Cosine Approximate Nearest Neighbors

David C. Anastasiu
Department of Computer Engineering
San José State University, San José, CA, USA
Email: david.anastasiu@sjsu.edu

***Abstract*—Cosine similarity graph construction, or all-pairs similarity search, is an important kernel in many data mining and machine learning methods. Building the graph is a difficult task. Up to n^2 pairs of objects should be naïvely compared to solve the problem for a set of n objects. For large object sets, approximate solutions for this problem have been proposed that address the complexity of the task by retrieving most, but not necessarily all, of the nearest neighbors. We propose a novel approximate graph construction method that leverages properties of the object vectors to effectively select few comparison candidates, those that are likely to be neighbors. Furthermore, our method leverages filtering strategies recently developed for exact methods to quickly eliminate unpromising comparison candidates, leading to few overall similarity computations and increased efficiency. We compare our method against several state-of-the-art approximate and exact baselines on six real-world datasets. Our results show that our approach provides a good tradeoff between efficiency and effectiveness, showing up to 35.81x efficiency improvement over the best alternative at 0.9 recall.**

***Index Terms*—Cosine, all-pairs similarity search, nearest neighbors, graph construction, similarity graph.**

I. Introduction

Cosine similarity graph construction, or all-pairs similarity search (APSS), is an important kernel in many data mining and machine learning methods, including ones for pattern recognition [1], online advertising [2], query refinement [3], and collaborative filtering [4]. The goal of APSS is to find, for each object in a set, which is usually referred to as a *query*, all other objects that are sufficiently similar, i.e., those with similarity of at least some threshold ϵ. In this work, we focus on objects that are represented as *non-negative sparse vectors*, which applies to many real-world objects. For example, a social network graph is often represented through its adjacency matrix, where a row encodes the neighborhood of a user and non-negative feature weights are assigned to describe the closeness of the relationship between the user and all other users in the network. Similarly, a book in a library can be represented using the bag-of-words model by a vector of word frequencies.

One way to construct the similarity graph is to compare each query object against all other objects, which we call *candidates*, and filter out those that have similarity below ϵ. However, this will require $n(n-1)/2$ similarity computations and does not scale to large sets of objects. Moreover, many of the candidates will likely be filtered, yet this naïve approach still computes their similarities to the query. A number of methods have been developed in the last decade to reduce the set of candidates. Chaudhuri et al. [5], for example, found that only some of the leading features in each vector (which they call the *prefix* of the vector) had to be considered to find all potential candidates. In other words, if a candidate does not have any non-zero value for a feature in the set of query prefix features, the similarity of that candidate with the query will necessarily be below ϵ and the candidate can be ignored, or pruned. Bayardo et al. [3] used this idea to develop an exact APSS method, AllPairs, which has since been extended by several researchers. In a previous work [6], we gave an overview of these extensions and provided exact and approximate cosine APSS algorithms, L2AP and L2AP-Approx, that significantly outperformed previous methods.

A popular approach for approximate nearest neighbor search has been locality sensitive hashing (LSH). LSH first constructs a search data structure by using families of locality sensitive hash functions, which map similar objects to the same bucket with high probability, to place each object in one or more buckets. Then, at query time, the objects in the buckets that the query maps to will be the candidate set that will be compared with the query. The generic LSH data structures have been found to perform poorly for the APSS problem (see [3], [6]) when expecting high average recall, due to large candidate sets that must be compared with the query. However, Satuluri and Parthasarathy [7] developed a principled Bayesian algorithm (BayesLSH) that uses LSH based estimates to prune away a large majority of the false positive candidates.

Solving a related problem, the k-nearest neighbor graph (kNNG) construction problem, where we are interested in the k objects with the highest similarity to each query, Park et al. [8] discovered that objects with shared high weight features are more likely to be neighbors. Additionally, Dong et al. [9] showed that additional neighbors may be found by considering a neighbor's neighbors as candidates. We combined these ideas [10] to design an approximate kNNG construction method which was used as the first step in an exact kNNG solution. The neighbor similarity concept was also independently used by Malkov et al. [11] to develop a greedy approximate kNNG construction algorithm that works by traversing a small-world neighborhood graph from a random start node.

In this paper, we describe a novel approximate APSS method that leverages properties of the object vectors and their neighbors to effectively select few comparison candidates. Our method works in two steps. First, we leverage the prefix filtering and feature weight priority ideas to quickly construct an approximate min $-\epsilon$ kNNG, while ignoring unsuitable candidates. In the second step, our method traverses the graph to identify candidates, prioritizing those objects that are likely to be a part of the APSS solution. Unlike the naïve approach, the complexity of our method is much smaller than $O(n^2)$, yet it leads to identifying most of the nearest neighbors up to 35.81x faster than the best alternative at 0.9 recall. Our contributions are as follows:

- We propose CANN, a novel approximate algorithm for solving the APSS problem. Unlike previous methods, which use filtering based candidate generation, CANN uses feature and neighborhood graph weights to gather a small set of favorable candidates that will be compared with a query.
- We analyze the properties of neighborhood graphs of six real-world datasets at varying minimum similarity thresholds and draw conclusions on the utility of the APSS output for solving data mining problems.
- We conduct extensive experiments that measure both the effectiveness and efficiency of our method, compared to several state-of-the-art approximate and exact APSS baselines.

The remainder of the paper is organized as follows. We give a formal problem statement and describe our notation in Section II. In Section III, we present our algorithm. In Section IV, we describe the datasets, baseline algorithms, and performance measures used in our experiments. We present our experiment results and discuss their implications in Section V, and Section VI concludes the paper.

II. Problem statement

Given object d_i in a set D of n objects, we seek to find its nearest neighbors, the set of objects in $D \setminus \{d_i\}$ whose cosine similarity with d_i is at least ϵ. The ϵNNG of D is a directed graph $G = (V, E)$ in which vertices correspond to the objects and an edge (v_i, v_j) indicates that the jth object is among the nearest neighbors of the ith object, i.e., their similarity is at least ϵ. An approximate ϵNNG may not contain all the nearest neighbors for each object, yet the present edges will denote actual neighbors. In other words, we assume the similarity between objects is computed exactly and those objects with similarity below ϵ are not considered neighbors.

Borrowing the notation from [10], we will use d_i to indicate the ith object, $\mathbf{d}_i$ 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. Since the cosine function is invariant to changes in vector lengths, we assume that all vectors have been scaled to be of unit length ($||\mathbf{d}_i|| = 1, \forall d_i \in D$), which simplifies computing the similarity between two vectors $\mathbf{d}_i$ and $\mathbf{d}_j$ to their dot-product, which we denote by $\langle \mathbf{d}_i, \mathbf{d}_j \rangle$.

An *inverted index* is a data structure that has been extensively used in Information Retrieval and Data Mining to speed up similarity computations for sparse data. It consists of a set of m lists, $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$, one for each feature, such that list I_j contains pairs $(d_i, d_{i,j})$, where d_i is an indexed object with a non-zero weight for feature j and $d_{i,j}$ is that weight.

Given some dimension p, the *prefix* (vector) $\mathbf{d}_i^{\le p}$ can be thought of as the same vector $\mathbf{d}_i$ with all values for features $j, j > p$, set to 0. The *suffix* vector $\mathbf{d}_i^{>p}$ is analogously defined. Given these definitions, it is easy to verify that the dot-product of a query vector $\mathbf{d}_q$ with a candidate $\mathbf{d}_c$ can be decomposed as the sum of the candidate prefix and suffix dot-products with the query,

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle = \langle \mathbf{d}_q, \mathbf{d}_c^{\le p} \rangle + \langle \mathbf{d}_q, \mathbf{d}_c^{>p} \rangle.$$

III. Constructing the similarity graph

In this section, we will describe our approximate APSS method, CANN. Our method works in two steps, as shown in Algorithm 1. First, CANN leverages features with high weight in object vectors to quickly construct an initial approximate min $-\epsilon$ kNNG. Unlike previous kNNG construction methods, CANN eliminates from consideration the majority of the objects that cannot have a similarity of at least ϵ when building the graph. Limiting the neighborhood size to k allows our method to construct an initial similarity graph while bounding the overall memory usage. In the second step, our method traverses the initial graph to identify candidates for building the ϵNNG, prioritizing objects that are likely to be in the APSS solution. The following subsections discuss the steps in detail.

Algorithm 1 The CANN algorithm

```
1: function CANN ($D$, $\epsilon$, $k$, $\mu_1$, $\mu_2$)
2:     $N \leftarrow InitialGraph\ (D, \epsilon, k, \mu_1)$
3:     $ImproveGraph\ (D, \epsilon, k, \mu_2, N)$
```

A. Min-ϵ kNNG construction

The initial graph construction step is detailed in Algorithm 2. CANN first builds a partial inverted index for the objects, indexing only a few of the leading features in each vector. Then, it uses a sorted version of the inverted lists and of the vectors to prioritize candidates that are selected for comparison with each query.

When choosing candidates to be compared with a query object, CANN takes advantage of the prefix filtering idea [5] as a way to automatically eliminate many objects that cannot be similar enough. As applied in our method, it states that a query object cannot be similar enough if it has no features in common with a candidate in its prefix, given an appropriately chosen prefix. There are several ways to select the prefix features for a vector that have been detailed in the literature [3], [5], [6]. We use a simple method, which we first proposed in [6], that is both effective and efficient to compute. CANN indexes the leading features of each vector until the L2-norm of its suffix falls below the threshold ϵ.

To see why prefix filtering is useful, consider a candidate d_c that has prefix features $\{2, 7, 11\}$ and suffix features $\{12, 15, 19\}$, and a query d_q that has non-zero features $\{1, 9, 12, 15, 19\}$. According to the prefix selection principle, the suffix norm of d_c, $\|\mathbf{d}_c^{>j}\| < \epsilon$. Additionally, note that $\|\mathbf{d}_q\| = 1$, since all vectors are normalized. Therefore, even though the query and candidate have many features in common in the candidate suffix, based on the Cauchy-Schwarz inequality, their suffix dot-product will be below ϵ,

$$\left\langle \mathbf{d}_q, \mathbf{d}_c^{>p} \right\rangle \leq \|\mathbf{d}_q\| \|\mathbf{d}_c^{>p}\| < \epsilon.$$

Since the query does not have any non-zero values for the $\{2, 7, 11\}$ features, thus it has no features in common with the candidate in its prefix, the prefix dot-product $\left\langle \mathbf{d}_q, \mathbf{d}_c^{\leq p} \right\rangle$ will be 0, which means the overall similarity of the vectors will be below ϵ. CANN automatically avoids this object by only choosing candidates for the query d_q from the posting lists in the partial inverted index associated with non-zero features in the query.

Algorithm 2 Min-ϵ kNNG construction in CANN

```
 1: function InitialGraph($D$, $\epsilon$, $k$, $\mu_1$)
 2:     $N_i \leftarrow \varnothing$ for $i = 1, \ldots, n$                ▷ Neighbor lists
 3:     $L \leftarrow \varnothing$                                          ▷ Candidate list
 4:     $T \leftarrow \varnothing$                                          ▷ Processed items
 5:     $H \leftarrow \varnothing$                                          ▷ Query hash table
 6:     for each $q = 1, \ldots, n$ do                                      ▷ Create partial inverted index
 7:         for each $j = 1, \ldots, m$ s.t. $d_{q,j} > 0$ and $\|\mathbf{d}_q^{>j}\| \geq \epsilon$ do
 8:             $I_j \leftarrow I_j \cup \{(d_q, d_{q,j})\}$
 9:     Sort inverted lists in non-increasing value order.
10:     for each $q = 1, \ldots, n$ do
11:         $T[d_c] \leftarrow 1$ for all $(d_c, s) \in N_q$; $l \leftarrow 0$
12:         for each $(j, q_j) \in sorted(\mathbf{d}_q)$ do ▷ non-increasing value order
13:             for each $(d_c, d_{c,j}) \in I_j$ while $l < \mu_1$ do
14:                 if $d_c > d_q$ and not $T[d_c]$ then
15:                     $s \leftarrow BoundedSim(d_q,\ d_c,\ \epsilon)$
16:                     if $s \geq \epsilon$ then
17:                         $L \leftarrow L \cup (d_c, s)$
18:                         if $|N_c| < k$ then
19:                             $N_c \leftarrow N_c \cup (d_q, s)$
20:                     $T[d_c] \leftarrow 1$
21:                 $l \leftarrow l + 1$
22:         Add current neighbors from $N_q$ to $L$.
23:         $N_q \leftarrow$ neighbors with top-k similarity values in $L$.
24: return $\bigcup_{i=1}^{n} N_i$
```

CANN chooses up to μ_1 candidates to compare with each query by iterating through objects in inverted index lists. Based on the idea that objects with high weight features in common with the query are more likely to be neighbors [10], our method prioritizes candidates in each list by first sorting the lists in non-increasing value order. Moreover, CANN chooses list processing order based on the non-increasing query weight value of their associated features. Taking advantage of the commutativity property of cosine similarity, CANN only chooses candidates that follow the query in the processing order. Additionally, it avoids comparing an object multiple times with the query by tagging it as done (1) in a bit-vector data structure (T). When the computed similarity is above ϵ, the candidate is added to the list L, and the query is added to the candidate's neighborhood if its size is below k.

Since each query will be compared against many candidates, CANN uses a hash table to store non-zero query features and their associated suffix norms. This allows dot-products to be computed in a similar way to a sparse-dense vector dot-product (Algorithm 3), iterating only through the non-zero values of the candidate and looking up query values in

Algorithm 3 Bounded similarity computation with pruning

```
1: function BoundedSim($d_q$, $d_c$, $\epsilon$)
2:     $s \leftarrow 0$
3:     for each $j = 1, \ldots, m$ s.t. $d_{c,j} > 0$ do
4:         if $d_{q,j} > 0$ then
5:             $s \leftarrow s + d_{q,j} \times d_{c,j}$
6:             if $s + \|\mathbf{d}_q^{>j}\| \times \|\mathbf{d}_c^{>j}\| < \epsilon$ then
7:                 return -1
8: return $s$
```

the hash table[1]. However, CANN does not fully compute the dot-product in most cases. After each successful multiply-add, it computes an upper-bound estimate on the similarity based on the Cauchy-Schwarz inequality applied to the query and candidate suffix vectors. If what has been computed thus far (s), which amounts to the prefix dot-product of the vectors, plus the suffix similarity estimate is below ϵ, the computation can be safely terminated and the candidate pruned.

B. Candidate selection for similarity search

In its second step, CANN finds the nearest neighbors for each query by traversing the initial kNNG. It first creates an inverted index of the graph's adjacency matrix, which helps avoid re-computing similarities for candidates in whose neighborhoods the query already resides (reverse neighborhood). After tagging both objects in the query's neighborhood and reverse neighborhood, CANN uses a max heap to prioritize neighborhoods it should traverse in search for candidates, namely the neighborhoods of neighbors with high similarity. From those neighborhoods, CANN will pick up to μ_2 candidates, in non-increasing order of their similarity values. Those candidates who are not pruned by the *BoundedSim* function are included in the output. Finally, CANN updates the stored top-k list of neighbors as a way to improve the search for subsequent queries.

Algorithm 4 ϵNNG construction in CANN

```
 1: function ImproveGraph($D$, $\epsilon$, $k$, $\mu_2$, $N$)
 2:     $I \leftarrow Index(N)$                     ▷ Graph adjacency matrix inverted index
 3:     for each $q = 1, \ldots, n$ do
 4:         $L \leftarrow \varnothing$; $T \leftarrow \varnothing$; $H \leftarrow \varnothing$; $l \leftarrow 0$
 5:         $Q \leftarrow \varnothing$                                  ▷ Max heap
 6:         $T[d_c] \leftarrow 1$ and $L \leftarrow L \cup (d_c, s)$ for all $(d_c, s) \in I_q$
 7:         $T[d_c] \leftarrow 1$ for all $(d_c, s) \in N_q$
 8:         $Insert(Q, (d_c, s))$ for all $(d_c, s) \in N_q$
 9:         while $Size(Q) > 0$ do
10:             $(d_c, s) \leftarrow Extract(Q)$
11:             $L \leftarrow L \cup (d_c, s)$
12:             if $l < \mu_2$ then
13:                 for each $(d_b, v) \in N_c$ do
14:                     if not $T[d_c]$ then
15:                         $s \leftarrow BoundedSim(d_q,\ d_c,\ \epsilon)$
16:                         if $s \geq \epsilon$ then
17:                             $Insert(Q, (d_b, s))$
18:                         $T[d_c] \leftarrow 1$
19:             $l \leftarrow l + 1$
20:         Output $d_q$ neighbors in $L$.
21:         $N_q \leftarrow$ neighbors with top-$k$ similarity values in $L$.
```

C. Complexity analysis

CANN pre-computes and stores suffix L2-norms for all non-zero features in the vectors, which takes $O(z)$ time, where z

[1]We omitted the hashing from the algorithms to simplify the presentation.

is the number of non-zeros. The pre-processing and sorting steps are overshadowed by the similarity computations. CANN computes at most $n \times \mu_1$ similarities in step 1 and $n \times \mu_2$ similarities in step 2. Therefore, the overall complexity of CANN is $O(n(\mu_1 + \mu_2)\nu) \ll O(n^2\nu)$, where ν is the average number of non-zeros in a dataset vector.

IV. Experiment setup

A. Methods

We compare our approach, CANN, against our previous exact method, L2AP [6], and two approximate APSS variants, L2AP-Approx [6] (which we denote by L2AP-a) and BayesLSH-Lite [7] (denoted as BLSH-l). Efficient C/C++ based implementations for the baselines were made available by their respective authors. Unlike our method, since it seeks the exact solution, L2AP generates all potential candidates, not just those likely to be in the ϵNNG. It uses a slightly more complex way to determine vector prefixes than our method, but uses the same L2-norm pruning strategy when generating candidates. L2AP uses many different filtering conditions to prune the majority of false-positive candidates, which results in efficient exact ϵNNG construction. BLSH-l uses the same candidate generation strategy as AllPairs [3], but prunes many of the candidates through similarity estimates obtained through Bayesian inference. L2AP-a combines the candidate generation step from L2AP with some of the L2AP verification strategies and the Bayesian pruning in BLSH-l.

B. Datasets

We consider six real-world datasets in our work, with were graciously provided by Venu Satuluri and were also used in [7] and [6]. They represent three text collections (RCV1, WW500k, and WW100k), and three social networks (Twitter, Wiki, Orkut), whose statistics are provided in Table I. Both link and text-based datasets are represented as TF-IDF weighted 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. [12].
- **WikiWords500k** was kindly provided to the authors by Satuluri and Parthasarathy [7], along with the WikiWords100k and WikiLinks datasets. It contains documents with at least 200 distinct features, extracted from the September 2010 article dump of the English Wikipedia[2] (Wiki dump).
- **WikiWords100k** contains documents from the Wiki dump with at least 500 distinct features.
- **TwitterLinks**, first provided by Kwak et al. [13], contains *follow* relationships of a subset of Twitter users that follow at least 1,000 other users. Vectors represent users, and features are users they follow.
- **WikiLinks** represents a directed graph of hyperlinks between Wikipedia articles in the Wiki dump.
- **OrkutLinks** contains the friendship network of over 3M users of the Orkut social media site, made available by Mislove et al. [14]. Vectors represent users, and features are friends of the users. A user could have at most 1000 friends in Orkut.

TABLE I
Dataset Statistics

Dataset	*n*	*m*	*nnz*
RCV1	804,414	43,001	61M
WW500k	494,244	343,622	197M
WW100k	100,528	339,944	79M
Twitter	146,170	143,469	200M
Wiki	1,815,914	1,648,879	44M
Orkut	3,072,626	3,072,441	223M

For each dataset, n is the number of vectors/objects (rows), m is the number of features (columns), and nnz is the number of non-zeros.

C. Execution environment and evaluation measures

Our method and all baselines are serial programs. CANN was implemented in C and compiled with gcc 6.0.1 (-O3 enabled). Each method was executed, without other running programs, on a server with dual-socket 2.8 GHz Intel Xeon X5560 (Nehalem) processors and 24 Gb RAM. We varied ϵ between 0.3 and 0.9, in increments of 0.1. We measure efficiency as the total execution time for the method (wall-clock, in seconds).

We tested each approximate method under a large range of meta-parameters. We tested BLSH-l and L2AP-a by setting the expected false negative rate (ϵ in [7]) to each value in $\{0.01, 0.02, \ldots, 0.09, 0.1, 0.2, \ldots, 1.0\}$. We also varied the number of hashes, h, testing values in $\{128, 256, 384, 512\}$. We tested CANN with k in $\{1, 5, 10, 25, 50, \ldots, 250\}$ and set $\mu_1 = \alpha \times k$, given $alpha \in \{1, 2, \ldots, 10\}$, and $\mu_2 = 5 \times \mu_1$. For each combination of parameters, we executed the method three times and averaged the resulting execution times. We report, at each level of recall, the best execution time for the method given our manual parameter search.

We use average recall to measure the accuracy of the constructed ϵNNG. We obtain the correct ϵNNG via a brute-force search, then compute the average recall as the mean of recall values for each query, were recall is computed as the fraction of (all) relevant/true neighbors that were included by the algorithm in the query's neighborhood.

V. Results & Discussion

Our experiment results are organized along several directions. First, we analyze statistics of the output neighborhood graphs for the real-world datasets we use in our experiments. Then, we examine the effectiveness of our method at choosing candidates and pruning the similarity search space. Finally, we compare the efficiency of our method against existing state-of-the-art exact and approximate baselines.

A. Neighborhood graph statistics

While sparsity of the input vectors plays a big role in the number of objects that must be compared to solve the APSS

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

problem, the number of computed similarities is also highly dependent on the threshold ϵ. We studied properties of the output graph to understand how the input threshold can affect the efficiency of search algorithms. Each non-zero value in the adjacency matrix of the neighborhood graph represents a pair of objects whose similarity must be computed and cannot be pruned. A fairly dense neighborhood graph adjacency matrix means any APSS algorithm will take a long time to solve the problem, no matter how effectively it can prune the search space. Table II shows the average neighborhood size (μ) and neighborhood graph density (ρ) for six of the test datasets and ϵ ranging from 0.1 to 0.9. Graph density is defined here as the ratio between the number of edges (object pairs with similarity at least ϵ) and $n(n-1)$, which is the number of edges in a complete graph with n vertices.

TABLE II
NEIGHBORHOOD GRAPH STATISTICS

ϵ	μ	ρ	μ	ρ	μ	ρ
	WW500k		**WW100k**		**RCV1**	
0.1	1,749	3.5e-03	641	6.4e-03	10,986	1.4e-02
0.2	233	4.7e-04	101	1.0e-03	2,011	2.5e-03
0.3	64	1.3e-04	33	3.3e-04	821	1.0e-03
0.4	25	5.1e-05	16	1.7e-04	355	4.4e-04
0.5	10	2.2e-05	10	1.0e-04	146	1.8e-04
0.6	4.7	9.5e-06	6.3	6.3e-05	57	7.2e-05
0.7	2.1	4.2e-06	4.4	4.3e-05	25	3.2e-05
0.8	0.93	1.9e-06	2.9	2.9e-05	14	1.8e-05
0.9	0.28	5.7e-07	0.96	9.6e-06	8.1	1.0e-05
	Wiki		**Twitter**		**Orkut**	
0.1	801	4.4e-04	875	6.0e-03	76	2.5e-05
0.2	220	1.2e-04	259	1.8e-03	21	6.9e-06
0.3	74	4.1e-05	185	1.3e-03	7.2	2.4e-06
0.4	20	1.1e-05	138	9.5e-04	2.3	7.6e-07
0.5	7.6	4.2e-06	93	6.4e-04	0.69	2.3e-07
0.6	3.3	1.8e-06	49	3.4e-04	0.22	7.2e-08
0.7	1.7	9.6e-07	15	1.1e-04	0.09	3.1e-08
0.8	0.87	4.8e-07	2.8	1.9e-05	0.07	2.1e-08
0.9	0.35	1.9e-07	0.11	7.3e-07	0.06	2.0e-08

The table shows the average neighborhood size (μ) and neighborhood graph density (ρ) for the test datasets and ϵ ranging from 0.1 to 0.9.

As expected, the similarity graph is extremely sparse for high values of ϵ, with less than one neighbor on average in all but one of the datasets at $\epsilon = 0.9$. However, the average number of neighbors and graph density increase disproportionally for the different datasets as ϵ increases. The Orkut objects have less than 10 neighbors on average even at $\epsilon = 0.3$, while the RCV1 objects have more than 100 neighbors on average at $\epsilon = 0.5$. To put things in perspective, the **8.84 billion** edges of the RCV1 neighborhood graph for $\epsilon = 0.1$ take up **204 Gb** of hard drive space, and more than half of those represent similarities below 0.3, which are fairly distant neighbors. Nearest neighbor based classification or recommender systems methods often rely on a small number (often 1-10) of each object's nearest neighbors to complete their task. This analysis suggests that different ϵ thresholds may be appropriate for the analysis of different datasets. Searching the RCV1 dataset using $\epsilon = 0.8$, Twitter using $\epsilon = 0.7$, WW100 and WW500 using $\epsilon = 0.5$, Wiki using $\epsilon = 0.4$, and Orkut using $\epsilon = 0.2$ would provide enough nearest neighbors on average to complete the required tasks.

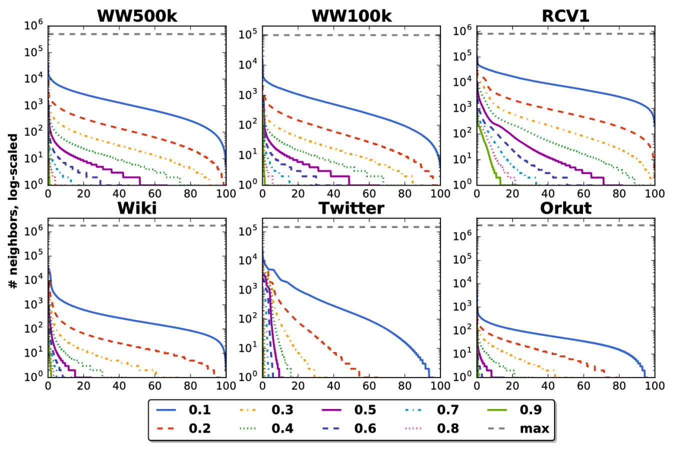

Fig. 1. Neighbor count distributions for several values of ϵ.

Figure 1 gives a more detailed picture of the distribution of neighborhood sizes for the similarity graphs in Table II. The *max* line shows the number of neighbors that would be present in the complete graph. The purple line for $\epsilon = 0.9$ is not visible in the figure for some datasets, due to the extreme sparsity of that graph. The vertical difference between each point on a distribution line and the *max* line represents the potential for savings in filtering methods, i.e., the number of objects that could be pruned without computing their similarity in full. Note that the y-axis is log-scaled. While there is a potential for pruning more than half of the objects in each object search in general, graph datasets show much higher pruning potential, especially given high minimum similarity thresholds.

B. Effectiveness of candidate choice and pruning

We instrumented our code to count the number of object pairs that were considered for similarity computation (# candidates) and the number of full executed dot-product/similarity computations (# dot-products). In Table III, we report the percent of candidates (*cand*) and dot-products (*dps*) executed by our method as opposed to those of a naïve APSS method ($n(n-1)/2$), for each of the six test datasets and ϵ ranging from 0.3 to 0.9. CANN was tuned to achieve 0.9 recall.

The results show that CANN is able to achieve high recall in computing the ϵNNG even while considering much fewer than 1% of the candidates in general. Moreover, most of the candidates are pruned by the L2-norm based filtering in CANN, resulting in 0.01% or fewer of the potential similarities being actually computed. The candidate selection and pruning effectiveness in our method lead to higher efficiency than competing methods, as we show in the next experiment.

C. Execution efficiency

Figure 2 shows the efficiency of the competing methods at different recall levels, for two of the datasets and $\epsilon \in \{0.5, 0.7, 0.9\}$. L2AP is an exact method and is thus only present once in each sub-figure. The execution of both L2AP-a and BLSH-l are dominated by their respective candidate

TABLE III
CANDIDATE CHOICE AND PRUNING EFFECTIVENESS

ϵ	cand	dps	cand	dps	cand	dps
	WW100k		WW500k		RCV1	
0.3	0.2908	0.0380	0.1400	0.0152	0.4040	0.1058
0.4	0.1335	0.0176	0.0488	0.0060	0.2014	0.0521
0.5	0.0931	0.0094	0.0268	0.0022	0.1408	0.0271
0.6	0.0650	0.0045	0.0216	0.0010	0.1165	0.0134
0.7	0.1546	0.0057	0.0209	0.0004	0.0963	0.0058
0.8	0.3505	0.0042	0.0710	0.0002	0.1117	0.0040
0.9	0.3480	0.0012	0.1403	0.0001	0.0864	0.0019
	Twitter		Orkut		Wiki	
0.3	1.2240	0.2905	0.0063	0.0044	0.0194	0.0075
0.4	0.8944	0.1990	0.0045	0.0029	0.0100	0.0031
0.5	0.8007	0.1501	0.0029	0.0018	0.0087	0.0020
0.6	0.5810	0.0852	0.0018	0.0010	0.0055	0.0010
0.7	0.5374	0.0419	0.0009	0.0005	0.0042	0.0006
0.8	0.4131	0.0164	0.0003	0.0002	0.0025	0.0003
0.9	0.4736	0.0070	0.0003	0.0001	0.0020	0.0001

The table shows the percent of potential object comparisons (cand) and computed dot-products (dps) executed by our method as opposed to those of a naïve approach, when tuned to achieve 0.9 recall, for the test datasets and ϵ ranging from 0.3 to 0.9.

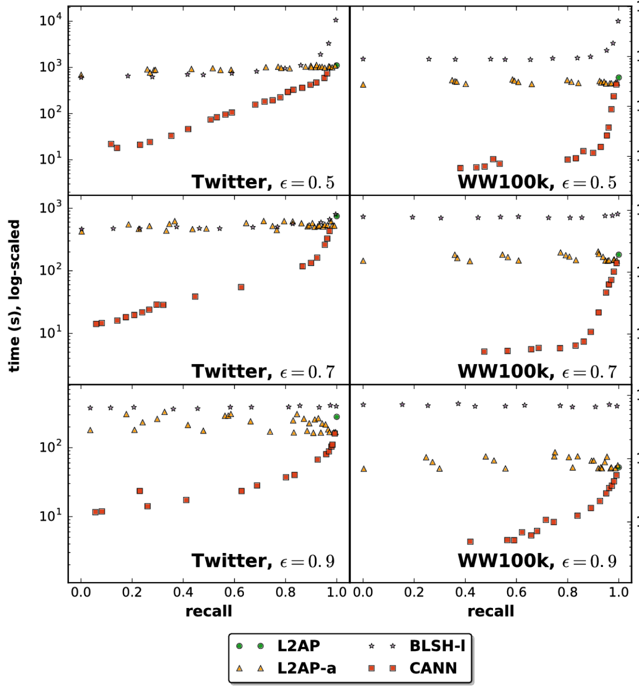

Fig. 2. Efficiency performance of the competing methods at different recall levels.

generation stage, and increasing the allowed error rate in the method seems to do little to improve the overall efficiency. In contrast, our method can be tuned, via the initial neighborhood size k and the candidate list size parameters μ_1 and μ_2, and can achieve over an order of magnitude performance improvement at lower recall levels as opposed to very high recall. CANN is still competitive at very high recall levels (e.g., 0.99) reaching execution times similar to the best exact method, L2AP.

The results of Data Mining algorithms are often not affected by an approximate nearest neighbor graph solution if average recall is sufficiently high [15], e.g., 0.9. As such, we compared the efficiency of all methods for this acceptable recall level and report results in Figure 3, for each of the six test datasets and ϵ between 0.3 and 0.9. As we also reported in [6], L2AP-a performs similarly to L2AP, and the exact method is sometimes faster than the approximate one due to time spent hashing in L2AP-a. The candidate generation in BLSH-l is not as competitive as the one in L2AP, and Bayesian pruning could not overcome the deficit in most cases, making BLSH-l the slowest baseline in general. CANN outperformed all baselines in all experiments except RCV1 at $\epsilon = 0.9$, where the speedup value was 0.97x. Speedup ranged between 0.97x–35.81x for text datasets, and 1.05x–6.18x for network datasets. Given the dataset-specific similarity thresholds suggested in Section V-A, which would result in at least 10 neighbors on average being included in the ϵNNG, CANN achieved an efficiency improvement of 2.1x–35.81x for the different datasets.

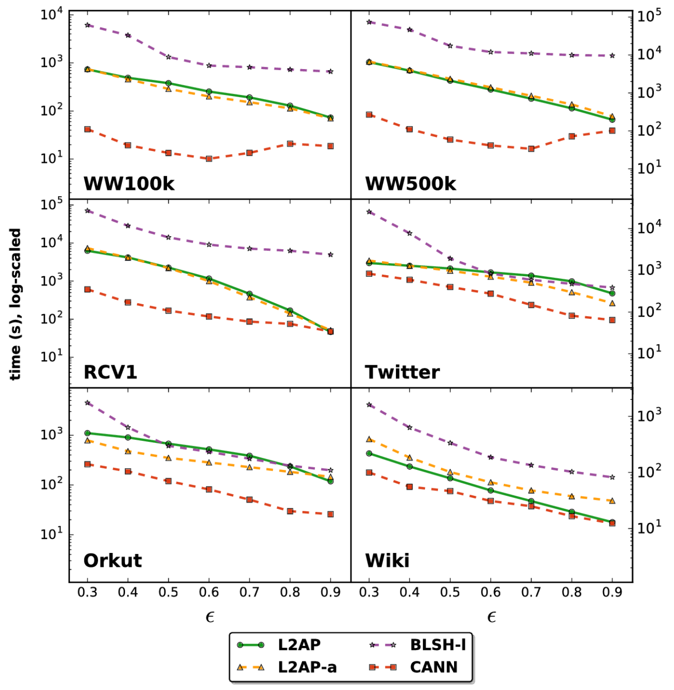

Fig. 3. Execution times for CANN and baselines for ϵ between 0.3 and 0.9, at minimum recall 0.9.

VI. CONCLUSION

In this paper, we presented CANN, an efficient approximate algorithm for constructing the cosine similarity graph for a set of objects. Our method leverages properties of the data and an initial incomplete neighborhood graph to prioritize choosing candidates for a query that are likely to be its neighbors. Furthermore, our method leverages recently developed filtering techniques to prune much of the search space both before and while computing candidate similarities. We conducted extensive experiments that measure both the effectiveness and efficiency of our method, compared to several state-of-the-art approximate and exact similarity graph construction baselines. Our method effectively reduces the number of candidates that should be compared to achieve a certain level or recall, and prunes many of the candidates without fully computing their similarity, resulting in up to 35.81x speedup over the best alternative method.

Acknowledgment

This work was in part made possible due to computing facilities provided by the Digital Technology Center (DTC) and the Minnesota Supercomputing Institute (MSI) at the University of Minnesota. We thank the reviewers for their helpful comments.

References

[1] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," in *Selected papers from the sixth international conference on World Wide Web*. Essex, UK: Elsevier Science Publishers Ltd., 1997, pp. 1157–1166.

[2] A. Metwally, D. Agrawal, and A. El Abbadi, "Detectives: Detecting coalition hit inflation attacks in advertising networks streams," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 241–250.

[3] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 131–140.

[4] G. Karypis, "Evaluation of item-based top-n recommendation algorithms," in *Proceedings of the Tenth International Conference on Information and Knowledge Management*, ser. CIKM '01. New York, NY, USA: ACM, 2001, pp. 247–254.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 5–.

[6] D. C. Anastasiu and G. Karypis, "L2ap: Fast cosine similarity search with prefix l-2 norm bounds," in *30th IEEE International Conference on Data Engineering*, ser. ICDE '14, 2014.

[7] V. Satuluri and S. Parthasarathy, "Bayesian locality sensitive hashing for fast similarity search," *Proc. VLDB Endow.*, vol. 5, no. 5, pp. 430–441, Jan. 2012.

[8] Y. Park, S. Park, S.-g. Lee, and W. Jung, "Greedy filtering: A scalable algorithm for k-nearest neighbor graph construction," in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2014, vol. 8421, pp. 327–341.

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

[10] D. C. Anastasiu and G. Karypis, "L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning," in *24th ACM International Conference on Information and Knowledge Management*, ser. CIKM '15, 2015.

[11] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[12] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, Dec. 2004.

[13] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[14] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. Internet Measurement Conf.*, 2007.

[15] J. Chen, H.-r. Fang, and Y. Saad, "Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection," *J. Mach. Learn. Res.*, vol. 10, pp. 1989–2012, Dec. 2009.