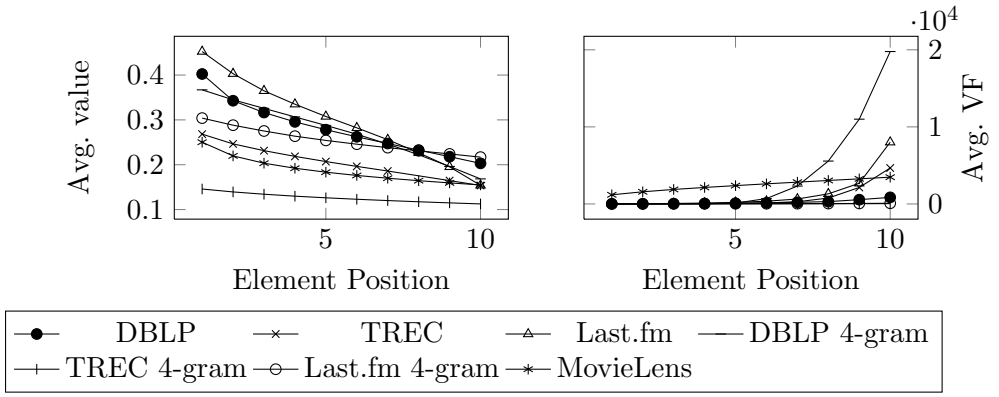


Figure 3.8: Distributions of all datasets

Chapter 4

Fast Collaborative Filtering

User-based and item-based collaborative filtering (CF) methods are two of the most widely used techniques in recommender systems. While these algorithms are widely used in both industry and academia owing to their simplicity and acceptable level of accuracy, they require a considerable amount of time to find k similar neighbors (items or users) and predict user preferences of unrated items. In this chapter, we present Reversed CF (RCF), a rapid CF algorithm which utilizes a k-nearest neighbor (k-NN) graph. One main idea of this approach is to reverse the process of finding k neighbors; instead of finding k similar neighbors of unrated items, RCF finds the k-nearest neighbors of rated items. Not only does this algorithm perform fewer predictions while filtering out inaccurate results, but it also enables the use of fast k-NN graph construction algorithms such as greedy filtering. The experimental results show that our approach outperforms traditional user-based/item-based CF algorithms in terms of both the pre-processing time and the query processing time without sacrificing the level of accuracy.

4.1 Introduction

User-based and item-based collaborative filtering (CF) methods are two of the most widely used techniques in recommender systems. When a user requests a recommendation, the user-based CF algorithm [1] predicts the user's preferences for all of the unrated items based on similar user preferences for those items. In a similar way, the item-based CF algorithm [2] predicts the preferences of the user for all unrated items based on the preference levels of similar items for the user.

Earlier studies in these areas indicated that CF algorithms produce movie recommendations of a higher quality compared to baseline algorithms, which only recommend the most popular movies or highly rated movies [50]. Although there have been proposed more efficient algorithms, such as those that use singular vector decomposition [51] or a random walk [50][51], CF algorithms are still widely used in both industry and academia owing to their simplicity and acceptable levels of accuracy. For example, the Amazon and YouTube recommender systems [52][3] utilize CF-based algorithms, and many modified versions of CF algorithms continue to be proposed for the purpose of building context-aware recommender systems [4][5].

One of the main drawbacks of CF algorithms is that predictions are necessary for all unrated items. While such an approach facilitates evaluations of the accuracy of various algorithms using the root-mean-square error (RMSE), this method consumes a significant amount of recommendation time. Moreover, the pre-processing time is also long, especially for a user-based CF algorithm, as it has to calculate all of the similarity values between users.

In this chapter, we present Reversed CF (RCF), a fast CF algorithm using a k-nearest neighbor (k-NN) graph. One main idea of this approach is that it reverses the process of finding k neighbors. Not only does this algorithm perform fewer predictions while filtering out inaccurate results, but it also enables the

use of fast k-NN graph construction algorithms, such as greedy filtering. More specifically, the contributions of this paper are as follows:

- We present RCF, a fast CF algorithm which uses a k-NN graph. Because RCF performs fewer rating predictions, the recommendation time (query processing time) is dramatically reduced compared to that of a user-based or an item-based CF algorithm.
- We apply a fast k-NN graph construction algorithm known as greedy filtering to reduce the RCF pre-processing time significantly. We also apply the TF-IDF weighting scheme to our dataset before executing the greedy filtering algorithm for further improvements.
- We conduct experiments with different parameter settings and show that RCF outperforms traditional user-based/item-based CF algorithms in terms of both the pre-processing time and the recommendation time without sacrificing the level of accuracy.

The rest of this paper is structured as follows. In Section 4.2, we review user-based/item-based CF algorithms and their general optimization techniques. In Section 4.3, we present RCF, a fast collaborative filtering algorithm. In Section 4.4, we show experimental results comparing our approach to the traditional CF algorithms. Finally, we conclude this chapter and present future research directions in Section 4.5.

4.2 Related Work

[6] classified existing recommender systems into six categories based on the types of recommendation approach (content-based filtering, collaborative filtering, and hybrid approach), and the types of recommendation techniques (heuristic-based approach and model-based approach) for the rating estimation. Although the traditional collaborative and heuristic-based approaches are

outperformed by the different types of recommender systems, especially the model-based approaches such as one using matrix factorization introduced by [54], singular vector decomposition presented by [59], or random walk proposed by [51] and [52], in terms of prediction accuracy, [58] state that the traditional approaches are still widely used due to their simplicity, justifiability, efficiency and stability. For example, according to [3] and [4], the Amazon and YouTube recommender systems exploit the collaborative and heuristic-based approaches. In this paper, we focus on the item-based CF and user-based CF algorithms, which are two of the most popular approaches among them.

User-based CF algorithms predict the preferences of all items unrated by the user based on similar user preferences for those items. According to [1], the predicted rating for active user a for item i is defined as follows:

$$p_{a,i} = \overline{r_a} + \frac{\sum_{n \in N(a)} (r_{n,i} - \overline{r_n}) * sim(a,n)}{\sum_{n \in N(a)} |sim(a,n)|}, \tag{4.1}$$

where $N(a)$ denotes the set of k-nearest neighbors of a among the users that have rated item i; $r_{n,i}$ denotes the rating of item i by user n; $\overline{r_a}$ and $\overline{r_n}$ are the average ratings of user a and neighbor n, respectively; $sim(a,n)$ is the similarity between a and n. We use the Pearson correlation coefficient as the similarity measure for the user-based CF algorithm:

$$sim(a,n) = \frac{\sum_{i \in \mathcal{C}_i} (r_{a,i} - \overline{r_a})(r_{n,i} - \overline{r_n})}{\sqrt{\sum_{i \in \mathcal{C}_i} (r_{a,i} - \overline{r_a})^2} \sqrt{\sum_{i \in \mathcal{C}_i} (r_{n,i} - \overline{r_n})^2}}, \tag{4.2}$$

where $\mathcal{C}_i$ denotes the set of co-rated items.

[1] and [58] state that there are common optimization techniques for the user-based CF algorithm, such as *significance weighting*, *variance weighting*, and *selecting neighborhoods*. The first two techniques are used to adjust the similarity values between users. If two users had fewer than 50 commonly rated items, significance weighting devalues the similarity between them by (1 - # commonly rated items / 50) * 100%; variance weighting decreases the influence of items with low variance, such as "Titanic." The third technique selects only

k neighbors when predicting the ratings of unrated items in that the use of less similar users may have a negative impact on the quality of recommendations.

Item-based CF algorithms predict the preferences of items unrated by the user based on the preference levels of similar items for the user. According to [2], the predicted rating for active user a for unrated item i is defined as follows:

$$p_{a,i} = \frac{\sum_{n \in N(i)} r_{a,n} * sim(i,n)}{\sum_{n \in N(i)} |sim(i,n)|}, \tag{4.3}$$

where $N(i)$ denotes the set of k-nearest neighbors of i among the items that have been rated by active user a. In order to find $N(i)$ efficiently, the algorithm first constructs a l-nearest neighbor graph, which represents the l-nearest neighbor relationships between items. Then we can find the k number of neighbors based on this pre-computed l-NN graph instead of calculating the item-by-item similarity matrix when a recommendation is requested. In this equation, we use the adjusted cosine similarity as the similarity measure:

$$sim(i,n) = \frac{\sum_{u \in \mathcal{C}_u} (r_{u,i} - \overline{r_u})(r_{u,n} - \overline{r_u})}{\sqrt{\sum_{u \in \mathcal{C}_u} (r_{u,i} - \overline{r_u})^2} \sqrt{\sum_{u \in \mathcal{C}_u} (r_{u,n} - \overline{r_u})^2}}, \tag{4.4}$$

where $\mathcal{C}_u$ denotes the set of co-raters. $\overline{r_u}$ is the average rating of user u.

[50] and [51] indicate that CF algorithms produce movie recommendations of a high quality compared to baseline algorithms, which only recommend the most popular movies or highly rated movies. Although more efficient algorithms have been proposed, CF algorithms are still widely used in industry and academia due to their simplicity and acceptable levels of accuracy. However, there are several barriers preventing the realization of rapid recommendations when using existing approaches. First, it is necessary to find different neighbors depending on the active users. Specifically, the user-based CF algorithm finds the k-nearest users from among all users who have rated a certain unrated item i when predicting the rating of i, whereas the item-based CF algorithm finds the k-nearest items from among all items that have been rated by active user u. Second, it is necessary to predict all of the unrated items when a recommendation is re-

quested by a user. This procedure is somewhat inefficient in that according to [50] and [7], we usually need only the top-N recommendation results in real-world scenarios. While CF algorithms would provide rapid recommendations if there were not too many recommendation requests in a short time frame, it would not be easy for commercial recommender systems in which numerous recommendations are being requested by numerous users to provide real-time recommendations. Although there have been a few approaches to reduce the recommendation time, such as the work of [2], [57], and [56], either the performance gain is not significant or the approaches are not based on user-based/item-based collaborative filtering.

4.3 Fast Collaborative Filtering

Our approach consists of two main steps: first, we approximately construct a k'-nearest neighbor graph (k'-NN graph) as a preprocessing step based on our previous work of [30] and [25] (Section 4.3.1). Here, we usually set k' such that $l \gg k' > k$. Second, we find the k neighbors of unrated items based on the k'-NN graph. Then we recommend items to users using the k neighbors and our revised version of the non-normalized cosine neighborhood (Section 4.3.2).

4.3.1 Nearest Neighbor Graph Construction

The construction of a k'-NN graph is a task which involves finding the k' nodes most similar to each node. Although other tasks, such as *k-NN search* presented by [20] and [23], *reverse k-NN search* proposed by [55], *similarity join* introduced by [32] and *top-k similarity join* presented by [60] and [35], can be used for recommender systems, we use the k'-NN graph because it is one of the most appropriate data structure for our algorithm. One of the easiest ways to construct a k'-NN graph is to calculate the similarities between all of the nodes and extract the nodes most similar to each node. In spite of its simplicity, this brute-force approach requires quadratic time complexity, which is burdensome

Items \ Users	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	u_{10}
i_1	0.52		0.37	0.31				0.33	0.23	
i_2	0.73	0.55	0.1		0.37			0.05		
i_3	0.25	0.4				0.27	0.29			0.1
i_4	0.25	0.27	0.3	0.35	0.8					
i_5			0.48	0.32	0.37		0.34			0.2

Figure 4.1: Example of greedy filtering

when used in conjunction with large amounts of data. An alternative way to cope with this problem is to use inverted indices given the fact that item-by-user matrices are usually very sparse. However, according to [13], this approach is also not appropriate for handling large amounts of high-dimensional data.

Our main idea is to construct an approximate k'-NN graph based on *greedy filtering* presented by [30] and [25] in order to speed up this process. It is known that greedy filtering outperforms other k'-NN graph construction algorithms, such as *NN-Descent* proposed by [24] or *kNN-Overlap* presented by [12], for high-dimensional sparse datasets. If there is no decline in the quality of recommendations when we use approximate graphs, we do not have to spend much time on building an exact k'-NN graph. The accuracy of the k'-NN graph is defined as follows:

$$Accuracy = \frac{\#\ correct\ k'\text{-}nearest\ neighbors}{\#nodes \cdot k'} \tag{4.5}$$

Figure 4.1 shows an example of how greedy filtering constructs an approximate k'-NN graph. In this figure, there are five items and ten users; the values in the matrix indicate the ratings, each corresponding to its item and user. The main idea of greedy filtering is to filter item pairs whose "large value dimensions" (the shaded portions in the figure) do not overlap at all. In this figure, i_1 and i_2 share a common large value dimension. Hence, we calculate the similar-

ity between i_1 and i_2. In contrast, i_2 and i_4 do not have a common large value dimension; accordingly, we do not calculate the similarity between i_2 and i_4. [25] describe in detail the manner in which large value dimensions are selected for each item. In an actual implementation of this method, we use adjacency lists instead of adjacency matrices. The empirical time complexity of greedy filtering is $O(|\mathcal{I}|)$, where $\mathcal{I}$ is a set of items.

According to [25], this algorithm performs much better when we apply the TF-IDF weighting scheme and this process does not decrease the quality of recommendations significantly. Thus we adjust the values in the input matrix based on the TF-IDF weighting scheme:

$$M'_{i,j} = \left(0.5 + \frac{0.5 \cdot M_{i,j}}{max\{M_{i,k} : u_k \in \mathcal{U}\}}\right) * log\left(\frac{|\mathcal{U}|}{F(u_j)}\right), \tag{4.6}$$

where $M_{i,j}$ denotes the original value of the matrix corresponding to the i^{th} item and the j^{th} user; $\mathcal{U}$ denotes a set of users; $F(u_j)$ denotes the number of items that have values corresponding to the users u_j.

For example, suppose that we use the cosine similarity with the TF-IDF weighting scheme and that there are two items i_1 and i_2 in a dataset. In such a case, greedy filtering would calculate their level of similarity if they are highly rated by at least one certain inactive user. Otherwise, it would filter out those item pairs.

4.3.2 Fast Recommendation Algorithm

Recall that the two main drawbacks of user-based or item-based CF algorithms are that they have to find different neighbors depending on active users and that they have to predict all of the unrated items. Our novel algorithm, RCF, solves these problems. Let $B[i]$ be a k-NN list of item i, which was already calculated in Section 4.3.1. Let $\mathcal{I}_u$ and $\mathcal{I}_r$ be sets of unrated items and rated items of an active user a, respectively. Then RCF works as follows:

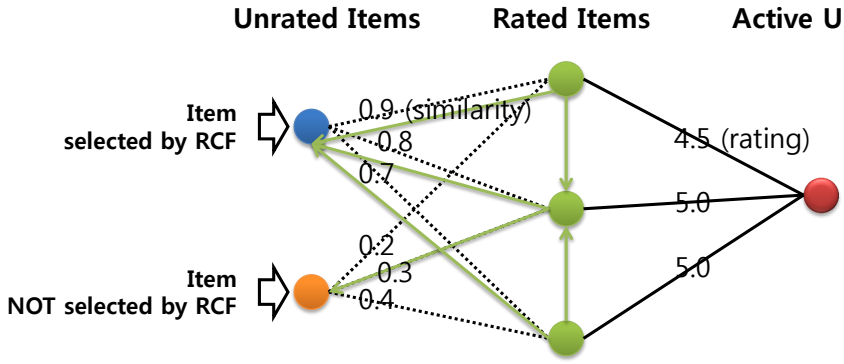


Figure 4.2: Example of reversed CF

1. For every item $i \in \mathcal{I}_u$, prepare an empty set $S[i]$.

2. For every item $i \in \mathcal{I}_r$ and every item such that $j \in \mathcal{I}_u$ and $j \in B[i]$, add i to $S[j]$.

3. For every item $i \in \mathcal{I}_u$, if $|S[i]| > k$, then delete all except for the most similar k_1 items from $S[i]$.

4. Then, predict the ratings for all items i such that $i \in \mathcal{I}_u$ and $|S[i]| = k$,

$$p_{a,i} = \overline{r_i} + \sum_{n \in S[i]} (r_{a,n} - \overline{r_n}) * sim(i,n) \tag{4.7}$$

Here, $sim(i,n)$ is the cosine similarity between i and n. It is defined as follows:

$$sim(i,n) = \frac{\sum_{c \in \mathcal{C}_u} r_{c,i} \cdot r_{c,n}}{\sqrt{\sum_{c \in \mathcal{C}_u} (r_{c,i})^2} \sqrt{\sum_{c \in \mathcal{C}_u} (r_{c,n})^2}} \tag{4.8}$$

If an unrated item does not have the list of k_1 number of items, RCF does not predict its rating.

Figure 4.2 shows an illustrative example of our approach. In this example, we set k and k' to 2 and 2, respectively. Thus we find 2-nearest neighbors for each rated item. An edge from a rated item i to an unrated item j indicates that j is one of the 2-nearest neighbors of i. The first unrated item has three incoming edges and the similarities between this item and the rated items are

0.9, 0.8 and 0.7 respectively. Because k is 2 in this example, we discard the edge labeled with 0.7, and predict the ratings of the item based on the remaining edges. On the other hand, the second unrated item has just one incoming edge so that we do not predict the rating of the item.

The intuition behind this algorithm is that if one of the nearest neighbors of rated item i is unrated item j, there would be a high probability that one of the nearest neighbors of unrated item j is rated item i. This is why the proposed algorithm is termed Reversed CF. One of the main characteristics of RCF is that it does not predict the preferences of all unrated items of a user. This approach does not sacrifice the level of recommendation quality for two reasons. First, if the rating of an unrated item is predicted by RCF, RCF and the item-based CF algorithm select the same neighbors for predicting the item in many cases. Second, if the rating of an unrated item is not predicted by RCF, the average similarity value of the k-nearest neighbors of the unrated item is usually lower than that of another item predicted by RCF, which is the case when it is difficult for the item-based CF algorithm to predict accurate ratings. In Section 4.4, we will discuss this in more detail.

Table 4.1: Summary of the recommendation algorithms

Algorithm	Phase	Task
UserCF	Preprocessing	Similarity matrix construction
		Significance weighting
	Recommendation	Selecting k users that have rated the item
		All rating predictions
ItemCF	Preprocessing	l-NN graph construction ($l \gg k$)
	Recommendation	Selecting k items that have been rated by an active user
		All rating predictions
RCF	Preprocessing	k'-NN graph construction ($k' > k$)
	Recommendation	Selecting k items
		Fewer rating predictions
RCF+TFIDF+GF	Preprocessing	k'-NN graph construction using GF ($k' > k$)
	Recommendation	Selecting k items
		Fewer rating predictions

4.4 Experiments

4.4.1 Experimental Setup

Dataset and Algorithms. We use the MovieLens dataset[1] for comparisons: there are $1,000,209$ ratings, $3,952$ movies, and $6,040$ users; each user rates at least 20 number of items; the rating scale ranges from 1 to 5 in which higher ratings indicate greater preference. We considered four types of algorithms for a comparison: UserCF implements the work by [1]; ItemCF implements the work by [2]; RCF implements only the fast recommendation algorithm presented in Section 3.2; RCF+TFIDF+GF implements the fast recommendation algorithm presented in both Section 4.3.1 and 4.3.2. We set the default parameters k, k', and l to 10, 20, and 300, respectively. Table 4.1 summarizes the abovementioned recommendation algorithms and their related parameters.

Quality Evaluation. We follow the testing methodology of a recommender system introduced by [50]. We divide the ratings into two groups. One group of data consisting of $986,206$ ratings (98.6% of ratings) is used for our training set, and the other group of data consisting of $14,003$ ratings (1.4% of the ratings) is

[1] http://grouplens.org/datasets/movielens/

used for the probe set. The test set consists of all of the five-star ratings (1,661 ratings) of 3,719 unpopular movies (99.65% of the movies) in the probe set. Then, for each rating of movie m rated by user u in the test set, we randomly select 1,000 movies unrated by u and recommend the top-N movies from among the 1,001 movies (the 1,000 items selected in addition to m); if we recommend m, we refer to this as a hit. Finally, we measure the degree of recall using the following equation:

$$recall = \frac{\#\ hits}{|test\ set|} \tag{4.9}$$

Performance Evaluation. We measure both the preprocessing time and the recommendation time for each algorithm. In UserCF, the preprocessing time is the overall time needed to construct the user-by-user similarity matrix plus the time for significance weighting. For ItemCF, we measure the l-nearest neighbor (l-NN) graph construction time as the preprocessing time. We use the inverted index-based method to calculate the l-NN graph, as it is one of the fastest algorithms for constructing an exact nearest neighbor graph. Similarly, the preprocessing time of RCF consists of only the time needed to construct the k'-NN graph; we construct this graph using inverted indices. The preprocessing time of RCF+TFIDF+GF is identical to that of RCF, except it uses greedy filtering to construct the k'-NN graph. The recommendation time is the total time to produce top-N recommendations for all 6,040 users, because according to [4], it is common to precompute all of the recommendation results in commercial systems.

4.4.2 Overall Comparison

Figure 4.3 shows the recall of the abovementioned algorithms while varying the number of recommended items. In this result, RCF outperforms both UserCF and ItemCF, which means that fewer rating predictions yield better results. When we apply the TF-IDF weighting scheme and use the approximate k'-

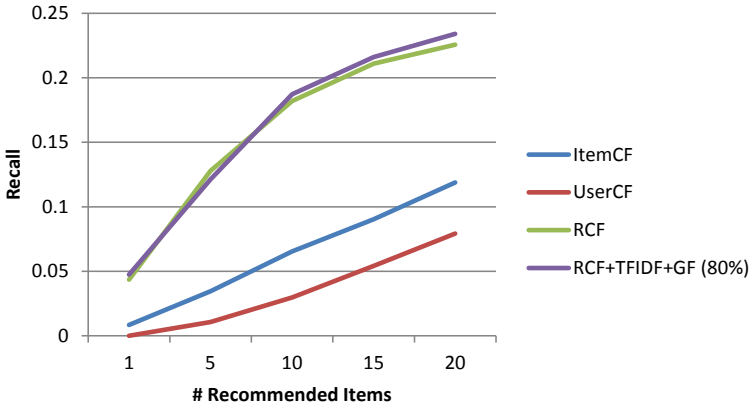


Figure 4.3: Comparison of all algorithms (recall)

NN graph with 80% accuracy instead of an exact k'-NN graph, the recall is decreased slightly, though this method still outperforms UserCF and ItemCF.

There are two main reasons why RCF outperforms ItemCF despite the fact that RCF simulates ItemCF. First, ItemCF usually predicts the ratings of unrated items based on fewer similar items. Second, rating predictions based on less similar items are less accurate than those based on similar items. Table 4.2 provides evidence of these assertions. First, we divided the items into two subsets, where one subset contains unrated items whose ratings are predicted by RCF and the other subset contains other unrated items. Then, for each subset, we measured the average similarity of selected neighbors, MAE, and RMSE after executing ItemCF. The average similarity value supports the first assertion, and MAE and RMSE support the second assertion. Note MAE and RMSE of user u are defined as follows:

$$MAE(u) = \frac{\sum_{i \in \mathcal{I}_S} |p_{u,i} - \hat{p}_{u,i}|}{|\mathcal{I}_S|} \tag{4.10}$$

$$RMSE(u) = \sqrt{\frac{\sum_{i \in \mathcal{I}_S} (p_{u,i} - \hat{p}_{u,i})^2}{|\mathcal{I}_S|}}, \tag{4.11}$$

where $\hat{p}_{u,i}$ denotes the predicted rating of item i by u, $p_{u,i}$ denotes its corresponding actual rating, and $\mathcal{I}_S$ denotes an item set, which can be a set of either

items selected by RCF or items not selected by RCF.

Table 4.2: Comparison of prediction accuracy with two different item sets

Item Set	Avg. Sim.	Avg. MAE	Avg. RMSE
Items selected by RCF	0.1954	0.6475	0.8361
Items NOT selected by RCF	0.1300	0.7353	0.9300

One limitation of RCF is that the algorithm cannot recommend many items if the parameter k' is not large enough. Because of this limitation, as shown in Figure 4.3, the recall of RCF and the RCF variant does not increase significantly when N is large enough. Although [2] and [7] state that we usually need only a small number of recommendations in real-world scenarios, if there is a need for a very large number of recommendations, the performance of RCF would be similar to that of ItemCF.

Figure 4.4 shows the pre-processing time and recommendation time of the abovementioned algorithms on a log scale: (1) UserCF is the slowest algorithm among these four algorithms. As a preprocessing step, this algorithm constructs a user-by-user similarity matrix and applies the significance weighting to the similarity matrix, which takes quadratic time complexity in total. It also consumes a considerable amount of recommendation time when a query is requested, because for each unrated item, it selects k users who have rated the item and predicts the rating of the item. (2) ItemCF is faster than UserCF in that it does not need to calculate a similarity matrix or complete the significance weighting step. Instead, it constructs a l-NN graph in which l greatly exceeds k. Although l is a large constant, we reduce the time to construct the graph using inverted index join, which is one of the fastest algorithms among all exact k-NN graph construction algorithms. (3) While the preprocessing time of RCF is similar to that of ItemCF, this algorithm significantly outperforms ItemCF in terms of recommendation time for two reasons. First, it does not take much time to select the neighbors of each unrated item in that it only checks the set size of each unrated item and then deletes all except for the most

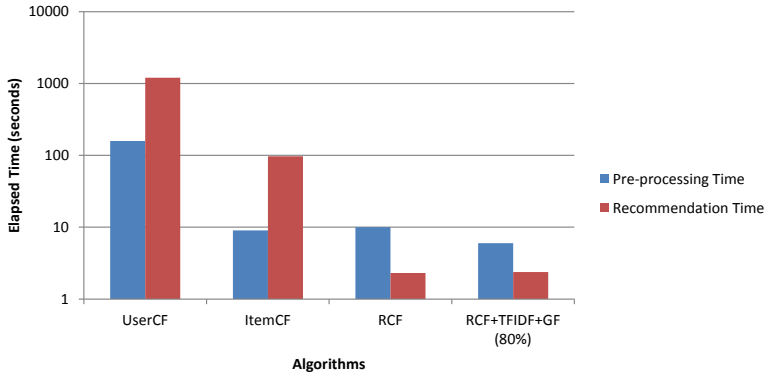


Figure 4.4: Comparison of all algorithms (elapsed time)

similar k items. Second, it calculates fewer item ratings, which dramatically decreases the recommendation time. (4) RCF+TFIDF+GF is the fastest algorithm among these four algorithms. While its recommendation time is similar to that of RCF, it outperforms RCF in terms of preprocessing time, as it constructs an approximate k'-NN graph by means of greedy filtering. In Section 4.4.3, we demonstrate even faster recommendations by changing the greedy filtering parameters.

4.4.3 Effects of Parameter Changes

We identified several important factors that affect the quality and performance of the algorithms: the parameters k,k',l, and the k'-NN graph accuracy. Because the parameters k and l were analyzed in the work of [1] and [2], we only analyze k' and the k'-NN graph accuracy in this paper. Figure 4.5 shows the recall of RCF variants with different parameter k', varying the number of recommended items. Note k' is directly related to the number of rating predictions performed by RCF. When we increase the parameter from 10 to 20 or from 20 to 30, the recommendation quality is improved because we can consider more items for the top-N recommendation. However, when we increase the parameters from 30 to 70, the recommendation quality is not improved for the reasons given in the previous subsection. Figure 4.6(a) and 4.6(b) show that the parameter k' is also

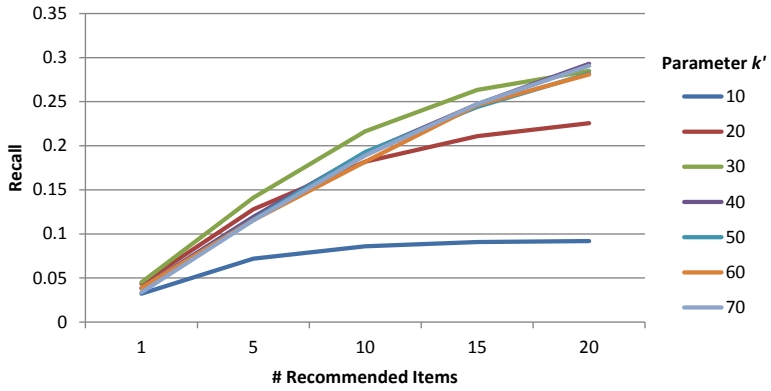
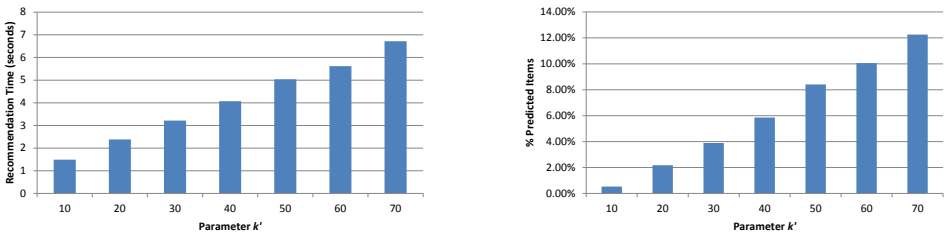


Figure 4.5: Recall of RCF variants with different k' parameters

related to the recommendation time and the percentage of rating predictions, respectively. Because we can improve the execution time by setting k' to a low value, it would be desirable to set k' to 20 or 30.



(a) Elapsed time

(b) Percentage of rating predictions

Figure 4.6: Effect of different k' parameters

Similarly, Figure 4.7 and Figure 4.8 show the recall and pre-processing time of RCF variants with different graph accuracy levels, varying the number of recommended items. There are two interesting findings in these figures: first, the recall is the highest when k'-NN graph accuracy is 70%. We can see this result because we cannot guarantee that we will always prefer the items more similar to the preferred items. Similar results are shown in the work of [12], where an approximate k-NN graph is used for fast agglomerative clustering. Second, the elapsed time of RCF is the highest when k'-NN graph accuracy is 90%, because

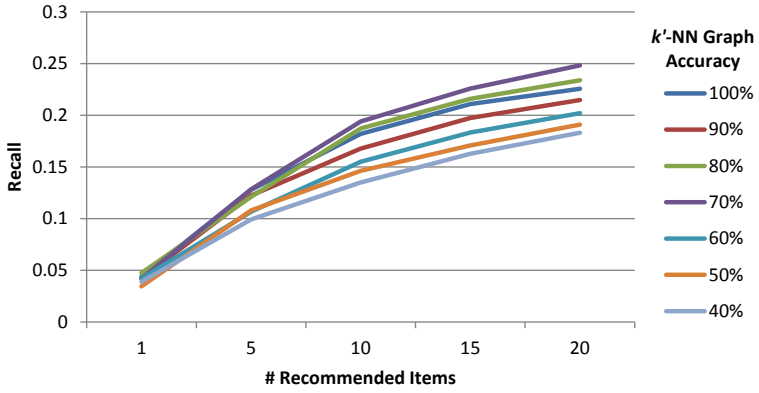


Figure 4.7: Recall of RCF variants with different k'-NN graph accuracy levels

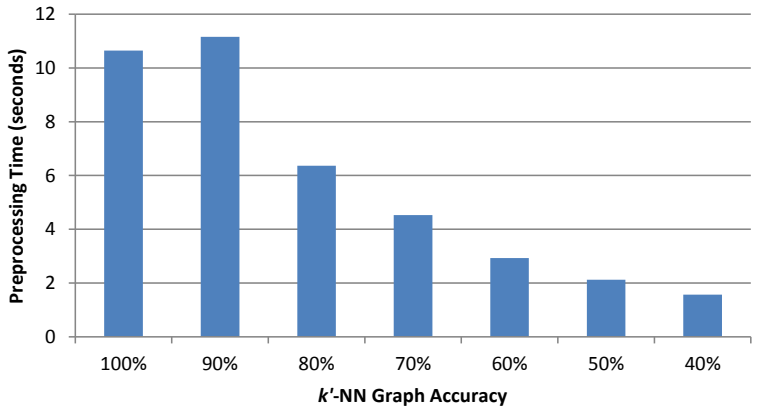


Figure 4.8: Elapsed time of RCF variances with different graph accuracy levels

we use inverted index join instead of greedy filtering when we construct the exact k'-NN graph. Generally, however, the quality of recommendations slightly drops off when we decrease the graph accuracy, whereas the pre-processing time is significantly reduced. We can infer that these RCF variants perform even better in terms of preprocessing time as the number of nodes or dimensions scales up due to the scalability gained when using greedy filtering.

4.5 Summary

This chapter presents RCF, a fast CF algorithm which utilizes a k'-NN graph. Not only does this algorithm perform fewer predictions while filtering out inaccurate results, but it also supports the rapid retrieval of similar users. The experimental results show that our approach outperforms traditional user-based/item-based CF algorithms in terms of both preprocessing time and query processing time without sacrificing the level of accuracy when we set k and k' to 10 and 20, respectively. While much of the recent work, such as [56] and [52], focuses on improving the recommendation quality, the main aim of our approach is to reduce the elapsed time required for recommendation.

The limitations of our approach are twofold: first, RCF is not appropriate for the case where we have to predict the ratings for all of the unrated items. In future work, we would like to present a novel algorithm for coping with this problem. Second, the performance of greedy filtering depends on the dataset so that the algorithm could be slower than inverted index join in the worst case. Thus we are currently developing a novel k'-NN graph construction algorithm that always guarantees high level of quality and performance.

Chapter 5

Fast Approximate k-NN Search

k-Nearest Neighbor (k-NN) search aims at finding k points nearest to a query point in a given dataset. k-NN search is important in various applications, but it becomes extremely expensive in high dimensional space with a number of data points. In response to this performance issue, *locality-sensitive hashing* (LSH) is suggested as a method of probabilistic dimension reduction while preserving the relative distances between points. Through experiments with various feature extraction methods, we observed that none of the existing LSH-based k-NN search methods showed consistent performance superiority, each exhibiting poor performance in some of the datasets.

In this chapter, we target on generating k-NN search results efficiently regardless of properties of a given dataset. In this regard, we present a novel algorithm called *Signature Selection LSH* (S2LSH), where we select query-specific signatures from a signature pool to pick high-quality k-NN candidates. First, we construct a highly diversified signature pool consisting of various signatures. The signatures are generated based on a data-dependent LSH algorithm to capture the global topological features specific to the given dataset. Then, for a given query point, we select multiple query-specific signatures from the signa-

ture pool in order to find high-quality k-NN candidates of the query vector. We also incorporate three additional optimization techniques to further improve the performance of S2LSH in a bulk execution setting such as k-NN graph construction. Extensive experiments show that our approach consistently outperforms the state-of-the-art LSH algorithms across various types of datasets. Furthermore, our approach in a bulk execution setting is comparable to or faster than the algorithms carefully designed for efficient k-NN graph construction.

5.1 Introduction

Typically, there are three types of k-NN computation tasks: k-NN computation for a single query (k-NN search), for every object in the database (k-NN graph construction), and for some of the objects in the database (partial k-NN graph construction). For example, the k-NN search is an essential part of Google image search and k-NN classification; k-NN graph construction is important in the YouTube video recommendation system and agglomerative clustering; partial k-NN graph construction can be used for incremental k-NN graph construction. An interesting thing is that although the k-NN graph construction can be implemented by the iterative executions of k-NN search, the k-NN search algorithms do not perform as well for k-NN graph construction (and vice versa). It is because they do not reuse the information that can be obtained from the k-NN computations of the other objects. Therefore, in cases where we need two or three types of k-NN computation tasks (e.g., a search engine that supports both search and similarity browsing), we have to find an effective algorithm for each task. If there is no such algorithm, then we have to use brute-force approach instead. Another interesting thing is that even using the same raw multimedia data, the performance of existing approaches significantly varies depending on the types of feature extraction methods being used. Thus in cases where we need two or more types of feature extraction methods (e.g., a search engine that uses facial features for facial images and global features for the other images),

we have to find an efficient algorithm for each feature extraction method if any.

In this paper, we present a novel algorithm, called *signature selection LSH* (S2LSH). The main contributions of this paper can be summarized as follows:

- We present a novel k-NN computation algorithm where we select query-specific signatures from a signature pool to pick high-quality k-NN candidates (Section 5.2 and 5.3).
- We incorporate three additional optimization techniques to further improve the performance of S2LSH in a bulk execution setting such as k-NN graph construction (Section 5.2.4 and 5.3.3).
- Through the extensive experiments, we show that our approach consistently outperforms the state-of-the-art algorithms across various types of k-NN computation tasks and datasets (Section 5.5).

5.2 Signature Selection LSH

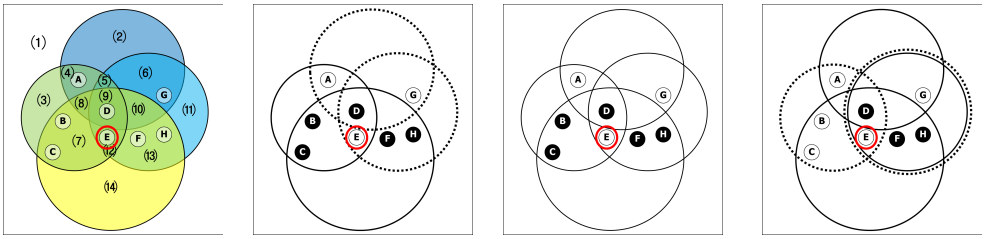
Let V be a set of vectors in a d-dimensional space. We refer to the vectors in V as *data vectors* in this paper. Without loss of generality, we assume Euclidean distance as a distance measure. Then we define approximate k-NN search as a process of finding approximate k-nearest neighbors among the vectors in V for a query vector q in a d-dimensional space.

Our approach consists of four main steps: 1) data-dependent locality sensitive hashing, 2) signature pool generation, 3) signature selection, and 4) finding k-nearest neighbors based on signature selection. The fourth step is highly coupled with the third step so that we will describe them in one subsection.

5.2.1 Data-dependent LSH

We assume that we generate the H number of LSH hash functions. Then the signature of length H for vector v can be generated by the following equation:

$$s(v) = \langle h_1(v), h_2(v), ..., h_H(v) \rangle \tag{5.1}$$



(a) Spherical Hashing (b) E2LSH+ (c) C2LSH+ (d) S2LSH

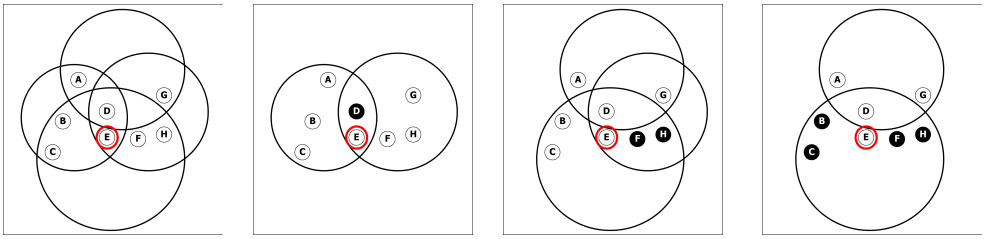
Figure 5.1: An illustrative example of S2LSH. In order to find the 2-NN of **E**, namely **D** and **F**, S2LSH only selects three candidates whereas E2LSH+ and C2LSH+ selects five candidates.

Here, $h_i(v)$ denotes the i^{th} hash function as described in Chapter 2. If the functions are based on random projections, they are determined by the selected a and b.

Although random projections like data-independent LSH schemes are widely used in industry and academia, their performance significantly varies depending on datasets in that they do not consider their data distributions. In recent years, there have been proposed various types of data-dependent LSH techniques, such as spectral hashing [42], anchor graph hashing [18], and spherical hashing [9]. Let $d(\cdot, \cdot)$ be the distance between two vectors. Then the data-dependent LSH aims at satisfying the following properties:

- $d(v_1, v_2) < d(v_1, v_3)$ if and only if $d(s(v_1), s(v_2)) < d(s(v_1), s(v_3))$
- For all i and j such that $i \neq j$ and $1 \leq i, j \leq H$, $\sum_v h_i(v)h_j(v) = 0$.
- $\sum_v s(v) = 0$.

Satisfying the first property is the primary purpose of all of the LSH algorithms. The second and third properties are important because the large amounts of distance information should be represented by compact signatures: they indicates the the independence between hashing functions and the balanced partitioning of vectors for each hash function, respectively.



(a) Diagram 1 (b) Diagram 2 (c) Diagram 3 (d) Diagram 4

Figure 5.2: An illustrative example of signature pool generation. Based on spherical hashing, hash functions are represented by spheres, and each signature is represented by a region.

As far as we know, spherical hashing is one of the most efficient data-dependent LSH techniques. Conceptually, it draws H number of spheres for a small number of training samples such that 1) the spheres are not too close or too far apart for the second property, and 1) each sphere contains about a half of the training samples for the third property. Then if a vector v is inside the i^{th} circle, the hash function h_i maps v to $+1$ (and -1 otherwise). Figure 1(a) shows an example of spherical hashing: in this figure, the signature of vector **A** is represented by $(+1, +1, -1, -1)$, and the signature of vector **B** is represented by $(-1, +1, -1, +1)$. In the rest of this paper, we use spherical hashing as our LSH scheme. Because it is a binary code embedding technique, we also assume that we use binary signatures. However, our approach is also effective when using other types of binary hashing methods, because we do not use any features that are dependent on spherical hashing. In Section 5.5.3, we will discuss this issue in more detail.

5.2.2 Signature Pool Generation

We now have the H hash functions so that we can generate the signature of length H using equation (5.1). However, it is not easy to find k-nearest neighbors efficiently using these signatures because the first property of data-dependent LSH described in Section 5.2.1 does not hold in many cases. Figure

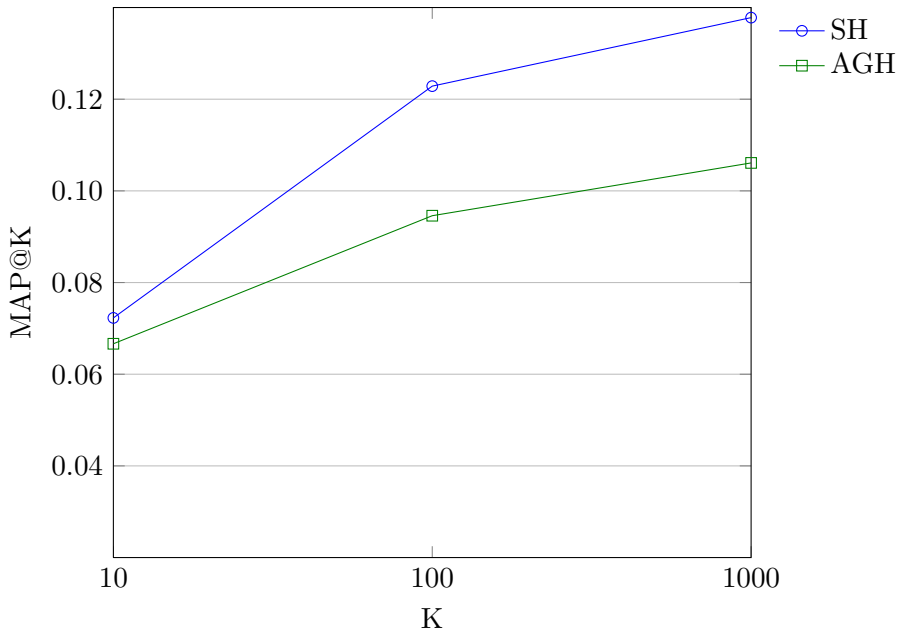


Figure 5.3: Mean Average Precision for spherical hashing (SH) and anchor graph hashing (AGH) based on 100-bit signatures in the 500D NUS-WIDE dataset

5.3 shows that MAP@10, MAP@100 and MAP@1000 of spherical hashing are not higher than 0.15 in the 500-dimensional NUS-WIDE dataset. Our solution is to generate a huge number of different signatures based on the H functions and then select the most effective ones for a given query vector. The intuition behind this solution is that for each query vector v, there would be a set of signatures more effective than $s(v)$.

In this subsection, we focus on generating a pool of signatures for every vector based on the H number of hash functions. The process is as follows: first, given integers m_1 and m_2 ($1 \leq m_1 \leq m_2 \leq H$), we set M to a random integer ranged from m_1 and m_2. Second, we randomly generate M integers r_1, r_2, ..., r_M each ranged from 1 and H, and generate the signature for every vector v as follows:

$$s_i(v) = \langle h_{r_1}(v), h_{r_2}(v), ..., h_{r_M}(v) \rangle \tag{5.2}$$

Here, h_i denotes the i^{th} hash function, where i is the iteration number. Given

an integer L ($1 \leq L$), we repeat the above process L number of times. Then the signature pool of vector v consists of L-dimensional vector as follows:

$$P(v) = \langle s_1(v), s_2(v), ..., s_L(v) \rangle \tag{5.3}$$

Note signatures with various lengths will be generated and each hash function is selected with equal probability. This process aims to make the signatures as diverse as possible because we do not know the query vectors in advance.

Conceptually, this process can be regarded as drawing multiple Venn Diagrams as shown in Figure 5.2: hash functions are represented by spheres, and each signature is represented by a *region*, which is a distinct area determined by the intersection of spheres in this figure. In this example, each vector has a pool of 4 signatures. For query vector **E**, there is no vector whose first signature is the same as that of **E** (as shown in Figure 5.2(a)), whereas there are two vectors **F** and **H** that have the same third signatures as that of **E** (as shown in Figure 5.2(c)).

Recall there are four parameters in generating a pool of signatures, H, L, m_1 and m_2. All of the parameters control the diversity of signatures. Obviously, there is a tradeoff between diversity (quality) and cost: 1) if we set H to a large constant, we can make various types of signatures by diversifying the types of hash functions while it takes much time to make hash functions. In our experimental settings, we set $H = 300$ or 1000. The optimal parameter setting of H highly depends on the LSH scheme and dataset being used. 2) If we set L to a large integer, we can also make various types of signatures using different combinations of hash functions while it consumes much memory to store signature pools. In our experimental settings, we set $L = H/2$, because if H is a very small integer (e.g., 1), it is hard to make various types of signatures even when L is a large integer. 3) If we set m_1 to a small integer or m_2 to a large integer, we can generate various types of signatures with different lengths while also generating many inefficient signatures. In order to find the near-optimal

parameters of m_1 and m_2, we can use the second and third properties of data-dependent LSH discussed in Section 5.2.1. First, we know that each region of Figure 2(a) contains approximately an equal number of vectors. In the ideal case, each region contains $N/2^M$ number of vectors. Here, N is the number of vectors. Assuming that $N = 100,000$, each region contains approximately $3,000$ vectors and 3 vectors when $M = 5$ and $M = 15$ respectively. In our experimental settings, we set $m_1 = 5$ and $m_2 = 15$, because if m_1 is lower than 5 or m_2 is higher than 15, there would be a too small or too large number of collisions.

5.2.3 Signature Selection

For a given vector v, our aim is to select the most effective signatures from $P(v)$. Assume that we will calculate the similarities between v and other vectors u such that u has the same signature as that of v. If there are many vectors u that are k-nearest neighbors of v, then the signature is effective in finding nearest neighbors for v. On the other hand, it will waste a huge amount of time if there are many vectors u that are not k-nearest neighbors of v. Thus we define the effectiveness E of a signature as follows:

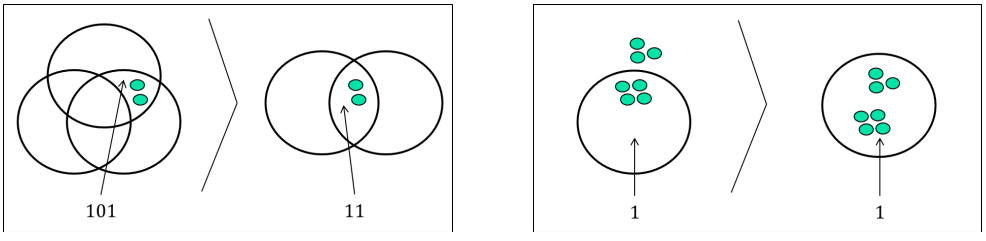
$$E_{v,i}(s \in S) = \frac{|kNN(v) \cap c_i(s)|}{|c_i(s)| - 1} \tag{5.4}$$

Here, S denotes a set of all generated signatures. $kNN(v)$ is a set of exact k-NN of v. $c_i(s \in S)$ is a set of vectors u such that $s_i(u) = s$.

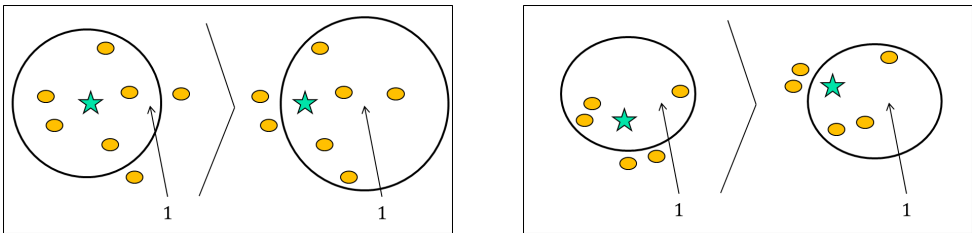
However, we do not know in advance the exact k-NN of v. In the following sections, we present a way to estimate the effectiveness of each signature based on feature selection.

Feature Selection

We have considered the eight types of features for k-NN search as follows: *signature length*, the number of *collisions*, the number of one (or zero) hash values, average radius of spheres that contain (or do not contain) a vector, and



(a) Signature length (b) # Collisions



(c) Maximum Distance (d) Average Distance

Figure 5.4: Four types of features for signature selection

average distance to the centers of spheres that contain (or do not contain) a vector. Here, signature length indicates the number of dimensions in a signature. For example, the signature length of "101" is 3 and that of "11" is 2. If two vectors have the same signature, we say there is a *collision* between them.

Signature length. Long signatures would indicate relatively small regions. Figure 5.4(a) shows an example of our intuition: the left signature has longer signature length so that its corresponding region is relatively small. Because the small region indicates that the vectors in the region are close to each other, there would be a high probability that long signatures have high effectiveness. Heo et al. [9] also show that the popular LSH algorithms achieve higher mean average precision when using longer binary codes.

The number of collisions. Signatures with a small number of collisions would have high effectiveness. For example, Figure 5.4(b) shows that the vectors of the left figure are more close to each other than those of the right figure even though they have the same signature length.

The number of one (or zero) hash values. Signatures with many positive hash values would have high effectiveness. If two vectors have positive hash values in common for a hash function, then their distance is at most the radius of the corresponding sphere. On the other hand, two vectors with negative hash values in common does not always indicate that they are close to each other. For example, if they do not have any positive hash values, all of the hyperspheres could split them in the worst case.

Average radius of spheres that contain (or do not contain) a vector. If a vector is inside a sphere and the radius of the sphere is small, then the region containing the vector is small. Likewise, if a vector is outside a sphere and the radius of the sphere is large, then the region containing the vector would be small.

Average distance to the centers of spheres that contain (or do not contain) a vector. If a vector is close to the center of the sphere that contains the vector, then the distance to other vectors would be short. Likewise, if a vector is far from the center of the sphere that does not contain the vector, then the distance to other vectors would be short. Note when making the signatures of all of the vectors, we calculate the distances from all of the vectors to all of the centers of the spheres.

We observed that the first two features are very effective in all of our seven types of datasets while the last six features are not always highly correlated with the effectiveness. Thus in our algorithms, we only use the first two features.

Signature Selection

Our observation is that the longer the signature and the smaller the number of collision counts, the greater the effectiveness. Based on our observation, we can

estimate the effectiveness of a signature as follows:

$$E'_{v,i}(s \in S) = \begin{cases} 0 & \text{if } |c_i(s)| = 1 \\ \frac{len(s)}{|c_i(s)-1|} & \text{otherwise} \end{cases} \tag{5.5}$$

Here, len(s) denotes the signature length of signature s.

Given a parameter μ, our aim is to select the most effective signatures such that query vector q has approximately μ number of candidates. Note if we select the i^{th} signature for q, we will select all other vectors u that have the same i^{th} signatures as candidates. The process of our signature selection consists of the following three steps:

1. Select the most effective signature of q among the unselected signatures.
2. Find all other vectors u such that $s_i(q) = s_i(u)$ and select them as candidates of q.
3. If q has equal to or more than μ number of candidates, then the algorithm terminates. Otherwise, repeat the whole process.

The remaining process for k-NN search is to calculate the distances between the candidates and q and find the k closest vectors among the candidates.

For example, assume that we have a pool of signatures described in Figure 5.2. Our aim is to select the 2-NN of **E**, namely **D** and **F**. In diagram 1, $E'_{q,1}(\mathbf{E}) = 0$ because there is no vector in the same region. In diagrams 2, 3 and 4, $E'_{q,2}(\mathbf{E}) = 1/1$, $E'_{q,3}(\mathbf{E}) = 1/2$, and $E'_{q,4}(\mathbf{E}) = 1/4$ respectively. Thus the most effective signature of **E** is the second signature of **E**, and **D** is selected as a candidate. If we set the parameter μ to 2, then μ is larger than the number of candidates so that we iterate this process. At the second iteration, we select the third signature in which there are one k-nearest neighbor among the two candidates. Now μ is larger than the number of candidates, and we terminate the signature selection process. Figure 5.1 shows that in this example, S2LSH

selects fewer candidates compared with E2LSH+ and C2LSH+ while finding the 2-nearest neighbors.

5.2.4 Optimization Techniques

In order to find the same signatures efficiently, we use bucket hashing that was applied to the E2LSH package[1]. First, we build the L number of hash tables. Then we define L number of hash functions b_1 such that the i^{th} hash function maps the i^{th} signatures into the specific positions of the i^{th} hash table. When signatures could map to the same positions even though they are not same, we can distinguish them through making different buckets for each signature. Thus we also define L number of hash functions b_2 such that the i^{th} hash function maps the i^{th} signatures into the short hash codes, which corresponds to bucket IDs of the i^{th} table. The hash functions b_1 and b_2 are defined as follows:

$$b_1(s_i(v)) = \left(\left(\sum_{j=1}^{M} r_j^{'} h_{r_j} \right) \mod 2^{32} - 5 \right) \mod \mathrm{T} \tag{5.6}$$

$$b_2(s_i(v)) = \left(\left(\sum_{j=1}^{M} r_j^{'} h_{r_j} \right) \mod 2^{32} - 5 \right) \tag{5.7}$$

Here, T is the size of the hash table. We set $T = 10,000$ in our experiments.

Eliminating the duplicates is another important technique in our algorithm, because we can select a number of signatures which can yield many duplicate candidates. However, this problem can be easily solved by maintaining an array of length $|V|$ and marking which vectors were calculated already. For k-NN graph construction, we need a more advanced way to deal with it. See Section 5.3.3 for more detail.

[1] http://www.mit.edu/ andoni/LSH/

5.3 S2LSH for Graph Construction

5.3.1 Feature Selection

In this section, we propose our signature selection algorithm for k-NN graph construction, called S2LSH-M (S2LSH in a bulk execution setting). Basically, S2LSH-M executes the S2LSH algorithm the $|V|$ number of times. In addition, S2LSH-M considers the maximum, average, and minimum distances between two vectors as additional features:

Maximum/average/minimum distance. If the maximum/average/minimum distance between query vector q and the other vectors that have the same i^{th} signature is small, then the signature would have high effectiveness.

The maximum and average distances approximately indicate the distance from the query vector to the boundary of the region. On the other hand, the minimum distance indicates the possibility of the existence of k-nearest neighbors. Our observation is that the maximum/average distances are more effective in finding k-nearest neighbors than minimum distance. Thus we use the the maximum/average distances in our experiments.

5.3.2 Signature Selection

Note the distance features are different from signature length and the number of collision counts in that they are personalized features: that is to say, two vectors with the same signature have different feature values. The personalized features are more effective than the non-personalized features in general. However, we need a significant amount of time to calculate the feature values. One way to cope with this problem is to sample a subset of training data to estimate the feature value. Although this solution is effective when there is a small number of signatures, this could make the algorithm even slower when there is a large number of signatures.

Our idea is to reuse the distance information that can be obtained through

the search task of the other vectors: when finding k-nearest neighbors of v, we check whether the distance between v and other vectors u are calculated beforehand. In our implementation, we do not store all of the previous distances between vectors because they consume a significant amount of memory. Instead, we only keep the maximum and average distances for each pair of a vector and a signature. Now the effectiveness of a signature can be defined as follows:

$$E''_{v,i}(s \in S) = 1/(\max_{u \in c_i(s)}(d'(u,v)) + \operatorname{avg}_{u \in c_i(s)}(d'(u,v))) \tag{5.8}$$

$$d'(u,v) = \begin{cases} d(u,v) & \text{if precomputed} \\ \infty & \text{otherwise} \end{cases} \tag{5.9}$$

Note if all of the distances from query vector q and the other vectors with the same i^{th} signatures are not precomputed at all, then we use the features of S2LSH, namely signature length and the collision counts.

5.3.3 Optimization Techniques

For S2LSH-M, we present two types of optimization techniques: the first technique is for eliminating duplicate calculations, and the second technique is for refining an approximate k-NN graph. In practice, we cannot use the duplicate elimination technique that was used in S2LSH because of memory limitation. For example, if we allocate memory for an $|V| * |V|$ matrix that stores all the computed distances to avoid duplicate calculations, then we need to allocate 4TB of memory, assuming that there are 1M vectors and each element consumes 4 bytes. In order to alleviate this problem, recursive Lanczos bisection (RLB) [12] uses a hash table to store the computed distances: in this solution, the i^{th} vector hashes to the i^{th} position of the hash table. However, this solution allocates about 200GB of memory if there are 1M vectors, each element consume 4 bytes, and there are only 10% of distance calculations. For this reason, the duplicate elimination technique has not been applied to other algorithms, such as the Zhang's approach [29] and greedy filtering [25].

Our simple algorithm removes all of the duplicate calculations while only requiring $O(|V|)$ amount of memory. Assume that there is one bucket for each signature and that vectors with the same signature reside in the same bucket. For each vector v, it performs the following process:

1. Prepare a false-initialized array A of size $|V|$.
2. For each bucket B that contains v, calculate the distance between v and other vectors u such that they reside in the same bucket and $A[u] = false$, and then set $A[u]$ to $false$ and remove v from B.

Our second optimization technique is for increasing the accuracy of an approximate k-NN graph. We slightly modified the widely used technique, called *neighborhood propagation* [12, 24, 29, 27] as follows: for each vector v, we calculate the similarities between v and its 2-hop neighbors and 3-hop neighbors. And then we update the k-nearest neighbor list of v. Even though we check the 3-hop neighbors, relatively only a small number of distance calculations are needed: we define the scan rate as the number of distance calculations of an algorithm divided by the number of distance calculations performed by brute-force approach. If the number of vectors is $100,000$ and we refine a 10-NN graph, the scan rate can be increased at most by 0.022 even without eliminating duplicate calculations.

5.4 Theoretical Analysis

Formally, S2LSH is an approximate algorithm for a *k-NN cover problem*, which is defined as follows:

Definition 1 (k-NN Cover) Let $\mathcal{U}$ be a set of k-nearest neighbors $n_1, n_2, ..., n_k$ and $\mathcal{S}$ be a set of L candidate sets $\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_L$. A k-NN cover is a collection of candidate sets from $\mathcal{S}$ satisfied that every k-nearest neighbor in $\mathcal{U}$ belongs to at least one of the candidate sets. The cost of a k-NN cover is the sum of

the costs of all of the candidate sets in the collection of selected candidate sets. Then k-NN cover returns the collection of subsets that minimizes the cost.

The k-NN cover problem is NP-hard, since the set cover problem is reducible to the k-NN cover problem in a polynomial time. Hence, this indicates that we need an approximate algorithm for the fast retrieval of k-nearest neighbors.

If we already know the k-NN of a query, then we can use the greedy set cover algorithm to solve the k-NN cover problem. The greedy set cover algorithm repeatedly picks a candidate set that minimizes the cost. According to [61], it is a $ln(k)$-approximate algorithm for the set cover problem. Thus this indicates that a good approximate algorithm can provide high quality results.

5.5 Experiments

5.5.1 Experimental Setup

In this section, we compare our approach with the state-of-the-art algorithms in different k-NN computation tasks. For the experiments, we use various types of datasets to which different types of feature extraction methods were applied.

k-NN computation tasks. We have considered three types of k-NN computation tasks: k-NN search, k-NN graph construction and partial k-NN graph construction. For partial k-NN graph construction, we randomly select 20% of vectors from the vectors in V.

Algorithms. For k-NN search, we compare S2LSH to E2LSH and C2LSH. However, because E2LSH and C2LSH are not faster than the brute-force approach in many cases, we enhanced their performance by applying spherical hashing instead of random projections. We will call their optimized versions as E2LSH+ and C2LSH+, respectively. We do not consider LSB-tree as a candidate, because it is outperformed by C2LSH [23]. For k-NN graph construction, we select the recursive Lanczos bisection (RLB) and NN-Descent (NND) as rep-

Dataset	$\|V\|$	d	Feature
Corel	300,000	14	Lv et al. [43]
NUS-WIDE (CH)	200,000	64	Color Histogram [44]
Audio	50,000	192	Marsyas [45]
NUS-WIDE (BoW)	100,000	500	SIFT [46]
Shape	25,000	544	SHD [47]
MNIST	60,000	784	Pixel [48]
GIST1M	100,000	960	GIST [49]

Table 5.1: Dataset summary

resentatives of hyperplane-based and heuristic-based algorithms, respectively. However, NND does not achieve the high level of accuracy for some datasets, because it is a heuristic-based approach. For example, it does not achieve the accuracy of 90% for NUS-WIDE (BoW) dataset. Thus we enhance NND to NND+, which iteratively execute the NND until achieving the accuracy of at least 90%.

We do not consider the clustering-based algorithms in our experiments because in our preliminary experiments, they are outperformed by NN-Descent or they show the inconsistent performance depending on input parameters.

Evaluation Measures. For k-NN search, we measure pre-processing time and k-NN search time (query processing time) for every algorithm. The pre-processing time consists of the time for signature generation and the time required for generating a pool of signatures. The k-NN search time is measured by averaging over 1000 sample queries, and it does not comprise the time for the preprocessing. For k-NN graph construction, we measure the total elapsed time except for the data matrix construction time. Note as shown in Figure 5.5, the task of similarity calculations takes most of the time.

The accuracy (quality) of the result is calculated as the following formula:

$$Accuracy = \frac{\#\ correct\ k\text{-}nearest\ neighbors}{\#\ queries * k} \tag{5.10}$$

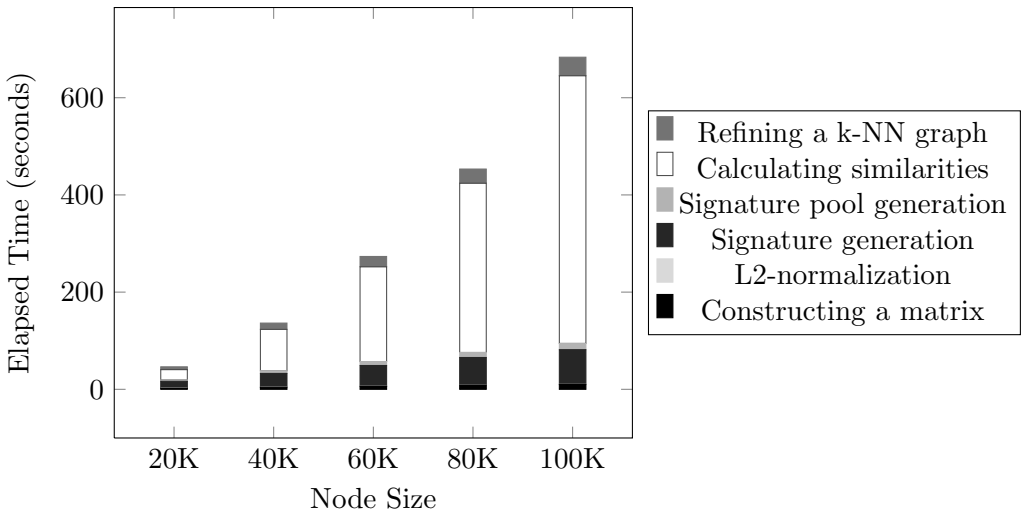


Figure 5.5: Elapsed time for each task (S2LSH, k-NN graph construction, and the NUS-WIDE dataset (BoW))

Algorithm	**Parameters**	
	NUS-WIDE-CH	NUS-WIDE-BoW
E2LSH+	$H = 100$, $K = 8$	$H = 1000$, $K = 10$
C2LSH+	$H = 20$, $l = 15$	$H = 100$, $l = 60$
RLB	$\alpha = 0.07$	$\alpha = 0.4$
NND+	$\rho = 0.3, \delta = 0.001$, $t = 0.9$	$\rho = 1.0, \delta = 0.001$, $t = 0.9$
S2LSH	$H = 100$, $L = 50$, $m_1 = 5$, $m_2 = 15$	$H = 1000$, $L = 500$, $m_1 = 5$, $m_2 = 15$

Table 5.2: Our parameter settings of all algorithms in the NUS-WIDE datasets.

Datasets. We use seven types of datasets for comparisons. They are represented by various types of feature vectors and different number of dimensions. Table 5.1 shows the summary of our datasets.

Algorithm	Average Accuracy					
	k-NNS		k-NNG		Pk-NNG	
	CH	BoW	CH	BoW	CH	BoW
E2LSH+	0.93	0.85	0.94	0.85	0.94	0.85
C2LSH+	0.92	0.91	0.92	0.91	0.93	0.92
RLB	N/A	N/A	0.89	0.85	0.89	0.89
NND+	N/A	N/A	0.93	0.91	0.93	0.90
S2LSH	0.92	0.91	0.92	0.91	0.91	0.92

Table 5.3: Average accuracy of the five executions (from 10K to 50K vectors) in the NUS-WIDE datasets. We set the parameters to achieve the similar level of accuracy.

Algorithm	Preprocessing Time (sec.)									
	NUS-WIDE-CH					NUS-WIDE-BoW				
	10K	20K	30K	40K	50K	10K	20K	30K	40K	50K
E2LSH+	1.48	1.86	2.41	2.69	3.16	189.41	230.19	258.82	270.52	302.20
C2LSH+	0.21	0.30	0.41	0.51	0.60	7.24	9.77	12.2	15.09	18.08
S2LSH	1.74	2.45	3.05	3.71	4.32	197.02	227.82	265.02	293.84	310.32

Table 5.4: Comparison of the k-NN search algorithms in terms of pre-processing time. The pre-processing time depends on parameter settings.

5.5.2 Experimental Results

First, we compare all of the algorithms using the two NUS-WIDE datasets. Table 5.2 represents our selected parameters in which all of the algorithms show their best performance and achieve the similar level of accuracy shown in Table 5.3. Note the parameter H of C2LSH+ is much lower than those of E2LSH+ and S2LSH because if there are many hash functions, the process of collision counting could be very slow. In other words, C2LSH+ needs a LSH algorithm that can represent the original vectors as very compact hash codes. We newly define the parameter t of NND+, which indicates the minimum accuracy that should be achieved. Because this parameter can be used only when we already calculated the answer set, the implementation of NND+ is not feasible in the real world. We do not specify the parameter μ used by S2LSH, because they are dependent on the number of data vectors.

Because every algorithm now has the similar level of accuracy, our remaining task is to compare their elapsed time. Table 5.4, Figure 5.6 and Figure 5.7 shows the comparison results of k-NN search algorithms: 1) in terms of pre-processing time, C2LSH+ is the fastest, and E2LSH+ is faster than S2LSH. However, their difference does not have significant meaningis because the preprocessing step is performed only once. Also, the difference of E2LSH+ and S2LSH indicates that the pool generation time is only a small portion of preprocessing time. In terms of k-NN search time, S2LSH outperforms both of the algorithms. 2) An interesting finding is that although the C2LSH+ is the newer algorithm than E2LSH+, C2LSH+ is slower than E2LSH+ when using the NUS-WIDE dataset extracted by color histogram features. In the following experiments, we can observe that E2LSH+ is better than C2LSH+ when using color histogram features, and C2LSH+ is better than E2LSH+ when using SIFT features.

Figure 5.8 and 5.9 show the comparison results of k-NN graph construction tasks. These figures show that S2LSH-M significantly outperforms the existing

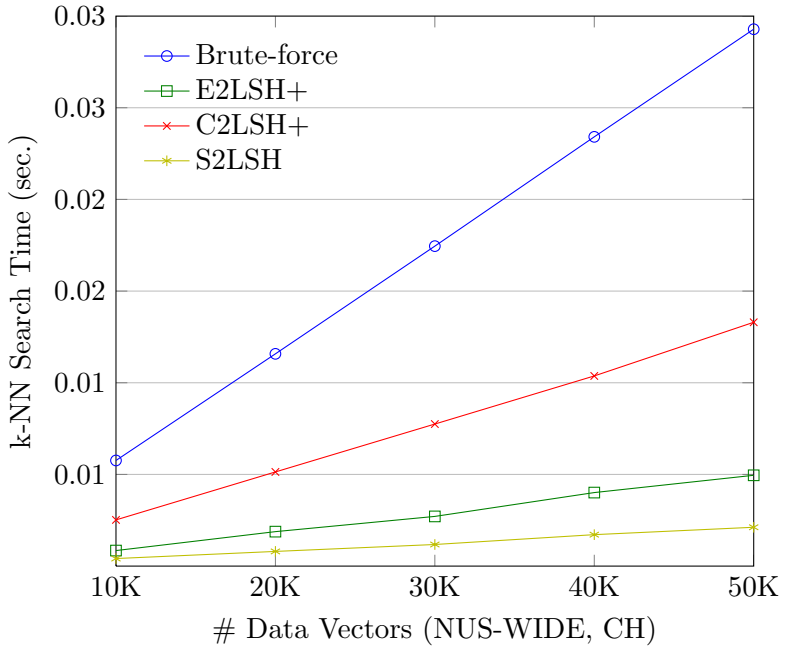


Figure 5.6: Comparsion results of all k-NN search algorithms over the NUS-WIDE dataset (color histogram)

k-NN search algorithms because of the new distance features and optimization techniques. Furthermore, S2LSH-M is even slightly faster than the state-of-the-art k-NN graph construction algorithms. Another interesting finding is that recursive Lanczos bisection is much slower than brute-force search in the dataset represented by SIFT features while it is much faster than brute-force search when using the color histogram features, respectively.

Figure 5.10 and 5.11 show the comparison results of partial k-NN graph construction tasks. In these experiments, now S2LSH-M significantly outperforms all of the other algorithms over two different datasets. Note the elapsed times of NN-Descent and RLB are the almost same as those for k-NN graph construction, because they do not support these types of tasks. Because partial k-NN graph construction is conceptually a combination of k-NN search and k-NN graph construction, k-NN search algorithms could perform better than k-NN graph construction algorithms in one dataset as shown in Figure 5.10,

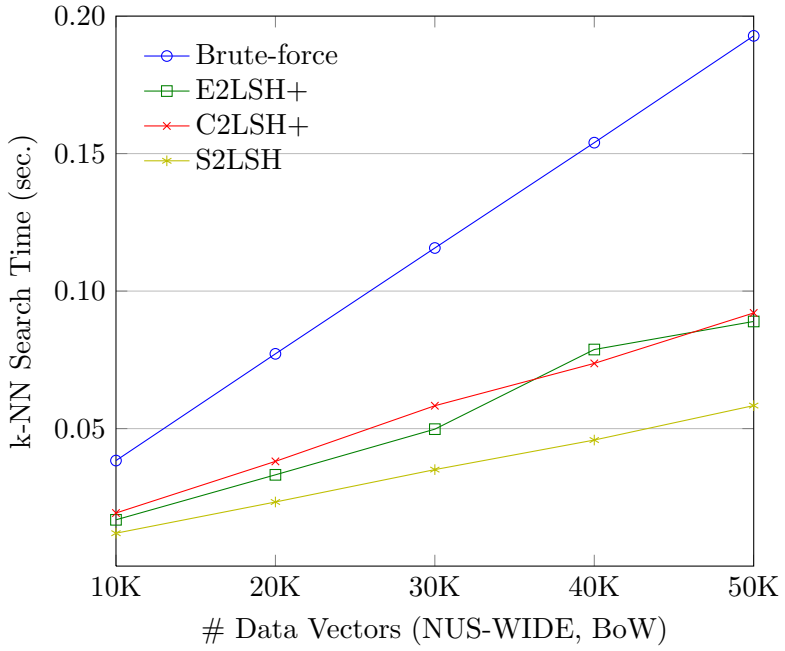


Figure 5.7: Comparison results of all k-NN search algorithms over the NUS-WIDE dataset (SIFT)

whereas k-NN graph construction algorithms could perform better in another dataset as shown in Figure 5.11.

Table 5.5 shows the k-NN search (or k-NN graph construction) time and its corresponding accuracy of each algorithm over different types of datasets. Note our approach outperforms the existing approaches in regardless of datasets (feature extraction methods) and k-NN computation tasks.

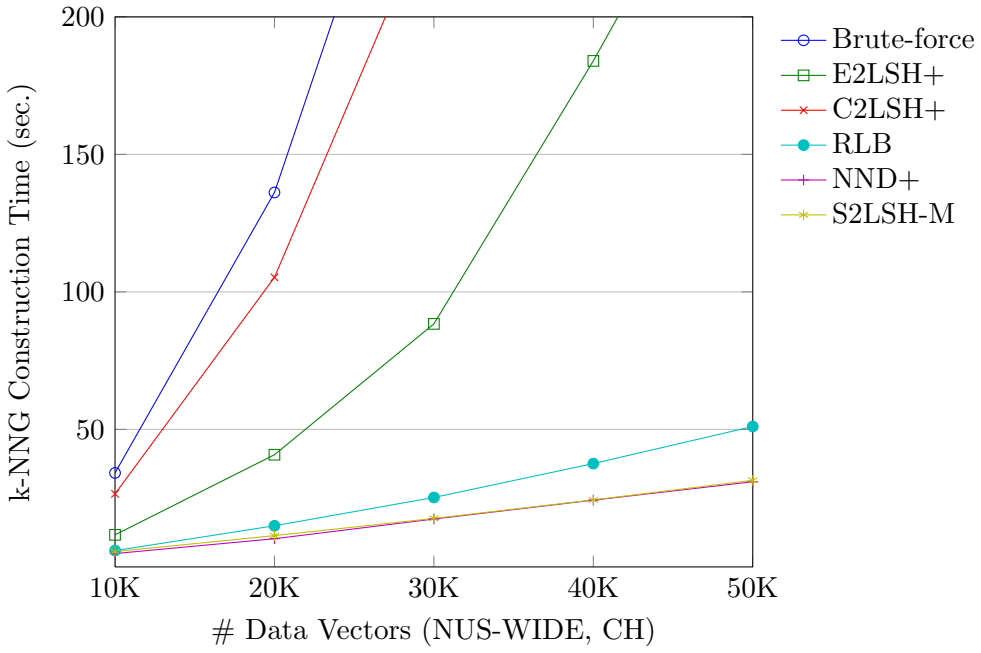


Figure 5.8: Comparison results of k-NN graph construction algorithms over the NUS-WIDE dataset (color histogram)

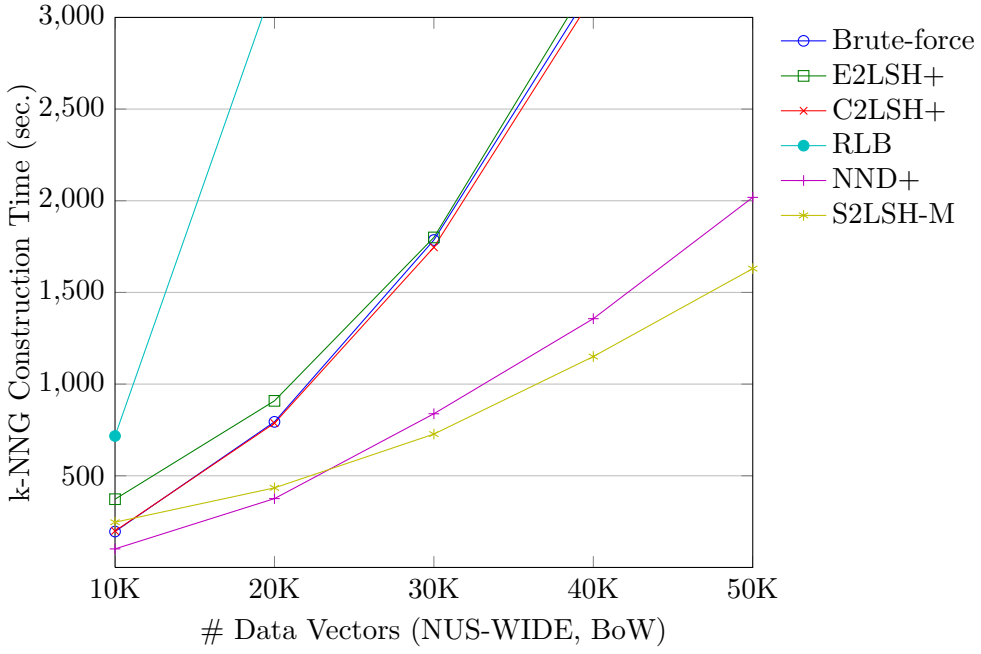


Figure 5.9: Comparison results of k-NN graph construction algorithms over the NUS-WIDE dataset (SIFT)

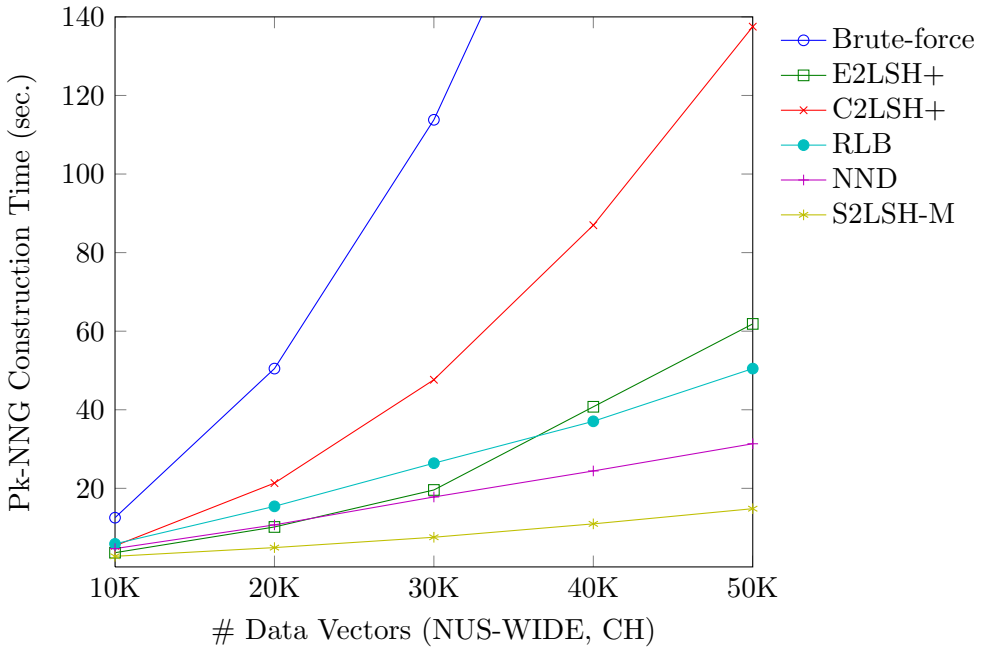


Figure 5.10: Comparison results of partial k-NN graph construction algorithms over the NUS-WIDE dataset (color histogram)

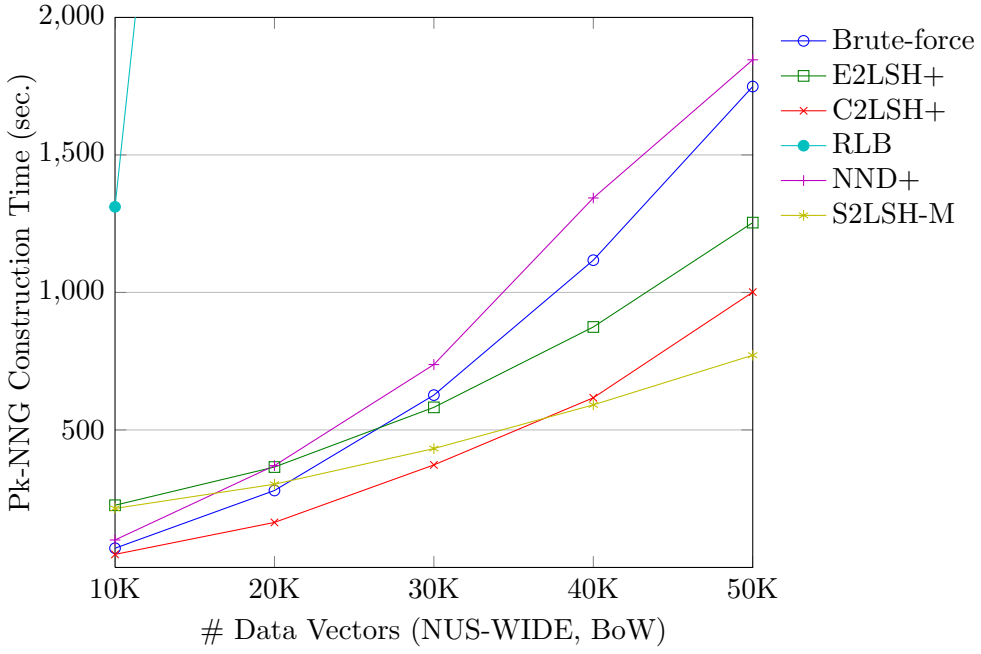


Figure 5.11: Comparison results of partial k-NN graph construction algorithms over the NUS-WIDE dataset (SIFT)

Dataset	k-NN Search				k-NN Graph Construction			Partial k-NN Graph Construction		
	Brute-force	E2LSH+	C2LSH+	S2LSH	Brute-force	NND+	S2LSH-M	Brute-force	NND+	S2LSH-M
Corel	74ms (1.00)	6ms (0.96)	66ms (0.85)	**2ms (0.97)**	14215s (1.00)	**151s (0.94)**	199s (0.91)	5066s (1.00)	145s (0.94)	**136s (0.94)**
NUS-WIDE (CH)	120ms (1.00)	9ms (0.91)	47ms (0.82)	**4ms (0.92)**	14116s (1.00)	314s (0.95)	**240s (0.95)**	5007s (1.00)	315s (0.95)	**94s (0.90)**
Audio	75ms (1.00)	9ms (0.89)	16ms (0.87)	**8ms (0.92)**	2044s (1.00)	175s (0.93)	**82s (0.94)**	734s (1.00)	116s (0.90)	**54s (0.95)**
NUS-WIDE (BoW)	384ms (1.00)	169ms (0.88)	187ms (0.90)	**103ms (0.91)**	19774s (1.00)	7185s (0.90)	**6262s (0.92)**	7017s (1.00)	6844s (0.90)	**2006s (0.90)**
Shape	97ms (1.00)	11ms (0.90)	11ms (0.90)	**10ms (0.93)**	1314s (1.00)	51s (0.93)	**48s (0.94)**	475s (1.00)	51s (0.93)	**40s (0.96)**
MNIST	346ms (1.00)	61ms (0.92)	32ms (0.81)	**21ms (0.94)**	10993s (1.00)	202s (0.91)	**189s (0.94)**	3878s (1.00)	198s (0.94)	**67s (0.97)**
GIST1M	726ms (1.00)	162ms (0.92)	134ms (0.82)	**75ms (0.92)**	36610s (1.00)	3830s (0.91)	**3411s (0.91)**	13068s (1.00)	3805s (0.91)	**1862s (0.92)**

Table 5.5: Comparison of all datasets. The values outside the parentheses are k-NN search (or k-NN graph construction) time, and the values inside the parentheses are the corresponding accuracies.

5.5.3 Performance Analysis

Our approach consists of the three steps so that there are three factors that affect that performance of our algorithm: 1) The performance of locality sensitive hashing, 2) number of signatures in a signature pool, and 3) the effectiveness of our selected features.

Figures 5.12 to 5.17 show that S2LSH also outperforms the existing approaches based on random hyperplane-based LSH, which is one of the most popular LSH schemes. It was originally developed for cosine distance so that we calculate the similarities between vectors based on the cosine similarity measure. Theoretically, for vectors u and v,

$$Pr[h(u) = h(v))] = 1 - \frac{\theta(u, v)}{\pi}, \tag{5.11}$$

where $h(\cdot)$ is a random hyperplane based hash function. Experimental results show that while the performance of the existing approaches significantly varies depending on the datasets being used, our approach delivers relatively consistent performance.

Figure 5.18 indicates that as the number of signatures increases, the performance (in terms of either accuracy or time) consistently increases. The result is intuitive because if there are many signatures, then there is a high probability that there will be more effective signatures. If we can select the effective signatures in a large pool of signatures, the performance would be increased. The S2LSH-OPT in this figure is the one variant of S2LSH that can always select the most effective signatures. This is infeasible in a real-world scenario because S2LSH-OPT knows the answer set in advance in order to optimally select the signatures. Obviously, when we expand the size of the pool, the performance gain of S2LSH-OPT would be higher than that of S2LSH.

Figure 5.19 represents that two S2LSH schemes with different signature selection methods could produce significantly different results. Even though finding the k-nearest neighbors in the 500-dimensional NUS-WIDE is difficult task,

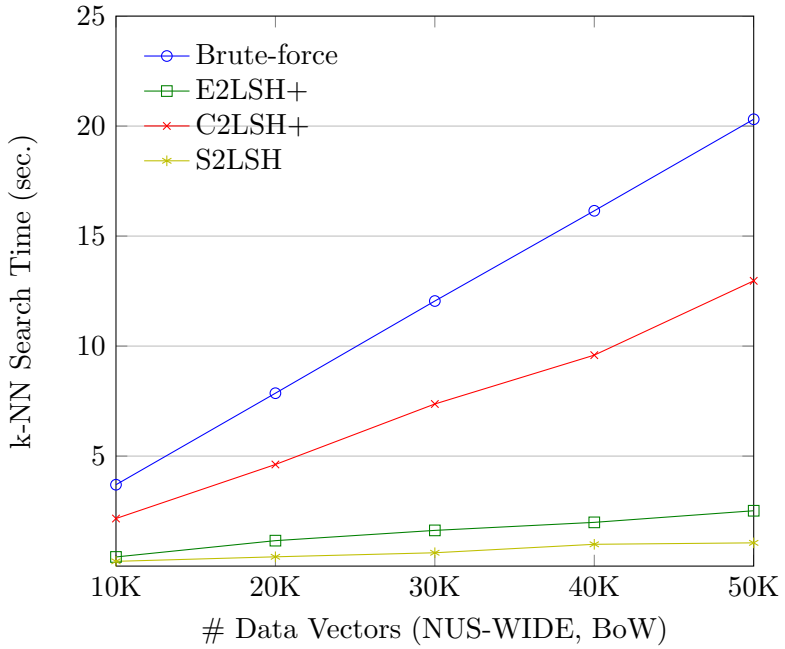


Figure 5.12: Comparison results of all k-NN search algorithms based on random hyperplanes over the NUS-WIDE dataset (color histogram)

S2LSH-OPT significantly reduces the search space. In other words, if we find a more advanced features, then we can expect a huge amount of performance gain.

5.6 Summary

k-Nearest Neighbor (k-NN) search aims at finding k vectors nearest to a query vector in a given dataset. In this chapter, we presented novel methods for generating k-NN search results efficiently regardless of properties of a given dataset. In this method, we construct a highly diversified signature pool consisting of various signatures. The signatures are generated based on a data-dependent LSH algorithm to capture the global topological features specific to the given dataset. Then, for a given query point, we select multiple query-specific signatures from the signature pool in order to find high-quality k-NN candidates of the query point. We also incorporated three additional optimization techniques

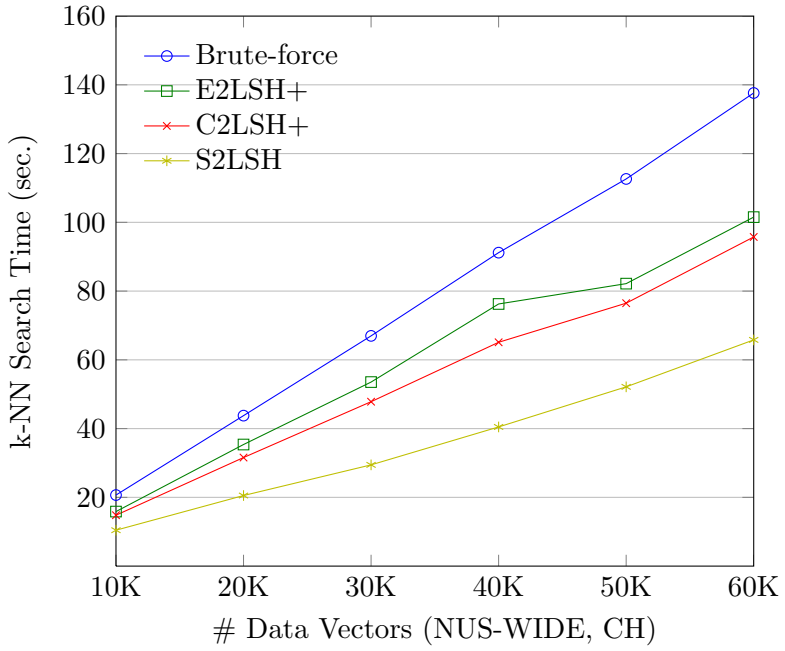


Figure 5.13: Comparsion results of all k-NN search algorithms based on random hyperplanes over the NUS-WIDE dataset (SIFT)

to further improve the performance of S2LSH in a bulk execution setting such as k-NN graph construction.

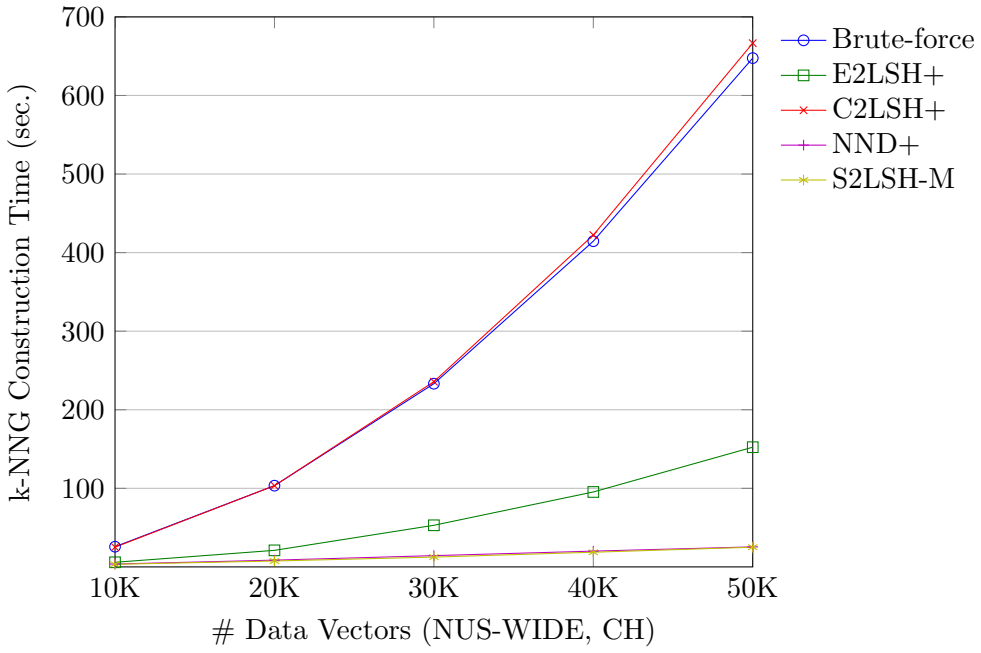


Figure 5.14: Comparison results of k-NN graph construction algorithms based on random hyperplanes over the NUS-WIDE dataset (color histogram)

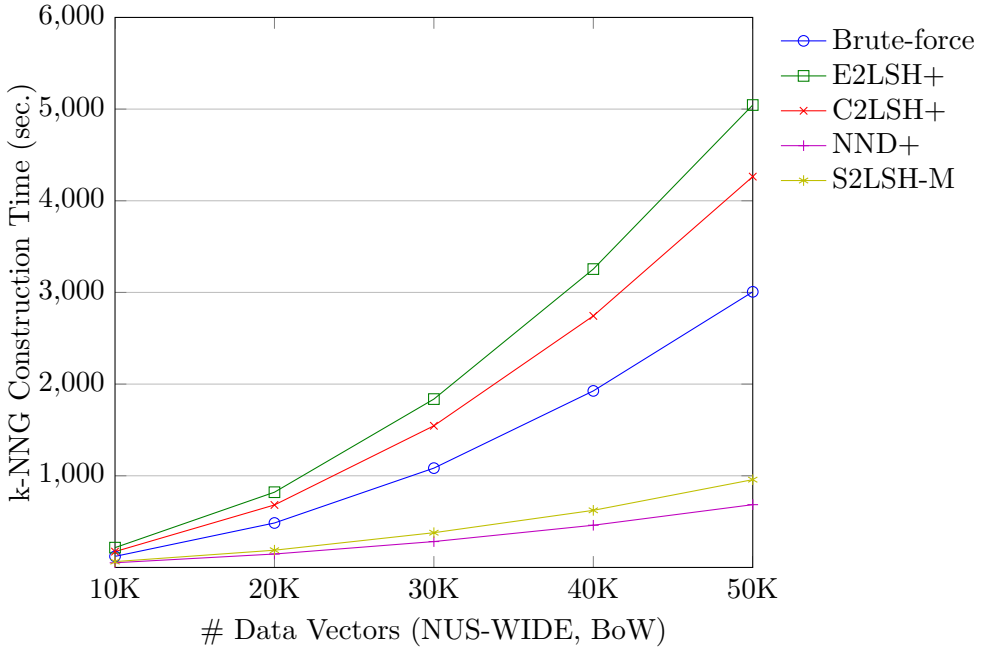


Figure 5.15: Comparison results of k-NN graph construction algorithms based on random hyperplanes over the NUS-WIDE dataset (SIFT)

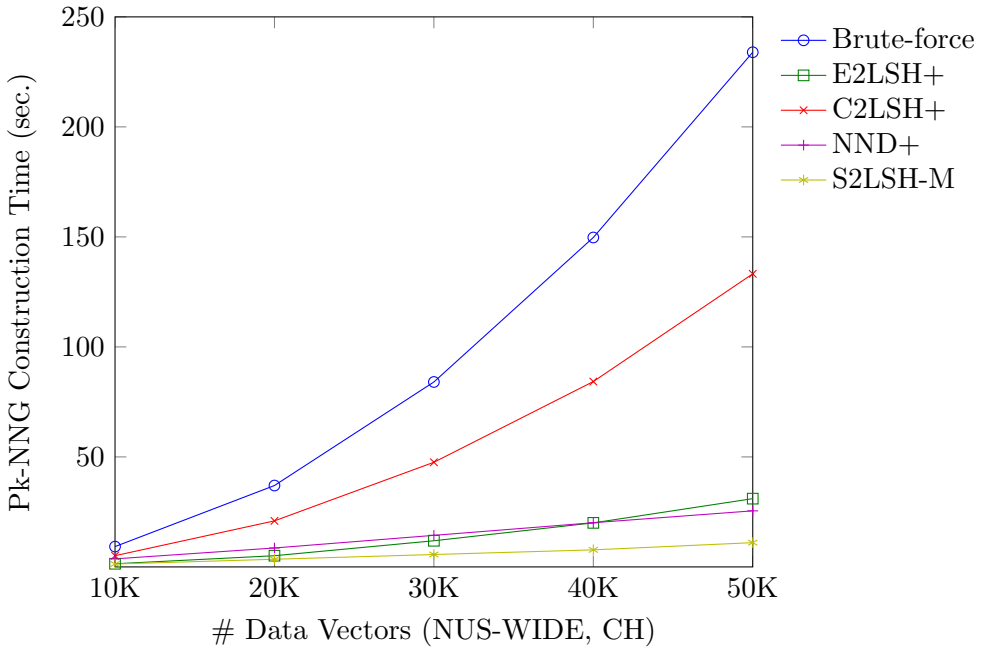


Figure 5.16: Comparison results of partial k-NN graph construction algorithms based on random hyperplanes over the NUS-WIDE dataset (color histogram)

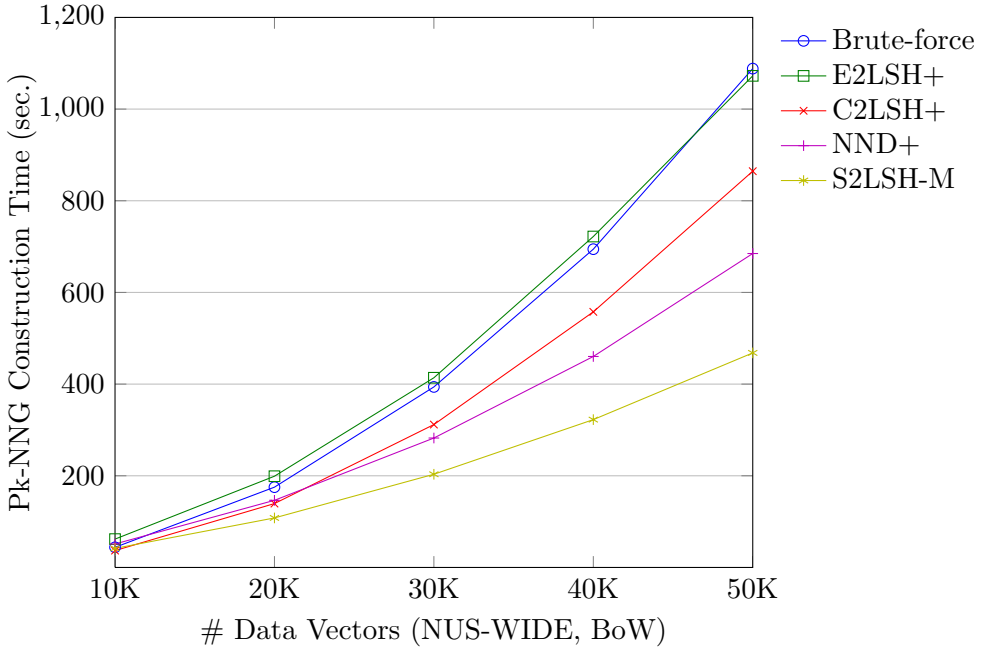


Figure 5.17: Comparison results of partial k-NN graph construction algorithms based on random hyperplanes over the NUS-WIDE dataset (SIFT)

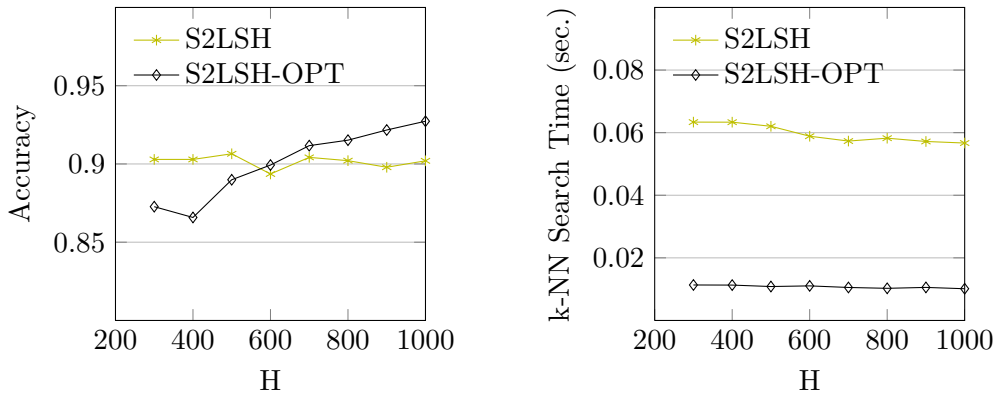


Figure 5.18: Effect of signature pool generation

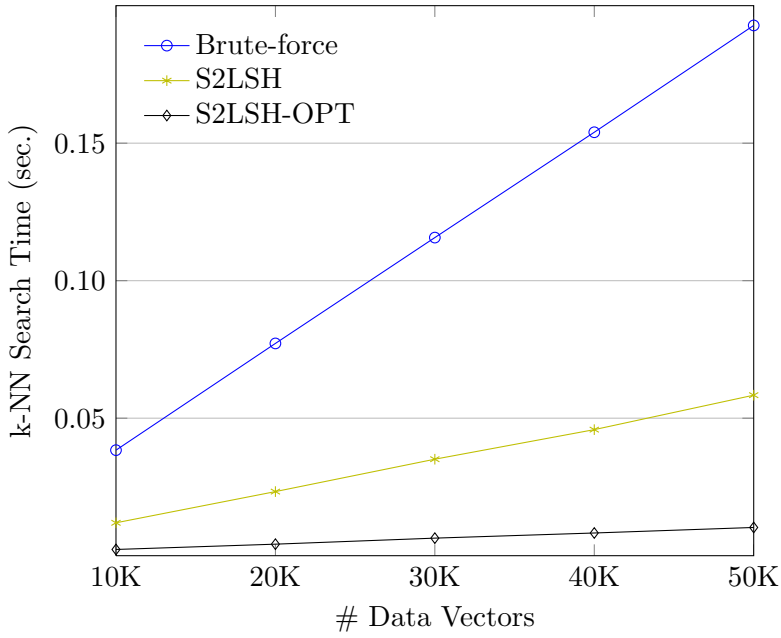


Figure 5.19: Effect of signature selection (NUS-WIDE, BoW). S2L2H-OPT significantly outperforms S2LSH in terms of query processing time; the average accuracy of brute-force search, S2LSH and S2LSH-OPT are 100%, 91% and 91% respectively.

Chapter 6

Conclusion

k-NN search and k-NN graph construction are two of the most important primitive operations in information retrieval, recommender systems and many algorithms in data mining and machine learning. However, existing approaches require a huge amount time for finding k-nearest neighbors and the experimental results do not show the consistent performance levels over different search tasks and types of data. In this dissertation, we introduced two main algorithms to solve these problems. Also, we introduced a fast collaborative filtering algorithm based on a k-NN graph. The contributions of this dissertation are as follows: 1) we developed an efficient and scalable algorithm for finding an approximate k-nearest neighbor graph called *greedy filtering*. The main idea of this approach is to filter node pairs whose large value dimensions do not match at all. In order to avoid skewness in the results and guarantee a linear time complexity, our algorithm chooses essentially a fixed number of nodes pairs as candidates for every node. 2) We presented a novel algorithm for approximate k-NN search called *signature selection LSH*. This approach selects query-specific signatures from a signature pool to pick high-quality k-NN candidates. In order to increase the performance, the signatures are generated based on spherical hashing, which is

one of the most efficient data-dependent LSH algorithms. We also incorporated three additional optimization techniques: bucket hashing, duplicate elimination, and our modified neighborhood propagation method. 3) We introduced a fast collaborative filtering algorithm based on a k-nearest neighbor graph, called *reversed CF*. The main idea of this approach to reverse the process of finding k-nearest neighbors in order to perform fewer predictions while filtering out inaccurate results. The experimental results show that not only are the proposed algorithms much faster than the existing approaches while retaining a high level of accuracy, but also the algorithms consistently outperform the state-of-the-art algorithms across various types of search tasks and datasets.

Bibliography

[1] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl. An Algorithmic framework for performing collaborative filtering. In Proceedings of the 22nd Annual International ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR'99), pp. 230-237.

[2] 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), pp. 285-295.

[3] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. IEEE Internet Computing, 7(1):76-80, 2003.

[4] J. Davidson, B. Liebald, J. Li, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, and B. Livingston and Dasarathi Sampath. The YouTube video recommendation system. In Proceedings of the 4th ACM Conference on Recommender Systems (RecSys'10), pages 293-296, 2010.

[5] D. Lee, S. E. Park, M. Kahng, S. Lee, S.-g. Lee, Exploiting contextual information from event logs for personalized recommendation. In Computer and Information Science, pp. 121-139, 2010.

[6] S. E. Park, S. Lee, and S.-g. Lee. Session-based collaborative filtering for predicting the next song. In Proceedings of the First ACIS/JNU Interna-

tional Conference on Computers, Networks, Systems and Industrial Engineering (CNSI'11), pp. 353-358, 2011.

[7] Y. Park, B. Yang, S. Song, S. Lee, and S.-g. Lee. Context-aware recommendation in smart cars. In Proceedings of the 6th Biennial Workshop on Digital Signal Processing for In-Vehicle Systems (DSP'13), pp. 64-67, 2013.

[8] C. Manning, R. Prabhakar and H. Schutze. Introduction to information retrieval. Cambridge University Press, 2008.

[9] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon. Spherical hashing. In Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12), pages 2957-2964, 2012.

[10] M. Smucker and J. Allan. Find-similar: similarity browsing as a search tool. In Proceedings of the 29th annual international ACM SIGIR conference on Research and Development in Information Retrieval (SIGIR'06), pp. 461-468, 2006.

[11] V. Hautamaki, I. Karkkainen, and P. Franti. Outlier detection using k-nearest neighbour graph. In Proceedings of the 17th International Conference on Pattern Recognition (ICPR'04), pages 430-433, 2004.

[12] J. Chen, H. Fang, and Y. Saad. Fast approximate kNN graph construction for high dimensional data via recursive lanczos bisection. The Journal of Machine Learning Research (JMLR'09), 10:1989-2012, 2009.

[13] Y. Park, S. Park, S.-g. Lee, and W. Jung. Fast collaborative filtering with a k-nearest neighbor graph. In Proceedings of the International Conference on Big Data and Smart Computing (BigComp'14), pages 92-95, 2014.

[14] L. Nie, M. Wang, Z. Zha and T. Chua. Oracle in image search: a content-based approach to performance prediction. ACM Transactions on Information Systems (TOIS'12), 30(2):13, 2012.

[15] G. Paltoglou and M. Thelwall. A study of information retrieval weighting schemes for sentiment analysis. In Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'10), pages 1386-1395, 2010.

[16] G. Qian, S. Sural, Y. Gu, and S. Pramanik. Similarity between Euclidean nd cosine angle distance for nearest neighbor queries. In Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC'04), pages 1232-1237, 2004.

[17] Y. Mu and S. Yan. Non-metric locality-sensitive hashing. In Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI'10), pages 539-544, 2010.

[18] W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In Proceedings of the 28th International Conference on Machine Learning (ICML'11), pages 1-8, 2011.

[19] M. Charikar. Similarity estimation techniques from rounding algorithms. In Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC'02), pages 380-388, 2002.

[20] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In Proceedings of the 20th Annual Symposium on Computational Geometry (SoCG'04), pages 253-262, 2004.

[21] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the 33th Annaul ACM symposium on Theory of Computing, 604-613, 1998.

[22] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In Proceedings of the ACM SIGMOD

International Conference on Management of Data (SIGMOD'09), pages 563-576. 2009.

[23] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12), pages 541-552. 2012.

[24] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In Proceedings of the 20th International World Wide Web Conference (WWW'11), pages 577-586, 2011.

[25] Y. Park, S. Park, S.-g. Lee, and W. Jung. Greedy filtering: a scalable algorithm for k-nearest neighbor graph construction. In Proceedings of the 19th International Conference on Database Systems for Advanced Applications (DASFAA'14). pages 327-341. 2014.

[26] O. Virmajoki and P. Franti. Divide-and-conquer algorithm for creating neighborhood graph for clustering. In Proceedings of the 17th International Conference on Pattern Recognition (ICPR'04), 1:264-267, 2004.

[27] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12), pp. 1106-1113. 2012.

[28] D. Wang, L. Shi, and J. Cao. Fast algorithm for approximate k-nearest neighbor graph construction. In Proceedings of the IEEE 13th International Conference on Data Mining Workshops (ICDMW'13), pages 349-356, 2013.

[29] Y.-M. Zhang, K. Huang, G. Geng, and C.-L. Liu. Fast knn graph construction with locality sensitive hashing. In Machine Learning and Knowledge Discovery in Databases (PKDD'13), pages 660-674, 2013.

[30] Y. Park, S. Park, S. Lee, and W. Jung. Scalable k-nearest neighbor graph construction based on Greedy Filtering. In Proceedings of the 22nd International World Wide Web Conference (WWW'13), pages 227-228, 2013.

[31] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00), pages 169-178, 2000.

[32] D. Lee, J. Park, J. Shim, and S. Lee. An efficient similarity join algorithm with cosine similarity predicate. In Proceedings of the 21st International Conference on Database and Expert Systems Applications (DEXA'10), pages 422-436, 2010.

[33] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In Proceedings of the 16th International World Wide Web Conference (WWW'07), pages 131-140, 2007.

[34] C. Xiao, W. Wang, X. Lin, J. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. ACM Transactions on Database Systems (TODS'11), 36(3):15-41, 2011.

[35] Y. Kim, and K. Shim. Parallel top-k similarity join algorithms using MapReduce. In Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE'12), pages 510-521, 2012.

[36] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), pages 518-529, 1999.

[37] B. Durme, and A. Lall. Online generation of locality sensitive hash signatures. In Proceedings ofthe 48th Annual Meeting of the Association for Computational Linguistics (ACL'10), pages 231-235, 2010.

[38] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. Computer Networks and ISDN Systems, 29(8):1157-1166, 1997.

[39] A. Z. Broder. On the resemblance and containment of documents. In Proceedings of the Compression and Complexity and Sequences 1997, pages 21-29, 1997.

[40] A. Said, B. Jain, and S. Albayrak. Analyzing weighting schemes in collaborative filtering: Cold start, post cold start and power users. In the 27th Symposium On Applied Computing (SAC'12), pages 2035-2040, 2012.

[41] C. Xiao, W. Wang, X Lin, and H. Shang. Top-k set similarity joins. In Proceedings of the 25th International Conference on Data Engineering (ICDE'09), pages 916-927, 2009.

[42] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In Advances in Neural Information Processing Systems (NIPS'09), pages 1753-1760, 2009.

[43] Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In Proceedings fo the 13th ACM International Conference on Information and Knowledge Management (CIKM'04), pages 208-217. 2004.

[44] L. Shapiro and G. C. Stockman. Computer vision. Prentice Hall, 2001.

[45] G. Tzanetakis and P. Cook. Marsyas. A framework for audio analysis. Organised sound, 4(03):169-175, 2000.

[46] D. G. Lowe. Object recognition from local sacle-invariant features. In Proceedings of the 7th IEEE International Conference on Computer Vision (ICCV'99), 2:1150-1157, 1999.

[47] M. Kazhdan, T. Funkhouser, and S. Rusinkiewicz. Rotation invariant spherical harmonic representation of 3 d shape descriptors. In Proceedings of the ACM Symposium on Geometry Processing 2003, 6:156-165, 2003.

[48] Y. LeCun, C. Cortes, and C. Burges. Mnist dataset, 2000.

[49] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. IEEE Transanctions on Pattern Analysis and Machine Intelligence, 33(1):117-128, 2011.

[50] P. Cremonesi, Y. Koren, and R. Turrin. Performance of recommender algorithms on top-n recommendation tasks. In Proceedings of the 4th ACM conference on Recommender Systems (RecSys'10), pp. 39-46, 2010.

[51] S. Lee, S.-i. Song, M. Kahng, D. Lee, S.-g. Lee, Random walk based entity ranking on graph for multidimensional recommendation. In Proceedings of the 5th ACM conference on Recommender Systems (RecSys'11), pp. 93-100, 2011.

[52] S. Lee, S. Park, M. Kahng, and S.-g. Lee. Pathrank: Ranking nodes on a heterogeneous graph for flexible hybrid recommender systems. Expert Systems with Applications (ESWA'13), 40(2):684-697, 2013.

[53] R. Jin, and J. Y. Chai, and L. Si. An automatic weighting scheme for collaborative filtering. In Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'04), pages 337-344, 2004.

[54] 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 (TKDE'05), 17:734-749, 2005.

[55] E. Achtert, C. Bohm, P. Kroger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06), pp. 515-526, 2006.

[56] C. Birtolo and D. Ronca. Advances in clustering collaborative filtering by means of fuzzy c-means and trust. Expert Systems with Applications (ESWA'13), 40:6997-7009, 2013.

[57] 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), pp. 271-280, 2007.

[58] C. Desrosiers and G. Karypis. A comprehensive survey of neighborhood-based recommendation methods. In Recommender Systems Handbook, pp. 107-144, 2011.

[59] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. Computer, 42, pp. 30-37, 2009.

[60] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In Proceedings of the IEEE 25th International conference on data engineering (ICDE'09), pp. 916-927, 2009.

[61] D. Johnson. Approximation algorithms for combinatorial problems. Journal of Computer and System Sciences (JCSS'74), 9(3):256-278, 1974.

초록

k-인접 이웃 계산은 추천 시스템, 검색 엔진, 그리고 많은 데이터 마이닝 및 기계 학습 알고리즘에서 핵심적인 역할을 수행한다. 학계에서는 협업적 필터링, 검색 및 브라우징, 군집화, 분류, 이상치 탐지, 차원 축소 등 k-인접 이웃에 기반한 다양한 알고리즘이 연구되고 있으며, 산업계에서는 YouTube 및 Amazon 추천 시스템, Google 검색 엔진 등 많은 상용 시스템들이 k-인접 이웃 정보를 활용하고 있다. 그러나 기존의 k-인접 이웃 계산 알고리즘들은 개체의 수 또는 차원의 수가 증가했을 때 상당한 계산 시간을 필요로 한다는 문제점이 있다. 또, 계산 작업의 종류와 데이터셋에 따라 일관된 성능을 나타내지 못하기 때문에, 응용 분야에 따라서는 작은 데이터에 대해서도 빠르게 k-인접 이웃을 찾기가 어려울 수 있다.

본 학위 논문은 위 문제들을 해결하기 위해 다목적 고속 근사 알고리즘을 제시하고, 관련된 응용 알고리즘을 제안한다. 본 학위 논문이 기여한 점은 다음과 같다. 첫째, 텍스트 데이터에 대해 특화된, 확장성 있는 고속 k-인접 이웃 그래프 생성 알고리즘(Greedy Filtering)을 제안하였다. 이 알고리즘은 데이터가 TF-IDF 가중치가 반영된 벡터로 표현되었다고 가정하고, 값이 큰 차원이 일치하지 않을 경우 유사도 계산을 수행하지 않는 방식을 사용한다. 둘째, 항목 기반 추천 알고리즘을 변형하여, k-인접 이웃 그래프를 활용한 고속 협업적 필터링 알고리즘(Reversed CF)을 제시하였다. 본 알고리즘은 항목 기반 추천 알고리즘의 k-인접 이웃을 찾는 과정을 개념적으로 뒤바꿈으로써 Greedy Filtering을 사용하여 빠른 추천이 가능하도록 설계되었다. 마지막으로, 텍스트, 로그, 멀티미디어 데이터에 대해 모두 사용할 수 있는 고속 k-인접 이웃 검색 알고리즘(Signature Selection LSH)을 제안하였다. 다양한 특징을 추출하고, 그 중에서 가장 검색에 효과적인 특징을 질의 의존적으로 선택하기 때문에 알고리즘의 성능이 데이터의 특성에 큰 영향을 받지 않는다. 특징 추출 방법을 변형하면, k-인접 이웃 검색뿐만 아니라 k-인접 이웃 그래프의 일부분 또는 전체를 생성

하는 데에도 효과적으로 사용될 수 있다. 위 알고리즘들은 이론적 분석 또는 실험적 검증을 통해, 기존 방법에 비해 훨씬 빠르면서 더 높은 정확도를 보장한다는 것을 보였다. 또, 다양한 종류의 k-인접 이웃 계산 작업과 데이터셋에 대해 일관된 성능을 나타냈다.

주요어: k-인접 이웃 검색, k-인접 이웃 그래프 생성, 협업적 필터링
학번: 2010-30219