L2Knng: Fast Exact K-Nearest Neighbor Graph Construction with L2-Norm Pruning

David C. Anastasiu
University of Minnesota, Twin Cities
Minneapolis, USA
dragos@cs.umn.edu

George Karypis
University of Minnesota, Twin Cities
Minneapolis, USA
karypis@cs.umn.edu

ABSTRACT

The k-nearest neighbor graph is often used as a building block in information retrieval, clustering, online advertising, and recommender systems algorithms. The complexity of constructing the exact k-nearest neighbor graph is quadratic on the number of objects that are compared, and most existing methods solve the problem approximately. We present `L2Knng`, an efficient algorithm that finds the exact cosine similarity k-nearest neighbor graph for a set of sparse high-dimensional objects. Our algorithm quickly builds an approximate solution to the problem, identifying many of the most similar neighbors, and then uses theoretic bounds on the similarity of two vectors, based on the ℓ^2-norm of part of the vectors, to find each object's exact k-neighborhood. We perform an extensive evaluation of our algorithm, comparing against both exact and approximate baselines, and demonstrate the efficiency of our method across a variety of real-world datasets and neighborhood sizes. Our approximate and exact `L2Knng` variants compute the k-nearest neighbor graph up to an order of magnitude faster than their respective baselines.

Keywords

k-nearest neighbor graph, similarity search, top-k, cosine similarity

1. INTRODUCTION

Computing the k-nearest neighbor graph (k-NNG) for a set of objects is a common task in fields such as information retrieval, clustering, recommender systems, and online advertising. For example, item-based nearest neighbor collaborative filtering algorithms recommend items (e.g., books or movies) to a user based on the k most similar items to each of the user's preferred items [16].

The task of computing the k-NNG is computationally expensive, requiring $O(n^2)$ similarity comparisons, given a set of n objects. As a result, current methods that tackle the problem focus on either pruning the similarity search space, or solving the problem approximately. Exact methods either rely on efficient index structures to limit computation to only those object pairs with non-zero similarities (e.g., *IDX* [21], *BMM* [9]) or on effective pruning techniques that enable efficient discovery of all object pairs with similarity above a threshold t, which is iteratively lowered until all objects have at least k neighbors (e.g., *SIM* [21]). Approximate methods employ heuristic strategies that aim to return a neighborhood that contains the majority of the *true neighbors*, i.e., those that would be found by an exhaustive search. These strategies include focusing on object pairs that share high-weight features (e.g., *Greedy Filtering* [21]) and iterative improvement of an initial random k-NNG by considering neighbors' neighbors as potential neighbors (e.g., *NN-Descent* [11]).

CIKM'15, October 19–23, 2015, Melbourne, Australia.
© 2015 ACM. ISBN 978-1-4503-3794-6/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2806416.2806534.

Despite this extensive body of work, each of these two classes of methods have their limitations. Exact methods either do not rely on pruning, use computationally expensive pruning estimates, or repeat many similarity estimations in the search for the true minimum neighborhood similarity threshold across all neighborhoods. Approximate methods use candidate selection and comparison strategies that can lead to unnecessary similarity computations.

In this work, we introduce `L2Knng`, which solves the *exact* cosine similarity k-NNG construction problem efficiently by effectively pruning much of the similarity search space. We focus on input objects that are encoded by *sparse high-dimensional non-negative vectors*. Some examples include Web pages, and user/item profiles in a recommender system. In this context, *cosine similarity* has long been a standard comparison measure, especially in the fields of information retrieval [19] and text mining [14]. To solve the k-NNG construction problem, `L2Knng` first obtains an initial approximate solution by considering, for each query object, a set of candidates that are likely to be part of the final k-NNG. Then, using the approximate graph as a guide to prune the search space, `L2Knng` executes a neighbor search for each input object. We introduce several filtering methods specific to the problem at hand that successfully prune most objects that will not be part of the final k-NNG, resulting in few objects having their similarity to a query object computed in full. We evaluate our methods experimentally on a variety of real-world datasets and neighborhood sizes, against both exact and approximate baselines. We find that `L2Knng` can provide over an order of magnitude performance improvement over baselines.

The remainder of the paper is organized as follows. Section 2 introduces the problem and notation used throughout the paper. Section 3 summarizes existing approaches to solving the k-NNG construction problem. Section 4 introduces our methods for constructing the exact and approximate k-NNG. We describe our evaluation methodology in Section 5 and analyze experimental results in Section 6, and Section 7 concludes the paper.

2. DEFINITION & NOTATIONS

Let $D = \{d_1, d_2, \ldots, d_n\}$ be a set of objects such that each object d_i is a (sparse) vector in an m dimensional feature space. We will use d_i to indicate the ith object, $\boldsymbol{d}_i$ to indicate the feature vector associated with the ith object, and $d_{i,j}$ to indicate the value (or weight) of the jth feature of object d_i.

We use the *cosine function* to measure vector similarity. To simplify the presentation of the algorithms, we assume that all vectors have been scaled to be of unit length ($||\boldsymbol{d}_i|| = 1, \forall d_i \in D$). Given that, the cosine between two vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_j$ is simply their dot-product, which we denote by $\text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j)$.

Given object d_i, its k nearest neighbors in D, denoted by N_{d_i}, is the set of objects in $D \setminus \{d_i\}$ whose similarity with d_i is the highest among all objects in $D \setminus \{d_i\}$. The k-NNG of D is a directed graph $G = (V, E)$ where vertices correspond to the objects and an edge (v_i, v_j) indicates that the jth object is among the k nearest neighbors of the ith object. An approximate k-NNG is one in which the k neighbors of each vertex do not necessarily correspond to the k most similar objects.

During its execution, our method keeps track of up to k neighbors in each object's neighborhood. We denote by the *minimum (neighborhood) similarity* σ_{d_i} the minimum similarity between object d_i and one of its current k neighbors. We say that a neighborhood is *improved* when its minimum similarity σ_{d_i} increases in value, and it is *complete* once all true neighbors that belong to a neighborhood have been added to it. Note that subsequently processed objects that have σ_{d_i} similarity with the query object will not be added to the neighborhood as they do not improve it.

An *inverted index* representation of D is a set of m lists, $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$, one for each feature. List I_j contains pairs $(d_i, d_{i,j})$, also called postings, where d_i is an indexed object that has a non-zero value for feature j and $d_{i,j}$ is that value. Postings may store additional information, such as the position of the feature in the given document or other statistics.

Given a vector $\boldsymbol{d}_i$ and a dimension p, we will denote by $\boldsymbol{d}_i^{\leq p}$ the vector $\langle d_{i,1}, \ldots, d_{i,p}, 0, \ldots, 0\rangle$, obtained by keeping the p leading dimensions in $\boldsymbol{d}_i$, which we call the *prefix* (vector) of $\boldsymbol{d}_i$. Similarly, we refer to $\boldsymbol{d}_i^{>p} = \langle 0, \ldots, 0, d_{i,p+1}, \ldots, d_{i,m}\rangle$ as the *suffix* of $\boldsymbol{d}_i$, obtained by setting the first p dimensions of $\boldsymbol{d}_i$ to 0. One can then verify that

$$\boldsymbol{d}_i = \boldsymbol{d}_i^{\leq p} + \boldsymbol{d}_i^{>p}, \text{and}$$
$$\text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j) = \text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j^{\leq p}) + \text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_j^{>p}).$$

Table 1 provides a summary of notation used in this work.

Table 1: Notation used throughout the work

	Description
D	set of objects
k	size of desired neighborhoods
$\boldsymbol{d}_i$	vector representing object d_i
$d_{i,j}$	value for jth feature in $\boldsymbol{d}_i$
$\boldsymbol{d}_i^{\leq p}, \boldsymbol{d}_i^{>p}$	prefix and suffix of $\boldsymbol{d}_i$ at dimension p
N_{d_i}	neighborhood for object d_i
σ_{d_i}	smallest similarity value in N_{d_i}
t_{d_i}	similarity threshold used when indexing d_i
$\mathcal{N}$	set of neighborhoods
$\hat{\mathcal{N}}$	set of initial approximate neighborhoods
$\mathcal{I}$	inverted index
it	minimum indexing threshold for indexed objects
μ	candidate list sizes
γ	number of neighborhood enhancement updates
δ	early neighborhood enhancement termination
ν	number of completion blocks

3. RELATED WORK

Relatively few k-NNG construction algorithms have been designed to address cosine similarity. Park et al. [21] describe *Greedy Filtering*, an approximate filtering-based approach which prioritizes computing similarities between objects with high weight features in common. After first reordering the dimensions of each vector based on their weight, in decreasing weight order, the algorithm builds a partial inverted index, which it uses to find candidates for each object. Candidates for an object d_i are those objects in the inverted index lists associated with the leading dimensions in $\boldsymbol{d}_i$, i.e., the prefix of $\boldsymbol{d}_i$. *Greedy Filtering* indexes enough of each vector's prefix as to lead to at least μ candidates for each object. After all prefixes are identified and the partial inverted index is constructed, *Greedy Filtering* computes pairwise similarities of objects in each inverted index list, which can lead to much more than μ similarity computations for each object, and repeated computations for pairs of objects with two or more common features in their prefixes.

In *NN-Descent*, Dong et al. [11] follow an iterative neighborhood improvement strategy based on the intuition that similar objects are likely to be found among the neighborhoods of objects in a query object's neighborhood. Starting with a randomly chosen initial k-NNG, they iteratively improve the graph by computing, for each object d_i, via a *local join*, pairwise similarities between d_i, objects in its neighborhood, and those objects that contain d_i in their neighborhoods. The neighborhoods of both objects participating in a similarity computation are updated with the result. They avoid duplication of effort between iterations by only allowing an object to participate in the local join if it has been added to some neighborhood in the last update. Sampling and early termination parameters provide a way to control the compromise between algorithm runtime and recall. However, *NN-Descent* computes $O(n \times k^2)$ object similarities in its first iteration. Furthermore, the algorithm does not provide a way to filter out candidates that are unlikely to improve the query object's neighborhood.

A number of k-NNG construction algorithms have been proposed for *metric spaces*, where we seek the k objects with the smallest metric distance from the query. Tree-based data structures are often used to facilitate partitioning the search space, allowing neighbor searches to be prioritized within grids close to the one the query object is in [5]. These types of methods have been shown effective in low dimensional spaces, but do not scale well as dimensionality increases.

Top-k document retrieval is a related problem from information retrieval, which has had many proposed solutions over the years. Most methods in this class have been designed for very large document collections, focused on minimizing and/or parallelizing operations needed to quickly answer fairly short input queries. Result sets are in most cases inexact. Some recent works use an in-memory inverted index and pruning, called *safe early termination*, to return the same result set as an exhaustive search [6, 9, 10, 22, 23]. One could then solve the exact k-NNG problem by executing n top-k queries with one of these methods, one for each of the input objects. In their Block-Max WAND (*BMW*) method [10], Ding and Suel use an augmented index structure, called a Block-Max index, which stores inverted lists as compressed blocks of postings, along with the maximum score that could be achieved given the values in the block postings. By using the block maximum scores for early termination, many blocks can be skipped, resulting in improved execution. Dimopoulos et al. [9] extended the work of Ding and Suel and designed several methods that take advantage of Block-Max type indexes. Among them, docID-oriented Block-Max Maxscore with variable block sizes (*BMM*) has been shown to outperform the others and several baselines (including *BMW*) for long queries. The method partitions the postings in each inverted list into blocks of equally-sized ID ranges, allowing fast look-up for the block a document's posting may be found in. Block sizes vary based on the number of postings in each list. For each block, *BMM* also keeps track of the maximum document ID and maximum score for any of the postings in the block. The *Maxscore* [24] algorithm described by Turtle and Flood is then adapted to use block maximum scores for early termination.

Locality Sensitive Hashing (LSH) [13, 15] uses families of functions that hash *signatures* of similar objects to the same bucket with high probability. The objects in the buckets that a query object hashes to can be considered its neighbors. The similarity with a neighbor can then either be estimated by comparing the object signatures or computed exactly. Created initially to solve the top-k retrieval problem, LSH has been shown effective at solving the nearest-neighbor problem (1-NNG), but suffers from low recall as the required neighborhood size increases [10]. In this work, we specifically focus on results with high recall (at least 95%). Some recent LSH variations have tackled the k-NNG problem specifically (e.g., E2LSH [2] and DSH [12]), but focus on metric distance functions between objects, such as the Euclidean distance.

Another related problem is All-Pairs Similarity Search (APSS), or *similarity join*, which returns all object pairs in D with a similarity value of at least some threshold t. Bayardo et al. [4] proposed an initial algorithm to solve APSS which employed several strategies to prune the search space, based on a predefined object processing order. The majority of subsequently developed APSS methods [1, 3, 17] use the same framework as in their work. In the context of cosine similarity, Anastasiu and Karypis recently introduced *L2AP* [1], which has been shown to outperform all previous APSS methods by introducing tighter pruning bounds in each phase of the framework.

There have been a large number of top-k retrieval and approximate k-NN search methods designed for distributed or parallel architectures, which can be used to search Web-scale datasets. The focus in these methods are query objects outside the input set. In contrast, L2Knng is an in-memory serial method designed for the efficient construction of a k-NNG from the input set of objects. We leave parallel and distributed extensions as future work.

4. METHODS

The L2Knng algorithm consists of two distinct steps. In the first step, it uses a fast method that identifies, for each object, k similar objects that may not necessarily be the k nearest neighbors. In the second step, it scans over all the objects and progressively updates the k most similar objects of each object. Specifically, while processing an object d_i, which we call the *query*, L2Knng updates the k nearest neighbors of all previously processed objects by taking into account their similarity to the query object. At the same time, it updates the k most similar objects of the query object by considering its similarity to the preceding objects. Since the second step potentially considers all pairs of objects, the final set of the k most similar objects for each object are guaranteed to be their k nearest neighbors.

The key to L2Knng's efficiency stems from the following: (i) It uses an index data structure that enables it to quickly find potential neighbors, while pruning some that do not have enough features in common with the object being indexed. (ii) When searching for neighbors, it uses several vector similarity theoretic bounds to filter out many of the potential neighbors found by traversing the index. (iii) It uses a block processing strategy, which leads to efficient traversal of the inverted index lists and additionally improves the effectiveness of the pruning bounds. (iv) Finally, the initial approximate k-NNG built in its first step is instrumental towards effective indexing and pruning in L2Knng.

4.1 Approximate graph construction

L2KnngApprox is the inexact k-NNG construction method used by L2Knng to build an initial approximate graph. It consists of two steps. First, it builds a set of initial neighborhoods, relying on the idea that *high-weight features count heavily towards the similarity of two vectors* [7, 21]. Then, given that *an object's neighbor's neighbor is also likely their neighbor* [11, 20], it iteratively enhances the k-NNG by looking for new candidates in each neighbor's neighborhood.

The first step is achieved as follows. For each object d_i, L2KnngApprox builds a list of up to μ ($\mu \geq k$) candidates, choosing among those objects that have features in common with d_i until there are no more features to check or μ candidates were found. It then computes the exact similarities of all candidates with d_i and adds the objects with the top k values to d_i's initial neighborhood.

The choice of candidate objects is crucial to obtaining an approximate graph that is close to the exact k-NNG. L2KnngApprox uses an inverted index to identify candidate objects with common features with d_i. As a heuristic way to prioritize high-weight common features, it sorts the features in each vector in decreasing weight order, and sorts each of the lists in the inverted index in decreasing order of feature weights. L2KnngApprox then traverses two index lists at a time, in decreasing order of their associated weights in $\boldsymbol{d}_i$. From the two lists, it chooses the candidate d_c with the higher prefix dot product, which is more likely to be a true neighbor.

In the second step, L2KnngApprox executes up to γ iterative neighborhood enhancement updates, in which, for each object d_i, its k neighbors are updated by taking into account its similarity to some of the objects that are neighbors of its neighbors. This is done as follows. For each object d_i, it traverses its neighborhood in decreasing order of d_i's neighbor similarities. Given some neighbor d_j, it then traverses its neighborhood, in decreasing order of d_j's neighbor similarities, to identify potential neighbors for d_i. Avoiding objects that are already in d_i's neighborhood or have d_i in their neighborhood, L2KnngApprox greedily chooses as candidates only those neighbor's neighbors d_k with a similarity value greater or equal than that between the query vector and its neighbor, $\text{sim}(\boldsymbol{d}_j, \boldsymbol{d}_k) \geq \text{sim}(\boldsymbol{d}_i, \boldsymbol{d}_j)$, and limits the size of the candidate list to be μ. L2KnngApprox then computes similarities between $\boldsymbol{d}_i$ and candidates, updating both relevant neighborhoods with the results. We use δ as an early termination parameter, stopping iterations early if less than $\delta \times k \times |D|$ neighborhood changes occurred in an update.

Our strategy for choosing candidates in the first step improves upon the work of Park et al. [21] by limiting, for each object, the number of computed similarities. High quality candidates are greedily chosen from few inverted index lists. The neighborhood enhancement step improves upon the work of Dong et al. [11] in two ways. First, it ensures an upper bound on the number of similarity computations and prioritizes those candidates more likely to improve the neighborhood. Second, the enhancement steps will probably converge faster and to higher recall, as the input neighbors likely have higher similarity values than the randomly chosen neighbors in their method.

4.2 Indexing

L2Knng uses information in the approximate k-NNG to prune some object pairs by indexing only a subset of the features in each object. In each iteration, L2Knng needs to identify among the previously processed objects those whose neighborhoods can be updated by including the *query* object d_i. In order to do this efficiently, it builds an inverted index incrementally, delaying the indexing of d_i until after its processing. However, future potential neighbors can only improve d_i's neighborhood if their similarity with d_i is higher than the minimum similarity in d_i's neighborhood, σ_{d_i}. In a similar context, Chaudhuri et al. [8] noted that, given a predefined feature order, one can stop indexing features in $\boldsymbol{d}_i$ as soon as they can ensure d_i will be found when processing future objects that have a greater similarity with d_i than σ_{d_i}. By indexing only the leading features of the query object, its prefix, those future objects with common features with d_i only in its suffix, which will not be able to improve its neighborhood, will not be encountered when traversing the index and will thus be automatically pruned.

L2Knng indexes objects until their suffix ℓ^2-norm falls below the minimum neighborhood similarity σ_{d_i}. Given that all vectors in the dataset have unit length, based on the Cauchy-Schwarz inequality, the suffix ℓ^2-norm of d_i is an upper bound of the similarity of d_i's suffix with any other object [1], including unprocessed objects in the set,

$$\text{dot}(\boldsymbol{d}_i^{>j}, \cdot) \leq ||\boldsymbol{d}_i^{>j}|| \times ||\cdot|| = ||\boldsymbol{d}_i^{>j}||.$$

Once the suffix norm $||\boldsymbol{d}_i^{>j}||$ falls below σ_{d_i}, if no common features were found between d_i and some object d_c within d_i's indexed prefix, then d_c cannot improve d_i's neighborhood, since,

$$\text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_c) = \text{dot}(\boldsymbol{d}_i^{\leq j}, \boldsymbol{d}_c) + \text{dot}(\boldsymbol{d}_i^{>j}, \boldsymbol{d}_c) < 0 + \sigma_{d_i}.$$

The query object d_i must also be identified when processing future objects if d_i can improve their neighborhoods. Let t_{d_i} be the minimum neighborhood similarity for object d_i at the time of its indexing (*indexing threshold*), which later may be different than σ_{d_i}. Consider indexing an object d_i at a threshold t_{d_i} and then processing an object d_j with a smaller minimum neighborhood similarity. If $\sigma_{d_j} \leq \text{sim}(\boldsymbol{d}_i, \boldsymbol{d}_j) < t_{d_i}$, then d_i is no longer guaranteed to be found when processing d_j, and d_j's neighborhood may be inexact at the end of the algorithm execution. Therefore, to ensure correctness, objects must be indexed in a strictly non-decreasing indexing threshold order. L2Knng thus fixes the object processing order based on the minimum similarities in the initial approximate k-NNG it builds before processing objects.

Algorithm 1 Indexing in L2Knng

```
1: function INDEX($\boldsymbol{d}_i, \mathcal{I}, se, t_{d_i}$)
2:     $b \leftarrow 1$
3:     for each $j = 1, \ldots, m$, s.t. $d_{i,j} > 0$ and $\sqrt{b} \geq t_{d_i}$ do
4:         $b \leftarrow b - d_{i,j} \times d_{i,j}$
5:         $I_j \leftarrow I_j \cup \{(d_i, d_{i,j}, ||\boldsymbol{d}_i^{>j}||)\}$
6:     end for
7:     $se[d_i] \leftarrow ||\boldsymbol{d}_i^{>j}||$
```

Algorithm 1 details the indexing procedure in L2Knng. The prefix of a vector $\boldsymbol{d}_i$ is indexed while its suffix ℓ^2-norm, computed in b, is above or equal to our threshold t_{d_i} (lines 3-6). The suffix ℓ^2-norm at each indexed feature (line 5) and the suffix ℓ^2-norm of the un-indexed portion of $\boldsymbol{d}_i$ (*suffix estimate*, line 7) are also stored, to be used in other stages of the algorithm. We denote by $\boldsymbol{d}_i^{\leq}$ the indexed prefix of object d_i and by $\boldsymbol{d}_i^{>}$ its un-indexed suffix.

4.3 Pruning the search space

L2Knng searches for neighbors of an object in two stages. During the *candidate generation* stage, L2Knng computes prefix similarities by traversing the index, using an accumulator [19] to keep track of partial dot-products between the query and encountered objects. An accumulator is a map based data structure that accumulates values for given keys. Each object with non-zero accumulated value becomes a *candidate* and is added to a candidate list. Then, during *candidate verification*, L2Knng traverses the list of candidates and finalizes their similarity computations with the query, accumulating suffix similarities for each candidate. In the end, the accumulator will contain the exact similarity with each un-pruned candidate.

A computed similarity can only improve the neighborhood of a query object d_i if it is above σ_{d_i}. Furthermore, it can only improve neighborhoods of already processed objects if it is greater than the minimum of all neighborhood similarities of indexed objects. To keep track of this value, L2Knng could update a heap data structure each time the neighborhood of an indexed object is improved, but we have found this affects overall efficiency. Instead, L2Knng approximates this value by the minimum indexing threshold among all indexed objects, denoted by it, which is strictly smaller than the current minimum of all indexed objects' neighborhood similarities. Using a similar idea as during indexing, L2Knng only starts accumulating for an object d_c while the query suffix ℓ^2-norm is above the lower of these two bounds, $\min(it, \sigma_{d_i})$. Once the suffix ℓ^2-norm falls below this threshold, only index values for objects with non-zero accumulated partial dot-products are processed. Additionally, L2Knng uses the initial approximate k-NNG and the current version of the k-NNG to bypass already computed similarities.

During both the candidate generation and verification stages, there is a further opportunity for pruning when a common feature j is encountered between the query and candidate vectors. To be useful, the final similarity value should improve the neighborhoods of either the query or candidate objects. The accumulator contains the exact similarity of the two prefix vectors, and the similarity of the suffix vectors can be estimated, based on the Cauchy-Schwarz inequality, as upper bounded by the product of their suffix ℓ^2-norms [1]. Thus, a candidate can be pruned if

$$A[d_c] + ||\boldsymbol{d}_i^{>j}|| \times ||\boldsymbol{d}_c^{>j}|| < \min(\sigma_{d_i}, \sigma_{d_c}).$$

L2Knng employs one additional pruning strategy during the candidate verification stage. The $se[d_c]$ suffix estimate value that was stored when indexing the candidate d_c estimates the dot-product between the un-indexed portion of d_c and any other vector in the dataset, $\text{sim}(\boldsymbol{d}_c^{>}, \cdot)$. We use this value here as an estimate for the similarity between the query and candidate suffix, $\text{dot}(\boldsymbol{d}_i, \boldsymbol{d}_c^{>})$. If the sum of the accumulated score and the estimate falls below $\min(\sigma_{d_i}, \sigma_{d_c})$, the candidate is discarded.

Having presented the different pruning bounds used in L2Knng, note that their effectiveness would be greatly reduced without first computing the initial approximate k-NNG. First, indexing thresholds for unprocessed objects would be unknown, and L2Knng would have to index all object features, missing an important pruning opportunity. Similarly, during a search, the algorithm would have to consider all possible candidates with common features, as the minimum indexing threshold it would be 0. Finally, the minimum neighborhood similarities of previously processed objects would likely be smaller, leading to less object pairs being pruned and more neighborhood updates. While L2Knng does not require the initial approximate graph to be computed by L2KnngApprox, an initial graph with high recall will lead to more effective pruning and higher efficiency in constructing the exact graph.

Algorithm 2 delineates the procedure used to find neighbors in L2Knng. The variable r computes the suffix ℓ^2-norm of the query vector, which is used to prevent accumulating similarity for objects that cannot improve neighborhoods (line 6) in the candidate generation stage. At the end of the verification stage, the accumulator contains the exact similarity between the query and objects that survive pruning. These objects are added to the candidate or query neighborhoods if they can improve them.

4.4 Block processing

Algorithm 3 gives an overview of L2Knng. The initial approximate graph k-NNG ($\hat{\mathcal{N}}$, line 2) bootstraps the search framework, providing the necessary processing order for the main loop. As suggested by Bayardo et al. [4], we reorder dimensions in all vectors

Algorithm 2 Searching for neighbors in L2Knng

```
 1: function FindNeighbors($\boldsymbol{d}_i, \mathcal{I}, se, it, \hat{\mathcal{N}}, \mathcal{N}$)
 2:     $r \leftarrow 1;\ A \leftarrow \varnothing$                                ▷ accumulator
 3:     $A[d_c] \leftarrow \emptyset$ for neighbors $d_c$ in $\hat{\mathcal{N}}$ and $\mathcal{N}$
 4:     for each $j = 1, \ldots, m$, s.t. $d_{i,j} > 0$ do        ▷ candidate generation
 5:         for each $(d_c, d_{c,j}, ||\boldsymbol{d}_c^{>j}||) \in I_j$ do
 6:             if $A[d_c] > 0$ or $[A[d_c] \neq \emptyset$ and $\sqrt{r} \geq \min(it, \sigma_{d_i})]$ then
 7:                 $A[d_c] \leftarrow A[d_c] + d_{i,j} \times d_{c,j}$
 8:                 if $A[d_c] + ||\boldsymbol{d}_i^{>j}|| \times ||\boldsymbol{d}_c^{>j}|| < \min(\sigma_{d_i}, \sigma_{d_c})$ then
 9:                     $A[d_c] \leftarrow \emptyset$
10:                 end if
11:             end if
12:         end for
13:         $r \leftarrow r - d_{i,j} \times d_{i,j}$
14:     end for
15:     for each $d_c$ s.t. $A[d_c] > 0$ do                  ▷ candidate verification
16:         next $d_c$ if $A[d_c] + se[d_c] < \min(\sigma_{d_i}, \sigma_{d_c})$
17:         for each $j$ s.t. $d_{c,j}^{>} > 0 \wedge d_{i,j} > 0$ do
18:             $A[d_c] \leftarrow A[d_c] + d_{i,j} \times d_{c,j}$
19:             if $A[d_c] + ||\boldsymbol{d}_i^{>j}|| \times ||\boldsymbol{d}_c^{>j}|| < \min(\sigma_{d_i}, \sigma_{d_c})$ then
20:                 next $d_c$
21:             end if
22:         end for
23:         $N_{d_c} \leftarrow N_{d_c} \cup \{(d_i, A[d_c])\}$ if $A[d_c] > \sigma_{d_c}$
24:         $N_{d_i} \leftarrow N_{d_i} \cup \{(d_c, A[d_c])\}$ if $A[d_c] > \sigma_{d_i}$
25:     end for
```

Algorithm 3 The L2Knng Algorithm

```
 1: function L2Knng($D,\ k,\ \mu,\ \gamma,\ \delta,\ \nu$)
 2:     $\hat{\mathcal{N}} \leftarrow$L2KnngApprox ($D,\ k,\ \mu,\ \gamma,\ \delta$)
 3:     Reorder dimensions in non-decreasing object frequency order
 4:     $\mathcal{N} \leftarrow \hat{\mathcal{N}};\ t_{d_i} \leftarrow \sigma_{d_i},\ it \leftarrow \min(it, t_{d_i}),$ for $i = 1, \ldots, n$
 5:     $I_j \leftarrow \varnothing,\ \mathcal{I} \leftarrow \mathcal{I} \cup I_j,$ for $j = 1, \ldots, m$
 6:     for each $i = 1, 2, \ldots, n$ s.t. $t_{d_i} \leq t_{d_j},\ \forall j > i$ do
 7:         FindNeighbors($\boldsymbol{d}_i, \mathcal{I}, se, it, \hat{\mathcal{N}}, \mathcal{N}$)
 8:         Index($\boldsymbol{d}_i, \mathcal{I}, se, t_{d_i}$)
 9:         $it \leftarrow$ CompleteBlock($i,\ \mathcal{I},\ se,\ \hat{\mathcal{N}},\ \mathcal{N},\ \nu$) if $i\% \frac{|D|}{\nu} = 0$
10:     end for
11: return $\mathcal{N}$
```

in non-decreasing object frequency order as a heuristic way to minimize the inverted index size.

The index keeps growing as more and more objects are processed. The minimum indexing threshold it defined by the initial k-NNG is likely very small, causing the majority of objects in the index to become candidates for each subsequent query. While many candidates will later be eliminated based on pruning bounds that take advantage of continuously updated neighborhood similarities, the delayed pruning can lead to slower execution. L2Knng improves the indexing threshold by periodically "flushing" the index. After completing the k-NNG construction for the already indexed objects, the index can be discarded, speeding up future candidate generation and providing an improved minimum indexing threshold it. Neighborhood construction can be finalized for a block of objects by executing *FindNeighbors* for all un-processed vectors, without indexing them. L2Knng then uses the updated minimum neighborhood similarities of unprocessed objects to define a new processing order. Given a number of blocks parameter ν, L2Knng finalizes a block of indexed objects after processing every $|D|/\nu$ objects.

4.5 L2Knng comparison with APSS

Many of the pruning schemes used by L2Knng are similar in nature with corresponding schemes that were developed to solve the all pairs similarity search (APSS) problem. However, there are a number of key differences between the solutions to the two problems. First, APSS seeks to prune object pairs with a similarity below a threshold t, while L2Knng filters those pairs that cannot improve k-neighborhoods. These distinct goals lead to very different pruning bounds in the two methods. The threshold t is an input to the APSS problem. In our problem, t could be chosen to be the minimum neighborhood similarity σ_{d_i} among all objects in the true k-NNG, which is unknown and would nonetheless be a sub-optimal choice for the k-NNG construction problem. Instead, we devise better thresholds, detailed in Sections 4.2 and 4.3, that can be used in each stage of the method to *safely* prune object pairs that cannot be a part of the true k-NNG. Second, the APSS solutions define an object processing order based on feature weights in each vector, which is not possible in our problem due to the absence of a common indexing threshold for all objects. L2Knng instead processes objects based on the minimum neighborhood similarities of an initial approximate graph, which ensures correctness and leads to higher pruning performance. Finally, we further improve performance by periodically finalizing a set of neighborhoods, which provides better pruning for unprocessed objects.

5. EXPERIMENTAL METHODOLOGY

In this section, we describe the datasets, baseline algorithms, and performance measures used in our experiments.

5.1 Datasets

Table 2: Dataset Statistics

Dataset	n	m	nnz	mrl	mcl
RCV1	804414	45669	62e6	76.5	1347.3
RCV1-400k	400000	45669	31e6	76.5	670.3
RCV1-100k	100000	45669	8e6	78.2	187.4
WW200	1017531	663419	437e6	429.9	659.4
WW500	243223	660600	202e6	830.3	305.7
WW200-250k	250000	663410	108e6	430.3	163.7

For each dataset, n is the number of vectors (rows), m is the number of features (columns), nnz is the number of non-zero values, and mrl and mcl are the mean row and column lengths (number of non-zeros).

We use six text-based datasets to evaluate each method. They represent some real-world and benchmark text corpora often used in text-categorization research. Their characteristics, including number of rows (n), columns (m), and non-zeros (nnz), and mean row/column length (mrl/mcl), are detailed in Table 2. Standard pre-processing, including tokenization, lemmatization, and *tf-idf* weighting, were used to encode text documents as vectors. We present additional details below.

- **RCV1** is a standard benchmark corpus containing over 800,000 newswire stories provided by Reuters, Ltd. for research purposes, made available by Lewis et al. [18].
- **RCV1-100k** and **RCV1-400k** are random subsets of 100,000 and 400,000 documents, respectively, from RCV1.
- **WW500** contains documents with at least 500 distinct features, extracted from the October 2014 article dump of the English Wikipedia[1] (Wiki dump).
- **WW200** contains documents from the Wiki dump with at least 200 distinct features.
- **WW200-250k** is a random subset of size 250,000 from WW200.

5.2 Baseline approaches

We compare our methods against the following baselines.

[1] http://download.wikimedia.org

- `kIdxJoin` is a straight-forward baseline similar to *IDX* in [21] that first builds a full inverted index. Then, without performing any pruning, it uses the index to compute exactly, via accumulation, the similarity of each object with all other objects in the set, returning the top-k matches for each query object.
- `kL2AP` solves the k-NNG problem by executing similarity searches using *L2AP* [1]. We modified *L2AP* to allow specifying a set of input query vectors. Then, as we iteratively reduce the search threshold t, we provide as input only those objects with incomplete neighborhoods.
- *BMM* refers to the docID-oriented with variable block sizes version of the Block-Max Maxscore method by Dimopoulos et al. [9]. The method splits inverted lists into blocks and uses maximum scores for postings in each block to prune the similarity search space. We adapted the method for cosine similarity ranking and chose the same block sizes as in their paper. Blocks were stored in compressed form, using PForDelta compression [25].
- *Maxscore* is an in-memory implementation of the *max_score* information retrieval algorithm [24], as described by Dimopoulos et al. in [9], adapted to rank based on cosine similarity.
- *Greedy Filtering* is a state-of-the-art approach for solving the approximate k-NNG construction problem applied to sparse weighted vectors, proposed by Park et al. [21].
- *NN-Descent* was designed by Dong et al. [11] to work with generic similarity measures and has been shown effective at solving the approximate k-NNG construction problem in both sparse and dense datasets.

While LSH has been a popular method for top-k search, it does not perform well in the k-NNG construction setting. Both *Greedy Filtering* and *NN-Descent* have been shown to outperform LSH when applied to this problem, for k typically ≥ 10. Additionally, *L2AP* outperformed LSH in the related APSS problem. As we will show in Section 6, `L2Knng` significantly outperforms `kL2AP`, the k-NNG method based on *L2AP*, as well as *Greedy Filtering* and *NN-Descent*. As a result, we have chosen not to compare against LSH in this work.

5.3 Performance measures

When comparing approximate k-NNG construction methods, we use average recall to measure the accuracy of the returned result. We obtain the true k-NNG via a brute-force search, then compute the average recall as,

$$R = \frac{1}{|D|} \sum_{d_i \in D} \frac{\text{\# true neighbors in } N_{d_i}}{|N_{d_i}|}.$$

We follow others in using the number of full similarity computations as an architecture and programming language independent way to measure k-NNG construction cost [11, 21]. However, we use a slightly different normalization constant, $NC = |D|(|D| - 1)$, as our `kIdxJoin` baseline does not take advantage of symmetry in similarity computations, and thus may compute up to $n - 1$ similarity values for each vector in the dataset. We report, for all algorithms, *scan rate* $=$ *# similarity evaluations*$/NC$, and *candidate rate* $=$ *# candidates*$/NC$.

An important characteristic in our experiments is CPU runtime, which is measured in seconds. Between a method A and a baseline method B, we report speedup as the ratio of B's execution time and that of A's.

5.4 Execution environment

Our method and all baselines are single-threaded, serial programs. A C++ based library implementing *NN-Descent* can be found at http://www.kgraph.org/. A C based implementation of *L2AP* can be found at http://cs.umn.edu/~dragos/l2ap. We implemented[2] `kIdxJoin`, `kL2AP`, *Greedy Filtering*[3], *Maxscore*, *BMM*, `L2Knng`, and `L2KnngApprox` in C and compiled our program using gcc 4.4.7 with -O3 optimization. Each method was executed on its own node in a cluster of HP ProLiant BL280c G6 blade servers, each with 2.8 GHz Intel Xeon processors and 24 Gb RAM.

We executed each method for $k \in \{1, 5, 10, 25, 50, 75, 100\}$ and tuned parameters to achieve balanced high recall and efficient execution. For all `L2Knng` and `L2KnngApprox` experiments, we set the parameter $\delta = 0.0001$. We tested `kL2AP` by decreasing the threshold t in steps of 0.1, 0.25, and 0.5, and report the best results among the step choices. For the *NN-Descent* library[4], we set $\rho = 1$, $S = 20$, and indexing $K = \mu$ (the candidate list size $\mu \geq k$). For all stochastic methods, we executed a minimum of 5 tries for each set of parameter values and we report averages of all tries.

6. RESULTS & DISCUSSION

We now present our experiment results, along several directions. First, we test `L2KnngApprox` against approximate baselines. We then evaluate the effectiveness of our exact k-NNG building strategies. We measure the influence of the initial approximate graph quality on `L2Knng`'s efficiency and the pruning effectiveness of different stages in the `L2Knng` filtering framework. Finally, we evaluate the runtime and memory scalability of `L2Knng` as the number of input objects increases, and its efficiency as opposed to exact baselines.

6.1 Evaluation of L2KnngApprox

6.1.1 *Candidate pool size parameter analysis*

The efficiency of all the approximate methods under consideration are dependent on the number of candidates they are allowed to consider for each object, μ. The larger the candidate pool is, the more likely the true neighborhood is found among the objects in the pool. We compare the recall and execution time of `L2KnngApprox` with other approximate baselines, given the same candidate list and neighborhood size parameters, μ and k. We tested each method, without changing any other parameters, given $\mu = k, 2 \times k, \ldots, 10 \times k$, on the RCV1-400k and WW200-250k datasets. We tested `L2KnngApprox` with $\gamma = 0$ ($\texttt{L2KnngApprox}_0$), which does not execute any iterative neighborhood updates, and with $\gamma = 3$ ($\texttt{L2KnngApprox}_3$).

Figure 1 plots recall versus execution time for our experiment results. For all methods, results for $\mu = k$ are marked with a "-" label, and those for $\mu = 10 \times k$ with a "+" label. The best results are those points in the lower-right corner of each quadrant in the figure, achieving high recall in a short amount of time. Due to lack of space, we include only results for $k \in \{50, 100\}$. Results for other k values exhibit similar trends.

Methods generally exhibit higher recall and higher execution time for larger μ values. $\texttt{L2KnngApprox}_0$ takes much less time to execute than *Greedy Filtering* and, given large enough

[2]Source code available at http://cs.umn.edu/~dragos/l2knng.

[3]The authors of *Greedy Filtering* kindly provided a Java-based implementation of their algorithm for comparison. On average, our C implementation achieved 1.13x speedup over the Java one.

[4]We thank Wei Dong for his invaluable assistance with using the KGraph library and finding *NN-Descent* evaluation parameters.

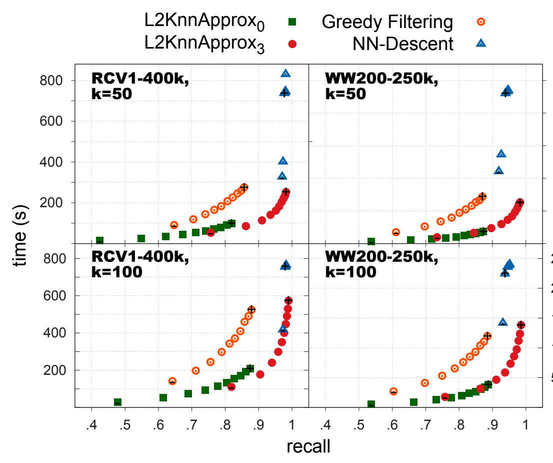

Figure 1: Recall and execution time of approximate methods given increasing candidate pool sizes.

μ, can achieve similar or higher recall. Both `L2KnngApprox` and *Greedy Filtering* require larger μ values than *NN-Descent* to achieve high recall. Yet, *NN-Descent* does not improve much as μ increases. `L2KnngApprox`$_3$ is able to outperform both competitors, with regards to both time and recall, for large enough μ.

6.1.2 L2KnngApprox efficiency

In this work, we focused on building the exact k-NNG. While approximate methods cannot easily achieve perfect recall, we compared their efficiency when seeking a close approximation of the true k-NNG. We executed each approximate method under a wide range of parameters and report the smallest time for which a minimum recall value of 0.95 was achieved. Figure 2 presents execution times for the approximate methods, for four of the datasets. Results for the other datasets are similar and we omit them here due to lack of space. We also include the times for our exact variant, `L2Knng`, as comparison. Note that execution times are log-scaled. Lower values are preferred. The *NN-Descent* result is not included for the WW200 dataset, as we could not obtain high enough recall (the highest recall for $k = 1$ was 0.8854, even with $\mu = 850$).

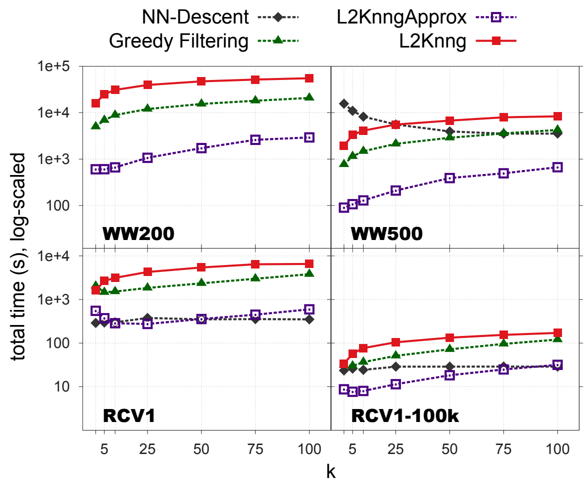

Figure 2: Approximate k-NNG construction efficiency.

`L2KnngApprox` was more efficient than *Greedy Filtering* in all cases and than *NN-Descent* in most cases, while achieving an order of magnitude improvement over our exact solution. For problems where perfect recall is not needed, `L2KnngApprox` can provide a close approximation in much less time. Even though *NN-Descent* had similar execution times as `L2KnngApprox` for some neighborhood sizes of the RCV1 datasets, it performed poorly on the WW datasets. This may be explained by the much higher dimensionality and mean row length of the WW datasets as compared to the RCV1 datasets, which can lead to repeated inclusion of objects in computationally expensive *NN-Descent* local joins. In contrast, `L2KnngApprox` uses several strategies that limit the number of computed dot-products. It builds a higher quality initial graph than *NN-Descent*, prioritizes candidate inclusion, and sets a hard limit on the candidate list size in each iterative update.

6.2 Evaluation of L2Knng

6.2.1 Initial graph influence

While the final recall for an `L2Knng` execution is 1.0, our method uses an initial approximate graph $\hat{\mathcal{N}}$ as a guide in its k-NN search. A graph $\hat{\mathcal{N}}$ with high recall provides `L2Knng` with higher minimum neighborhood similarity values, which translate into tighter pruning bounds and leads to fewer full vector dot-products being computed (smaller scan rate) and faster runtime. We tested the influence of the initial graph quality in three scenarios on the RCV1-400k and WW200-250k datasets. In the first scenario (*random*), we generated an initial graph by randomly picking k neighbors for each object from the set of objects with which they shared at least one feature in common. In the second scenario (*fast*), we chose parameters that ensure fast execution, without guaranteeing high recall ($\mu = k$, $\gamma = 1$). We executed the search using 10 completion blocks in both scenarios ($\nu = 10$). Finally, we include for comparison the best results we achieved after a parameter search (*best*).

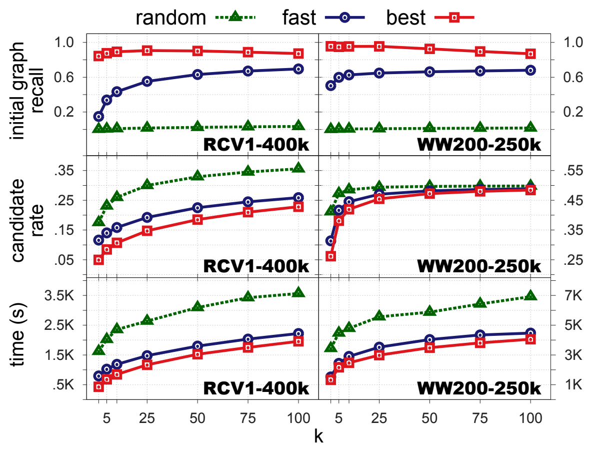

Figure 3: Initial graph influence over `L2Knng` efficiency.

Figure 3 presents our experiment results. The top of the figure shows, for each k value, the recall of the initial graph for the two tested datasets. The middle and bottom of the figure show, for each k value, the candidate rate and execution time after completing the *exact* k-NNG construction. The results emphasize the importance of the initial graph quality in `L2Knng`. The initial graph $\hat{\mathcal{N}}$ for the *random* test case had recall 0.018 and 0.009 on average across all k values for the RCV1-400k and WW200-250k datasets, respectively. The recall was 0.496 and 0.628 in the *fast* and 0.883 and 0.929 in the *best* test cases. These better initial graphs translate into both lower candidate rates and smaller execution times. The *fast* case performed similarly to the *best* case, showing that `L2Knng` can be used with reasonable parameter values and does not require extensive parameter tuning.

6.2.2 Parameter sensitivity

The parameters μ, γ, and ν can influence the effectiveness and efficiency of our exact and approximate algorithms. Larger values for μ increase the number of candidates considered for building the initial graph and will likely lead to increased recall for this stage. Similarly, higher γ values translate to more iterations of initial neighborhood enhancement, at the cost of more similarity computations. Increasing ν values can lead to improved candidate generation pruning and faster index traversal, at the cost of reading vectors in unprocessed blocks several times to find similarities with indexed vectors. There is a trade-off between the benefit of more efficacious pruning bounds and the time taken to achieve them.

We executed parameter sensitivity experiments on the RCV1-400k and WW200-250k datasets, for $k \in \{25, 50, 75, 100\}$. In each experiment, we fixed two of the parameters and varied the third. In the first experiment, given $\gamma = 0$, and $\nu = 10$, we varied μ between 100 and 1000. In the second experiment, given $\mu = 300$, and $\nu = 10$, we varied γ between 0 (no initial neighborhood enhancement) and 10. Finally, to verify the sensitivity of the number of blocks parameter, ν, given $\mu = 300$, and $\gamma = 1$, we varied ν between 1 and 500. Due to lack of space, we include here only a summary of the parameter study results. As expected, we found recall improves when either μ or γ are increased. While the rise is sharp at first, it levels off quickly as the parameter values get larger, showing that the most benefit is gained from checking a relatively small number of initial candidates and executing few neighborhood enhancement rounds. In general, the best results we obtained after parameter tuning were executed with $1 \le \gamma \le 3$ and $300 \le \mu \le 500$. The execution time was not greatly affected as we increased the values of μ or γ, showing that L2Knng is not very sensitive to these parameter choices. We found that increasing the number of blocks ν initially leads to improved performance for all k values. While the improvement is more drastic at first, ν values greater than 50 do not improve the results much, and can eventually lead to decreased efficiency.

6.2.3 *Pruning effectiveness*

L2Knng works by pruning the majority of the candidates that are not true neighbors. Candidates can be pruned while checking the suffix ℓ^2-norm at a common feature during the candidate generation stage (*cg*), during the candidate verification stage (*cv*), or after checking the suffix estimate score (*ses*). Since a partial dot-product is accumulated for each candidate before being pruned, it is important that candidates be pruned as early as possible. In an experiment in which we used fast defaults for all parameters ($\mu = k$, $\gamma = 1$, $\nu = 10$), we counted the number of candidates that were pruned in each stage of the algorithm. Additionally, we display the number of candidates that survived all pruning and had full dot-products computed (*dps*). Figure 4 shows the results of this experiment for the RCV1-400k and WW200-250k datasets, as stacked bar charts showing the number of candidates for each category.

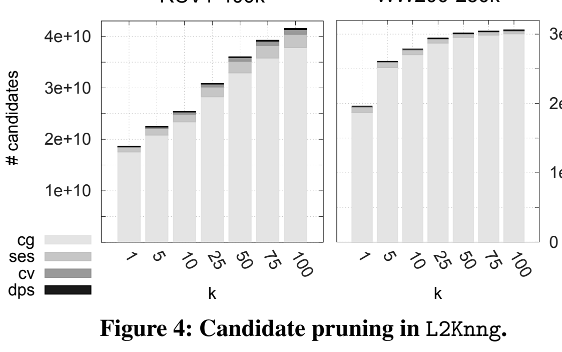

Figure 4: Candidate pruning in L2Knng.

Table 3: Execution time and memory scalability.

Mean search time (ms)			L2Knng				L2KnngApprox			
dataset	# rows	Δ_{sz}	$k = 25$	50	75	100	25	50	75	100
WW200-250k	250000	1.00	12.0	14.0	15.2	16.2	0.7	1.3	1.6	1.9
WW200	1017531	4.07	38.9	46.5	50.9	54.1	1.1	1.7	2.6	2.9
RCV1-100k	100000	1.00	1.0	1.3	1.5	1.7	0.1	0.2	0.2	0.3
RCV1-400k	400000	4.00	2.9	3.8	4.4	4.9	0.3	0.4	0.5	0.6
RCV1	804414	8.04	5.3	6.8	8.0	8.1	0.3	0.4	0.6	0.7

Memory usage (Gb)			L2Knng				L2KnngApprox			
dataset	# rows	Δ_{sz}	$k = 25$	50	75	100	25	50	75	100
WW200-250k	250000	1.00	8.9	9.5	10.0	10.6	7.2	7.8	8.4	9.0
WW200	1017531	4.07	35.9	38.2	40.5	42.8	29.1	31.7	34.0	36.4
RCV1-100k	100000	1.00	0.9	1.1	1.4	1.6	0.7	1.0	1.2	1.4
RCV1-400k	400000	4.00	3.5	4.4	5.3	6.2	2.9	3.8	4.7	5.6
RCV1	804414	8.04	7.0	8.8	10.6	12.4	5.8	7.7	9.5	11.3

Results show that the majority of objects are pruned soon after becoming candidates, in the candidate generation stage (*cg*). Of the remainder, most are pruned by the suffix estimate bound (*ses*), which is checked once, at the beginning of the candidate verification stage, and by additional pruning in the candidate verification stage (*cv*). On average, across all k values, 0.15% and 0.02% of candidates survived all pruning for the RCV1-400k and WW200-250k datasets, respectively. A large number of objects never become candidates in L2Knng, as a result of either the ℓ^2-norm based candidate acceptance bound in the candidate generation stage of the algorithm, or due to the prefix-filtering based index reduction. On average across all k values, only 38.17% and 88.66% of all potential candidates actually became candidates for the RCV1-400k and WW200-250k datasets.

6.2.4 *Scalability testing*

As the dataset size increases, the exact k-NNG problem will take longer to solve, as each object has more potential neighbors that have to be vetted. As a way to verify scalability, we tested our methods on three subsets of the RCV1 and two subsets of the WW200 datasets. For each data subset, Table 3 reports, for $k \in \{25, 50, 75, 100\}$, the mean per-vector search time (top) and the maximum amount of memory used (bottom) for L2Knng and L2KnngApprox. The Δ_{sz} column shows the relative dataset size increase. Parameters were tuned to achieve efficient execution, and, in the case of L2KnngApprox, high recall (95%).

The results show that the performance of both methods scales linearly compared to the dataset size. As the dataset size increases, our two methods perform better than they did for the smaller datasets (e.g., it takes much less than 8.04x the time of searching RCV1-100k to search RCV1 for 100 neighbors), while using memory directly proportional to the number of objects in the set.

6.3 Comparison with other methods

The primary goal of this work is the efficient construction of the exact k-NNG. Figure 5 presents execution times for the exact methods, for all six of the tested datasets. We also include the times for our approximate algorithm, L2KnngApprox, as comparison. Note that execution times are log-scaled, and lower values are preferred. The *Maxscore* and *BMM* experiments on the WW200 and WW500 datasets were terminated early, after executing for 5 days, which is more than twice the execution time of kIdxJoin for these datasets. Additionally, Table 4 shows the average speedup, across all k values, of our algorithms against the best time achieved by competing approximate (left) and exact (right) methods.

L2Knng performed best among all exact methods, achieving over an order of magnitude improvement versus kIdxJoin for small values of k. The speedup is less pronounced as the value of k increases. This may be partially due to an increased number of neigh-

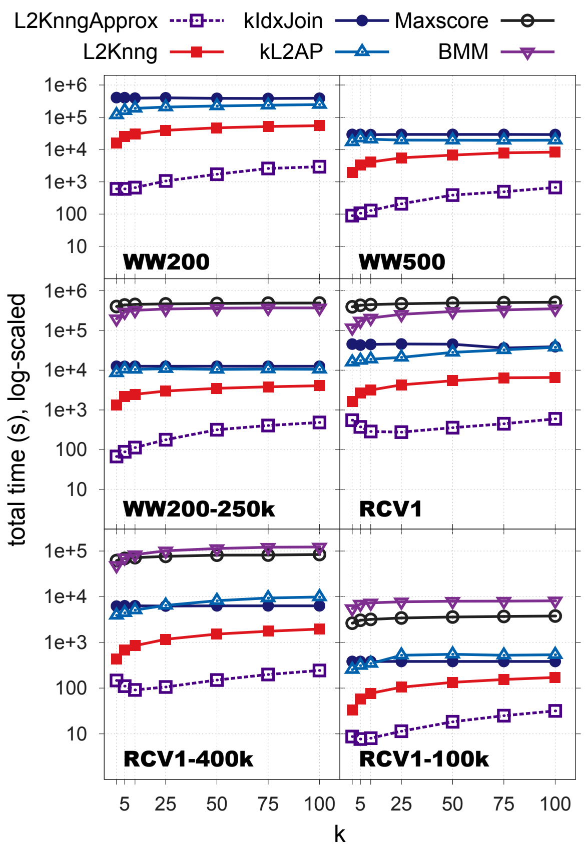

Figure 5: Exact k-NNG construction efficiency comparison.

Table 4: Average speedup of `L2Knng` and `L2KnngApprox` over best competing method.

	versus approx		versus exact	
dataset / method	`L2Knng`	`L2KnngApprox`	`L2Knng`	`L2KnngApprox`
WW200	0.32	9.60	5.57	178.01
WW200-250k	0.41	6.01	3.94	69.08
WW500	0.40	8.68	4.58	110.98
RCV1	0.09	0.87	6.18	61.79
RCV1-100k	0.33	2.19	4.11	27.86
RCV1-400k	0.17	1.19	5.41	40.24

borhood updates during the search, which, in our implementation, incur the cost of heapifying neighborhood data structures.

While using a similar filtering framework as `L2Knng`, `kL2AP` performs poorly, at times taking longer to execute even than `kIdxJoin`, which is equivalent to a brute-force search. This may be due to repeated indexing in `kL2AP` and the size of its final index. As t nears 0, even if we are only interested in finalizing a few neighborhoods, the inverted index lists will contain the majority of values in the dataset, and traversing it will produce many candidates. In contrast, `L2Knng` indexes each vector only once and uses *block completion* as an effective strategy to improve pruning.

Maxscore and *BMM* performed worst among all exact methods, which may be explained by the length of the query vectors used in solving the k-NNG problem. The methods were designed for short queries. They perform a sorting operation with each query and simultaneously traverse as many inverted lists as the number of features in the query vector, which can lead to loosing cache locality. In contrast, our method traverses one inverted list at a time and updates an accumulator in increasing index order, which is much more cache friendly for long queries.

Table 5 presents timing and scan rate results for the three top performing exact and approximate methods. As in Section 6.1.2, we report the smallest time for which a minimum recall value of 0.95 was achieved for all approximate methods. Due to lack of space, we only include results for the WW500 and RCV1 experiments and $k \in \{1, 25, 100\}$. We use bold font to highlight the best result among approximate (top) and exact (bottom) methods, which are separated in the table by a dashed line. `L2Knng` and `L2KnngApprox` achieve most of the lowest scan rates among the competing methods, highlighting the ability of `L2KnngApprox` to find a quality approximate solution using few similarity comparisons and the pruning ability of the `L2Knng` filtering framework. For the RCV1 datasets, *NN-Descent* achieves similar execution times as `L2KnngApprox` while having much higher scan rates. *NN-Descent* does not use accumulation and can take advantage of cache locality afforded by the short vector sizes in the RCV1 datasets. However, much longer vectors, combined with high scan rates, lead to poor *NN-Descent* performance on the WW datasets.

7. CONCLUSIONS AND FUTURE WORK

We presented `L2Knng`, our *exact* filtering-based solution to the cosine similarity k-NNG construction problem. `L2Knng` uses an initial approximate solution graph as a guide to find the desired true neighborhoods, through a modified similarity search framework. We introduced several new pruning bounds specific to this problem, which leverage the Cauchy-Schwarz inequality in partial vector dot-products at each stage in the framework to prevent full similarity computation for most object pairs. `L2Knng` achieves an order of magnitude improvement against exact baselines. Our inexact k-NNG construction method, `L2KnngApprox`, achieves high recall in less time than competing approximate methods, and is an order of magnitude faster than our approximate baselines.

In this paper, we have focused on the *cosine similarity* function for building the k-NNG. It would be interesting to evaluate the efficiency of ℓ^2-norm filtering in the context of other similarity functions, such as the Dice and Tanimoto similarities. Another avenue of research involves scaling up the number of threads and processors used to solve the problem, which in turn will scale the size of the problem that can be efficiently solved.

Acknowledgment

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute. We thank the reviewers for their helpful comments.

Table 5: Execution time and scan rate for competing algorithms. Best results are emphasized in bold.

		WW500			**RCV1**		
result	method / k	1	25	100	1	25	100
time:	*Greedy Filtering*	766.3	2135.5	4239.3	2039.9	1846.0	3809.8
	NN-Descent	15586.8	5562.9	3547.1	**289.6**	377.9	**350.0**
	`L2KnngApprox`	**90.0**	**209.9**	**667.3**	550.1	**275.1**	596.3
	`kIdxJoin`	29389.8	29412.7	29243.9	45456.7	45585.6	38914.4
	`kL2AP`	17201.7	19626.5	19588.1	15823.6	21067.7	37705.9
	`L2Knng`	**1923.2**	**5543.6**	**8340.0**	**1614.8**	**4280.5**	**6550.6**
scan rate:	*Greedy Filtering*	0.0017	0.0045	0.0086	0.0046	0.0034	0.0049
	NN-Descent	1.2913	0.1071	0.8568	0.6805	0.8402	0.6914
	`L2KnngApprox`	**0.0005**	**0.0014**	**0.0045**	**0.0022**	**0.0010**	**0.0018**
	`kIdxJoin`	1.0000	1.0000	1.0000	0.8951	0.8951	0.8951
	`kL2AP`	0.0407	0.4981	0.5003	**0.0003**	0.0249	0.0017
	`L2Knng`	**0.0005**	**0.0011**	**0.0036**	0.0004	**0.0012**	**0.0013**

8. REFERENCES

[1] David C. Anastasiu and George Karypis. L2ap: Fast cosine similarity search with prefix l-2 norm bounds. In *30th IEEE International Conference on Data Engineering*, ICDE '14, 2014.

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[3] Amit Awekar and Nagiza F. Samatova. Fast matching for all pairs similarity search. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '09, pages 295–300, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 131–140, New York, NY, USA, 2007. ACM.

[5] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 97–104, New York, NY, USA, 2006. ACM.

[6] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 426–434, New York, NY, USA, 2003. ACM.

[7] Chris Buckley and Alan F. Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '85, pages 97–110, New York, NY, USA, 1985. ACM.

[8] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 5–, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 113–122, New York, NY, USA, 2013. ACM.

[10] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 993–1002, New York, NY, USA, 2011. ACM.

[11] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM.

[12] Jinyang Gao, Hosagrahar Visvesvaraya Jagadish, Wei Lu, and Beng Chin Ooi. Dsh: Data sensitive hashing for high-dimensional k-nnsearch. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1127–1138, New York, NY, USA, 2014. ACM.

[13] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers.

[14] Andreas Hotho, Andreas Nürnberger, and Gerhard Paaß. A brief survey of text mining. *LDV Forum - GLDV Journal for Computational Linguistics and Language Technology*, 2005.

[15] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[16] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 247–254, New York, NY, USA, 2001. ACM.

[17] Dongjoo Lee, Jaehui Park, Junho Shim, and Sang-goo Lee. An efficient similarity join algorithm with cosine similarity predicate. In *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part II*, DEXA'10, pages 422–436, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, December 2004.

[19] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[20] Rodrigo Paredes, Edgar Chávez, Karina Figueroa, and Gonzalo Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Proceedings of the 5th International Conference on Experimental Algorithms*, WEA'06, pages 85–97, Berlin, Heidelberg, 2006. Springer-Verlag.

[21] Youngki Park, Sungchan Park, Sang-goo Lee, and Woosung Jung. Greedy filtering: A scalable algorithm for k-nearest neighbor graph construction. In *Database Systems for Advanced Applications*, volume 8421 of *Lecture Notes in Computer Science*, pages 327–341. Springer-Verlag, 2014.

[22] Cristian Rossi, Edleno S. de Moura, Andre L. Carvalho, and Altigran S. da Silva. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 183–192, New York, NY, USA, 2013. ACM.

[23] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 175–182, New York, NY, USA, 2007. ACM.

[24] Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, November 1995.

[25] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.