Int J Data Sci Anal (2017) 4:153–172
DOI 10.1007/s41060-017-0064-z

REGULAR PAPER

Efficient identification of Tanimoto nearest neighbors

All-pairs similarity search using the extended Jaccard coefficient

David C. Anastasiu[1] · **George Karypis**[2]

Received: 26 December 2016 / Accepted: 16 July 2017 / Published online: 2 August 2017

Abstract Tanimoto, or extended Jaccard, is an important similarity measure which has seen prominent use in fields such as data mining and chemoinformatics. Many of the existing state-of-the-art methods for market basket analysis, plagiarism and anomaly detection, compound database search, and ligand-based virtual screening rely heavily on identifying Tanimoto nearest neighbors. Given the rapidly increasing size of data that must be analyzed, new algorithms are needed that can speed up nearest neighbor search, while at the same time providing reliable results. While many search algorithms address the complexity of the task by retrieving only some of the nearest neighbors, we propose a method that finds all of the exact nearest neighbors efficiently by leveraging recent advances in similarity search filtering. We provide tighter filtering bounds for the Tanimoto coefficient and show that our method, TAPNN, greatly outperforms existing baselines across a variety of real-world datasets and similarity thresholds.

Keywords Tanimoto similarity · Extended Jaccard · All-pairs similarity search · Nearest neighbors · Graph construction · TAPNN

1 Introduction

Tanimoto, or extended Jaccard, is an important similarity measure which has seen prominent use both in data mining and chemoinformatics. While Strehl and Ghosh note that "there is no similarity metric that is optimal for all applications" [2], Tanimoto was shown to outperform other similarity functions in text analysis tasks such as clustering [3–5], plagiarism detection [6–8], and automatic thesaurus extraction [9]. It has also been successfully used to visualize high-dimensional datasets [2], analyze market basket transactional data [10], recommend items [11], and detect anomalies in spatiotemporal data [12].

In the chemoinformatics domain, data mining and machine learning approaches are increasingly used to boost the effectiveness of the drug discovery process [13]. Fueled by the generally valid premise that structurally similar molecules exhibit similar binding behavior and have similar properties [14], many chemoinformatics methods use the computation of pairwise similarities as a kernel within their algorithms. Virtual screening (VS), for example, uses similarity search, clustering, classification, and outlier detection to identify structurally diverse compounds that display similar bioactivity, which form the starting point for subsequent chemical screening [15].

The numeric representation of chemical compounds has been of great interest to the chemoinformatics community.

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 (DTC) and the Minnesota Supercomputing Institute (MSI).

This paper is an extended version of the DSAA'2016 paper with the same name [1].

✉ David C. Anastasiu
david.anastasiu@sjsu.edu

George Karypis
karypis@cs.umn.edu

[1] Department of Computer Engineering, San José State University, San José, CA, USA

[2] Department of Computer Science and Engineering, University of Minnesota, Twin Cities, MN, USA

Initial studies focused on capturing the presence or absence of features within the compound and represented a compound as a binary or bit vector, referred to as a *fingerprint*. In recent years, frequency (or counting) vectors, which capture how many times a feature is present, and real-valued vectors, called *descriptors*, have gained popularity [13,16]. Arif et al. [17], for example, investigated the use of inverse frequency weighting of features in frequency descriptors for similarity-based VS and found marked increases in screening effectiveness in some circumstances.

In this work, we address the problem of computing pairwise similarities with values of at least some threshold ϵ, also known as the *all-pairs similarity search* (APSS) problem, and focus on objects represented numerically as nonnegative real-valued vectors. Examples of such objects include text documents [18], user and item profiles in recommender systems [11], market basket data [10], and most existing chemical descriptors. We use the Tanimoto coefficient to measure the similarity of two objects.

Within the chemoinformatics community, a great deal of effort has been spent trying to accelerate pairwise similarity computations using the Tanimoto coefficient. Swamidass and Baldi [19] described a number of *bounds* for fast exact threshold-based Tanimoto similarity searches of binary- and integer-based vector representations of chemical compounds. These bounds allow skipping many object comparisons that will theoretically not be similar enough to be included in the result, a technique often referred to as *filtering*, or *pruning*. Other pruning methods relied on hashing techniques [20,21] or tree-based data structures [22,23] to accelerate neighbor searches. However, most recent approaches focus on speeding up chemical searches using inverted index data structures borrowed from information retrieval [20,24,25].

Data mining methods initially designed to efficiently search databases [26] or the Web [27] were later adapted to solve the APSS problem [28]. Most of the existing work addresses either binary vector object representations [29–31] or cosine similarity [32,33]. Nevertheless, Bayardo et al. [28] and Lee et al. [34] show how their cosine filtering-based APSS methods can be extended to the Tanimoto coefficient for binary- and real-valued vectors, respectively. Focusing on real-valued vectors, Kryszkiewicz [35–37] proves several theoretic bounds on the Tanimoto similarity and sketches an inverted index-based algorithm for efficient similarity search.

We describe a new method for Tanimoto APSS of nonnegative real-valued vectors, named TAPNN, which solves the problem *exactly*, finding *all* pairs of objects with a Tanimoto similarity of at least some input threshold ϵ. Our method extends the indexing techniques prevalent in the literature with tighter bounds on the similarity of two vectors, which yield dramatic performance improvements. We experimentally evaluated our method against several baselines on chemical datasets derived from the Molecular Libraries Small Molecule Repository (MLSMR) and the SureChEMBL database, and on text collections comprised of newswire stories and USPTO patents. We show that TAPNN significantly outperforms baselines for both chemical and text datasets. In particular, it was able to find all near-duplicate pairs among 5M SureChemBL chemical compounds in minutes, using a single CPU core, and is over two orders of magnitude more efficient than linear search in general at $\epsilon = 0.99$.

The remainder of the paper is organized as follows. We give a formal problem statement and describe our notation in Sect. 2. In Sect. 3, we present our algorithm. In Sect. 4, we describe the datasets, baseline algorithms, and performance measures used in our experiments. We present our experiment results and discuss their implications in Sect. 5, and Sect. 6 concludes the paper.

2 Problem statement

Given a set of objects $D = \{d_1, d_2, \ldots, d_n\}$, such that each object d_i is represented by a (sparse) nonnegative vector in an m dimensional feature space, and a minimum threshold ϵ on the similarity of two vectors, we wish to find the set of all pairs (d_i, d_j) satisfying $d_i, d_j \in D$, $d_i \neq d_j$, and $\text{sim}(d_i, d_j) \geq \epsilon$ and compute their similarities. Let $\mathbf{d}_i$ indicate the feature vector associated with the ith object and $d_{i,j}$ its value (or weight) for the jth feature. We measure vector similarity as the Tanimoto coefficient for real-valued vectors, computed as,

$$\text{T}(d_i, d_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle}, \tag{1}$$

where $\langle \mathbf{d}_i, \mathbf{d}_j \rangle = \sum_{l=1}^{m} d_{i,l} \times d_{j,l}$ denotes the vector dot-product and $\|\mathbf{d}_i\| = \sqrt{\langle \mathbf{d}_i, \mathbf{d}_i \rangle}$ denotes its Euclidean norm, or length. For a given object d_i, we call an object d_j a *neighbor* of d_i if $\text{sim}(d_i, d_j) \geq \epsilon$.

The majority of feature values in sparse vectors are 0. As a result, a vector $\mathbf{d}_i$ is generally represented as the set of all pairs $(j, d_{i,j})$ satisfying $1 \leq j \leq m$ and $d_{i,j} > 0$. For a set of objects represented by sparse vectors, an *inverted index* representation of the set is made up 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 in the information retrieval literature, where d_i is an indexed object that has a nonzero value for feature j, and $d_{i,j}$ is that value. Postings may store additional statistics related to the feature within the object it is associated with.

The APSS problem seeks, for each object in D, all neighbors with a similarity value of at least ϵ. The similarity graph of D is a graph $G = (V, E)$ where vertices correspond to the objects and an edge (v_i, v_j) indicates that the jth object is

Table 1 Notation used throughout the paper

	Description
D	Set of objects
d_i	The ith object
$\mathbf{d}_i$	Vector representing ith object
$d_{i,j}$	Value for jth feature in $\mathbf{d}_i$
$\mathbf{d}_i^{\leq p}$, $\mathbf{d}_i^{>p}$	Prefix and suffix of $\mathbf{d}_i$ at dimension p
$\mathbf{d}_i^{<}$, $\mathbf{d}_i^{>}$	Un-indexed/indexed portion of $\mathbf{d}_i$
$\hat{\mathbf{d}}_i$	Normalized version of $\mathbf{d}_i$
$\mathcal{I}$	Inverted index
$\mathbf{f}_j$	Vector with jth feature values from all vectors $\hat{\mathbf{d}}_i$
ϵ	Minimum desired similarity

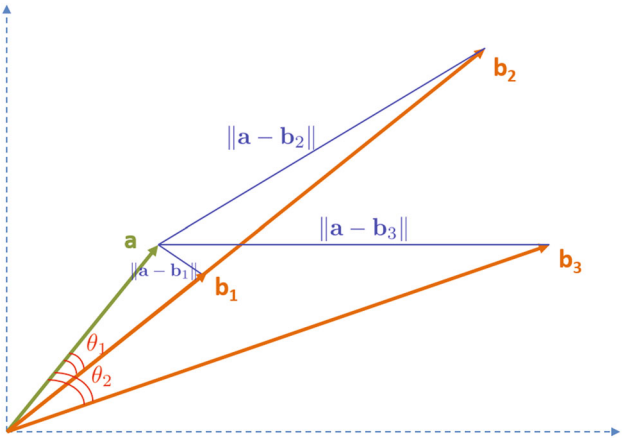

Fig. 1 Comparison of cosine and Euclidean proximity measures

in the neighborhood of the ith object and is associated with a weight, namely the similarity value $\text{sim}(d_i, d_j)$.

Given a vector $\mathbf{d}_i$ and a dimension p, we will denote by $\mathbf{d}_i^{\leq p}$ the vector $(d_{i,1}, \ldots, d_{i,p}, 0, \ldots, 0)$, obtained by keeping the p leading dimensions in $\mathbf{d}_i$, which we call the *prefix* (vector) of $\mathbf{d}_i$. Similarly, we refer to $\mathbf{d}_i^{>p} = (0, \ldots, 0, d_{i,p+1}, \ldots, d_{i,m})$ as the *suffix* of $\mathbf{d}_i$, obtained by setting the first p dimensions of $\mathbf{d}_i$ to 0. Vectors $\mathbf{d}_i^{<p}$ and $\mathbf{d}_i^{\geq p}$ are analogously defined. Table 1 provides a summary of the notation used in this work.

3 Methods

Tanimoto has several advantages that make it ideally suited for measuring proximity in sparse high-dimensional data. It can be efficiently computed via sparse dot-products for asymmetric data, and it takes into consideration both the angle and the length of vectors when indicating their proximity. Consider, for example, the vectors in Fig. 1. When comparing vector $\mathbf{a}$ against $\mathbf{b}_1$ and $\mathbf{b}_2$, cosine similarity reports the cosine of the angle θ_1, which is the same for both $\text{sim}(a, b_1)$ and $\text{sim}(a, b_2)$. On the other hand, the lengths $\|\mathbf{a} - \mathbf{b}_1\|$ and $\|\mathbf{a} - \mathbf{b}_2\|$, denoted by the blue lines with the same labels, are obviously different, showing that Euclidean distance can capture the length difference between $\mathbf{b}_1$ and $\mathbf{b}_2$ in their comparison with $\mathbf{a}$. When comparing $\mathbf{a}$ against $\mathbf{b}_1$ and $\mathbf{b}_3$, however, the lengths $\|\mathbf{a} - \mathbf{b}_1\|$ and $\|\mathbf{a} - \mathbf{b}_3\|$ are identical, and Euclidean distance cannot tell the difference between the two vectors with respect to $\mathbf{a}$. The angles between $\mathbf{a}$ and the two vectors, θ_1 and θ_2, are obviously different, so cosine similarity is able to capture the angle difference between $\mathbf{a}$ and $\{\mathbf{b}_1, \mathbf{b}_3\}$. By definition (Eq. 1), Tanimoto coefficient captures both the angle difference between the two vectors, via the dot-products, and the difference in their lengths, via the square lengths in the denominator.

In certain domains, capturing both angle and length differences can lead to improved performance. Plagiarism detection seeks to find sections of documents that may have been copied from other documents. If a section of a query document was "Veni, vidi, vici. Veni! Vidi! Vici!," it would not be considered as plagiarizing a candidate document containing "Veni, vidi, vici!" if the employed proximity measure was Euclidean distance and the objects were represented as term frequency vectors. However, both Tanimoto and Cosine would be able to identify the sections as very similar and thus a potential plagiarism case. In the chemoinformatics domain, two compounds with very similar proportions of base atoms may be considered quite similar according to Cosine similarity, but may have a very different structure due to the presence of more overall atoms. Both Euclidean distance and Tanimoto coefficient would be able to discern these differences.

Solving the APSS problem is a difficult challenge, requiring $O(n^2)$ similarities to be computed. In the remainder of this section, we show how we can improve search performance by taking advantage of several properties of the problem input, delineated in Fig. 2. In Sect. 3.1, we describe how our method, `TAPNN`, ignores many similarity computations, namely those pairs of objects that do not have any features in common, by leveraging the sparsity structure of the input data. We then demonstrate how, based on the length of each query vector and the input threshold ϵ, our method efficiently ignores many of the remaining potential candidates whose lengths are too short of too long. In Sect. 3.3, we describe how `TAPNN` further ignores many candidates whose angles differ greatly from that of the query. Finally, in Sect. 3.4, we discuss how an upper bound estimate of the angle between a query and a candidate object, in conjunction with the difference in their lengths, can further be used to ignore candidates. The remaining number of object pairs whose similarity is exactly computed is a small portion of the initially considered $O(n^2)$ object pairs, and only slightly larger than the number of *true neighbors*, those in the output of our method.

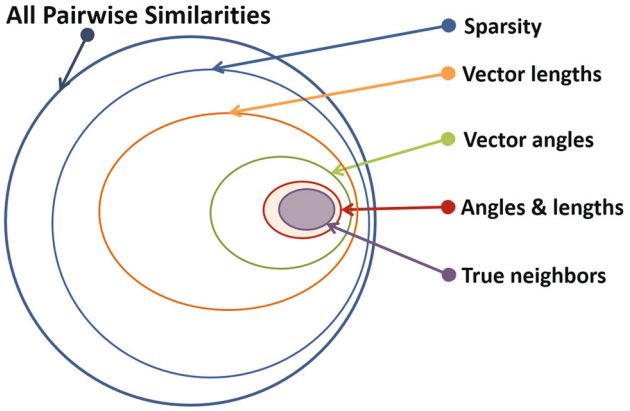

Fig. 2 Pruning strategy in TAPNN

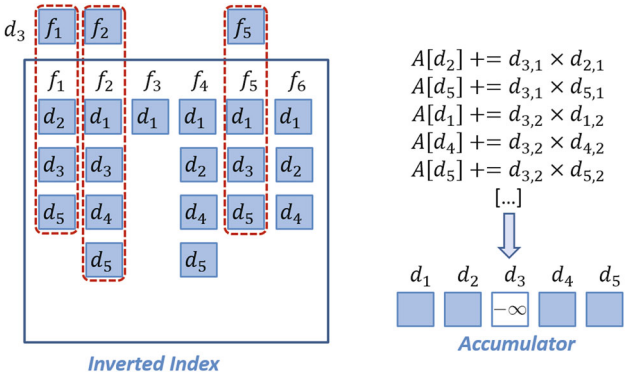

Fig. 3 Using an inverted index and accumulator to compute dot-products

3.1 A basic indexing approach

One approach to find neighbors for a given query object that has been reported to work well in the similarity search literature [20,24,25,28,32–34] has been to use an inverted index, which makes it possible to avoid computing similarities between the query and objects that do not have any nonzero features in common with it. A map-based data structure, called an *accumulator*, can be used to compute the dot-product of the query with all objects encountered while iterating through the inverted lists for nonzero features in the query.

Figure 3 shows how an inverted index and accumulator data structures can be used to compute dot-products for the query object d_3 with all potential neighbors of d_3. We call an object that has a nonzero accumulated dot-product a *candidate*, and forgo computing the query object self-similarity, which is by definition 1. Using precomputed lengths for the object vectors, the dot-products of all candidates can be transformed into Tanimoto coefficients according to Eq. 1 and those coefficients at or above ϵ can be stored in the output.

One inefficiency with this approach is that it does not take advantage of the commutativity property of the Tanimoto coefficient, computing $\text{sim}(d_i, d_j)$ both when accumulating similarities for d_i and for d_j. To address this issue, authors in [28] and [33] have suggested building the index dynamically, adding the query vector to the index only after finding its neighbors. This ensures that the query is only compared against previously processed objects in a given processing order. We suggest a different approach that is equally efficient given modern computer architectures. Given an object processing order, we first re-label each document to match the processing order and then build the inverted index fully, adding objects to the index in the given processing order. The result will be inverted lists sorted in non-decreasing order of document labels. Then, when iterating through each inverted list, we can stop as soon as the encountered document label is greater or equal to that of the query. Since the document label will have already been read from memory to perform the accumulation operation and will be resident in the processor cache, the additional check against the value of the query label will be very fast and will be hidden by the latency associated with loading the next cache line from memory.

3.2 Length-based pruning

Kryszkiewicz [35] has shown that some of the objects whose vector lengths are either too small or too large compared to that of the query object cannot be its neighbors and can thus be ignored. An object d_j cannot be a neighbor of a query object d_i if its length $\|\mathbf{d}_j\|$ falls outside the range $[(1/\alpha)\|\mathbf{d}_i\|,\ \alpha\|\mathbf{d}_i\|]$, where $\|\mathbf{d}_i\|$ is the length of the query vector and

$$\alpha = \frac{1}{2}\left(\left(1+\frac{1}{\epsilon}\right)+\sqrt{\left(1+\frac{1}{\epsilon}\right)^2-4}\right). \tag{2}$$

In Sect. 3.4, we show this bound is actually the limit of a new class of Tanimoto similarity bounds we introduce in this paper. Here, we will show how candidate length pruning can be efficiently integrated into our indexing approach.

A given object will be encountered as many times in the index as it has nonzero features in common with the query. To avoid checking its length against that of the query each time, we could use a data structure, such as a map or bit vector, to mark when a candidate has been checked. While checking this data structure may be less demanding than a multiplication and a comparison, it can actually be slower if the number of candidates is high and the data structure does not fit in the processor cache. Instead, we propose to process objects in non-decreasing vector length order. By relabeling objects as discussed earlier, objects whose lengths are too short will be potentially found at the beginning of the inverted lists, while objects whose lengths are too long can be automatically ignored, as they will come after the query

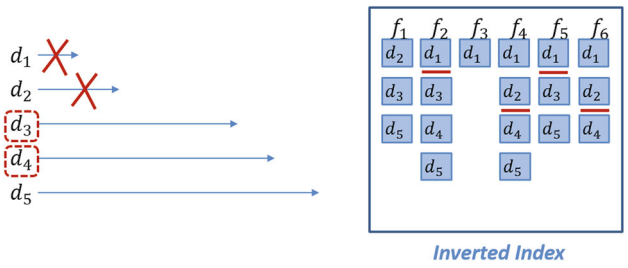

Fig. 4 Efficient length pruning via re-labeling and starting points

object in the processing order. Note also that, for an object d_j following d_i in the processing order,

$$\frac{1}{\alpha}\|\mathbf{d}_j\| \geq \frac{1}{\alpha}\|\mathbf{d}_i\|,$$

since $\|\mathbf{d}_j\| \geq \|\mathbf{d}_i\|$ and both vector lengths and α are nonnegative real values. As such, the label of the maximum candidate that can be ignored will be non-decreasing. Our approach thus uses a list of starting pointers, one for each inverted list, and updates the starting pointer of a list each time a new candidate whose length is too small is found in it.

Figure 4 shows an example of the utility of our processing order re-labeled inverted index, coupled with the use of inverted index starting pointers. In the example, while finding neighbors for objects d_3 and d_4, objects d_1 and d_2 were found to be too short, respectively. The red horizontal lines in the index structure represent the starting pointers in the respective index lists, which were advanced while finding neighbors for d_3 and d_4. When searching for potential neighbor candidates for d_5, objects d_1 and d_2 are automatically ignored by iterating through the inverted lists f_1, f_2, f_4, and f_5 from the current start pointers. In addition to the skipped length check comparison between d_5 and d_1 and d_2, the method also benefits from fewer memory loads by iterating through shorter inverted lists.

Algorithm 1 provides a pseudocode sketch for our basic inverted index-based approach. The method first permutes objects in non-decreasing vector length order and indexes them. Then, for each query object d_q, in the processing order, the maximum object d_{max} satisfying $(1/\alpha)\|\mathbf{d}_{max}\| < \|\mathbf{d}_q\|$ is identified. When iterating through the jth inverted list, TAPNN avoids objects in the list whose lengths have already been determined too small by starting the iteration at index $S[j]$, which is incremented as more objects are found with small lengths. At the end of the accumulation stage, the accumulator contains full dot-products between the query and all objects that could be its neighbors. For each such object, the algorithm computes the Tanimoto coefficient using the dot-product stored in the accumulator and adds the object to the result set if its similarity meets the threshold.

Algorithm 1 TAPNN inverted index approach

```
1: function TAPNN- 1(D, ϵ)
2:     A ← ∅                                          ▷ accumulator
3:     S ← ∅                                          ▷ list starts
4:     N ← ∅                                          ▷ set of neighbors
5:     Compute and store vector lengths for all objects
6:     Permute objects in non-decreasing vector length order
7:     for each q = 1, ..., |D| s.t. ||d_c|| ≤ ||d_q|| ∀c ≤ q do
8:         for each j = 1, ..., m s.t. d_{q,j} > 0 do          ▷ Indexing
9:             I_j ← I_j ∪ {(d_q, d_{q,j})}
10:    for each q = 1, ..., |D| s.t. ||d_c|| ≤ ||d_q|| ∀c ≤ q do
11:        Find label d_max of last object that can be ignored
12:        for each j = 1, ..., m s.t. d_{q,j} > 0 do
13:            for each k = S[j], ..., |I_j| do
14:                (d_c, d_{c,j}) ← I_j[k]
15:                if d_c ≤ d_max then
16:                    S[j] ← S[j] + 1
17:                else if d_c ≥ d_q then
18:                    break
19:                else                                    ▷ Accumulation
20:                    A[d_c] ← A[d_c] + d_{q,j} × d_{c,j}
21:        for each d_c s.t. A[d_c] > 0 do                 ▷ Verification
22:            Scale dot-product in A[d_c] according to Eq. 1
23:            if A[d_c] ≥ ϵ then
24:                N ← N ∪ (d_q, d_c, A[d_c])
25: return N
```

3.3 Incorporating cosine similarity bounds

A number of recent methods have been devised that use similarity bounds to efficiently solve the cosine similarity APSS problem. Moreover, Lee et al. [34] have shown that, for non-negative vectors and the same threshold ϵ, the set of Tanimoto neighbors of an object is actually a subset of its set of cosine neighbors. This can be seen from the formulas of the two similarity functions.

$$\mathrm{T}(d_i, d_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle}$$

$$\mathrm{C}(d_i, d_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|}$$

Given a common numerator, it remains to find a relationship between the denominators in the two functions. Since, for any real-valued vector lengths, $(\|\mathbf{d}_i\| - \|\mathbf{d}_j\|)^2 \geq 0$, it follows that,

$$\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - 2\|\mathbf{d}_i\|\|\mathbf{d}_j\| \geq 0,$$
$$\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \|\mathbf{d}_i\|\|\mathbf{d}_j\| \geq \|\mathbf{d}_i\|\|\mathbf{d}_j\|,$$
$$\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle \geq \|\mathbf{d}_i\|\|\mathbf{d}_j\|,$$

where the last equation follows from the Cauchy–Schwarz inequality, which states that $\langle \mathbf{d}_i, \mathbf{d}_j \rangle \leq \|\mathbf{d}_i\|\|\mathbf{d}_j\|$. As a result, the following relationships can be observed between the cosine and Tanimoto similarities of two vectors,

$$\mathrm{T}(d_i, d_j) \leq \mathrm{C}(d_i, d_j),$$
$$\mathrm{T}(d_i, d_j) \geq \epsilon \Rightarrow \mathrm{C}(d_i, d_j) \geq \epsilon,$$
$$\mathrm{C}(d_i, d_j) < \epsilon \Rightarrow \mathrm{T}(d_i, d_j) < \epsilon.$$

One can then solve the Tanimoto APSS problem by first solving the cosine APSS problem and then filtering out those cosine neighbors that are not also Tanimoto neighbors. Given the computed cosine similarity of two vectors and stored vector lengths, the Tanimoto similarity can be derived as follows.

$$\begin{aligned}\mathrm{T}(d_i, d_j) &= \frac{\frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|}}{\frac{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|}} \\ &= \frac{\frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|}}{\frac{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|} - \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|}}\end{aligned}$$

Applying the definition for cosine similarity, we have

$$\mathrm{T}(d_i, d_j) = \frac{\mathrm{C}(d_i, d_j)}{\frac{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|} - \mathrm{C}(d_i, d_j)}. \tag{3}$$

Note that

$$(\|\mathbf{d}_i\| - \|\mathbf{d}_j\|)^2 \geq 0 \Rightarrow \frac{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|} \geq 2,$$

which provides a higher pruning threshold [34] when searching for cosine neighbors given a Tanimoto similarity threshold ϵ,

$$\begin{aligned}\mathrm{T}(d_i, d_j) \geq \epsilon &\Rightarrow \frac{\mathrm{C}(d_i, d_j)}{2 - \mathrm{C}(d_i, d_j)} \geq \epsilon \\ &\Rightarrow \mathrm{C}(d_i, d_j) \geq \frac{2\epsilon}{1 + \epsilon} = t\end{aligned} \tag{4}$$

Unlike the Tanimoto coefficient, cosine similarity is length invariant. Vectors can thus be normalized as a preprocessing step, which reduces cosine similarity to the dot-product of the normalized vectors. Denoting by $\hat{\mathbf{d}}_i = \mathbf{d}_i / \|\mathbf{d}_i\|$, the normalized version of the ith object vector,

$$\mathrm{C}(d_i, d_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|} = \langle \hat{\mathbf{d}}_i, \hat{\mathbf{d}}_j \rangle.$$

This step, in fact, reduces the number of floating point operations needed to solve the problem and is standard in cosine APSS methods. Note that the method outlined in Algorithm 1 can also be applied to normalized vectors, adding only a normalization step before indexing and replacing the scaling factor in line 22, using Eq. 3 instead of Eq. 1.

In a recent work [33], we described a number of cosine similarity bounds based on the ℓ^2-norm of prefix or suffix vectors that have been found to be more effective than previous known bounds for solving the cosine APSS problem. It may be beneficial to incorporate this type of filtering in our method. However, some of the bounds we described in that work rely on a different object processing order. Our method, therefore, uses similar ℓ^2-norm-based bounds that are processing order independent. This allows our method to still take advantage of the vector length-based filtering described in Sect. 3.2. In the remainder of this section, we will describe the ℓ^2-norm-based filtering in our method.

Normalized vector prefix ℓ^2-norm-based filtering

Given a fixed feature processing order and the prefix and suffix of a query object at feature p, it is easy to see that

$$\begin{aligned}\langle \hat{\mathbf{d}}_q, \hat{\mathbf{d}}_c \rangle &= \langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle + \langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \rangle \\ &\leq \|\hat{\mathbf{d}}_q^{\leq p}\| \|\hat{\mathbf{d}}_c\| + \langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \rangle,\end{aligned}$$

where the inequality follows from applying the Cauchy–Schwarz inequality to the prefix dot-product. Since the maximum value of $\|\hat{\mathbf{d}}_c\|$ is 1, the prefix dot-product can further be upper-bounded by the length of the prefix vector,

$$\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle \leq \|\hat{\mathbf{d}}_q^{\leq p}\|. \tag{5}$$

Another bound on the prefix dot-product can be obtained by considering the maximum values for each feature among all normalized object vectors. Let $\mathbf{f}_j$ denote the vector of all feature values for the jth feature within the normalized vectors and $\mathbf{mx}$ the vector of maximum such feature values for each dimension, defined as,

$$\mathbf{f}_j = (\hat{d}_{1,j}, \hat{d}_{2,j}, \ldots, \hat{d}_{n,j}),$$
$$\mathbf{mx} = (\|\mathbf{f}_1\|_\infty, \|\mathbf{f}_2\|_\infty, \ldots, \|\mathbf{f}_m\|_\infty).$$

Then,

$$\begin{aligned}&\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle = \sum_{l=1}^{m} d_{q,l} \times d_{c,l} \leq \sum_{l=1}^{m} d_{q,l} \times mx_l \Rightarrow \\ &\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle \leq \langle \hat{\mathbf{d}}_q^{\leq p}, \mathbf{mx} \rangle.\end{aligned} \tag{6}$$

Combining the bounds in Eqs. 5 and 6, we obtain a bound on the prefix similarity of a vector with any other object in D, which we denote by $ps_q^{\leq p}$,

$$\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle \leq ps_q^{\leq p} = \min(\|\hat{\mathbf{d}}_q^{\leq p}\|, \ \langle \hat{\mathbf{d}}_q^{\leq p}, \mathbf{mx} \rangle). \tag{7}$$

We define $ps_q^{<p}$ analogously.

Algorithm 2 describes how we incorporate cosine similarity bounds within our method. Following examples in [28] and [33], we use the ps bound to index only a few of the nonzeros in each object. Note that, if $ps_q^{<p} < t$, with t defined as in Eq. 4, and an object d_c has no features in common with the query in lists I_j, $p \leq j \leq m$, then its cosine similarity to the query will be below t, and its Tanimoto similarity will then be below ϵ. Conversely, if $\langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \rangle > 0$, the object may potentially be a neighbor. By indexing values in each query vector starting at the index p satisfying $ps_q^{\leq p} \geq t$, and then iterating through the index and accumulating, the nonzero values in the accumulator will contain only the suffix dot-products, $\langle \hat{\mathbf{d}}_q, \hat{\mathbf{d}}_c^{>} \rangle$, where $d_c^{>}$ represents the indexed suffix for some object d_c found in the index. Once some value has been accumulated for an object, we refer to it as a *candidate*. This portion of the method can be thought of as *candidate generation* (CG) and is similar in scope to the screening phase of many compound search methods in the chemoinformatics literature. Our method uses the un-indexed portion of the candidate, $d_c^{\leq}$, to complete the dot-product computation during the verification stage, before the scaling and threshold checking steps. We call this portion of the method, which is akin to the verification stage in other chemoinformatics methods, *candidate verification* (CV).

Our method adopts a non-increasing inverted list size (object frequency) order for processing features, which heuristically leads to shorter lists in the inverted index. The *partial indexing* strategy presented in the previous paragraph improves the efficiency of our method in two ways. First, objects that have nonzero values in common with the query only in the un-indexed set of query features will be automatically ignored. Our method will not encounter such an object in the index when generating candidates for the query and will thus not accumulate a dot-product for it. Second, the verification stage will require reading from memory only those sparse vectors for un-pruned candidates, iterating through fewer nonzeros in general than exist in the un-indexed portion of all objects.

We use the ps bound in two additional ways to improve the pruning effectiveness of our method. First, when encountering a new potential object in the index during the CG stage ($A[d_c] = 0$), we only accept it as a candidate if $ps_q^{\leq j} \geq t$. Note that we process index lists in reverse feature processing order in the CG and CV stages, and $A[d_c]$ contains the exact dot-product $\langle \hat{\mathbf{d}}_q, \hat{\mathbf{d}}_c^{>j} \rangle$. Therefore, if $A[d_c] = 0$ and $ps_q^{\leq j} < t$, the candidate cannot be a neighbor of the query object. Second, as a first step in verifying each candidate, we check whether $ps_c^{<}$, the ps bound of the candidate at its last indexed feature (line 10 in Algorithm 2), added to the accumulated suffix dot-product, is equal or greater than the threshold t. The value $ps_c^{<}$ is an upper bound of the dot-product of the un-indexed prefix of the candidate vector with any other vector in the dataset. Thus, the candidate can be safely pruned if the check fails.

Algorithm 2 TAPNN with cosine bounds

```
1: function TAPNN- 2(D, ϵ)
2:     A ← ∅, S ← ∅, N ← ∅
3:     t ← 2ϵ/(1 + ϵ)
4:     Compute and store vector lengths for all objects
5:     Permute objects in non-decreasing vector length order
6:     for each q = 1, ..., |D| s.t. ‖d_c‖ ≤ ‖d_q‖ ∀c ≤ q do
7:         Normalize d_q
8:         for each j = 1, ..., m s.t. d̂_{q,j} > 0 and ps_q^{≤p} ≥ t do
9:             I_j ← I_j ∪ {(d_q, d̂_{q,j}, ‖d̂_q^{<j}‖)}                ▷ Indexing
10:        Store ps_q^{<}
11:    for each q = 1, ..., |D| s.t. ‖d_c‖ ≤ ‖d_q‖ ∀c ≤ q do
12:        Find label d_max of last object that can be ignored
13:        for each j = m, ..., 1 s.t. d̂_{q,j} > 0 do                  ▷ CG
14:            for each k = S[j], ..., |I_j| do
15:                (d_c, d_{c,j}) ← I_j[k]
16:                if d_c ≤ d_max then
17:                    S[j] ← S[j] + 1
18:                else if d_c ≥ d_q then
19:                    break
20:                else if A[d_c] > 0 or ps_q^{≤j} ≥ t then
21:                    A[d_c] ← A[d_c] + d̂_{q,j} × d̂_{c,j}
22:                    Prune if A[d_c] + ‖d̂_q^{<j}‖‖d̂_c^{<j}‖ < t
23:        for each d_c s.t. A[d_c] > 0 do                              ▷ CV
24:            Prune if A[d_c] + ps_c^{<} < t
25:            for each j = m, ..., 1 s.t. d̂_{c,j}^{≤} > 0 and d_{q,j} > 0 do
26:                A[d_c] ← A[d_c] + d̂_{q,j} × d̂_{c,j}
27:                Prune if A[d_c] + ‖d̂_q^{<j}‖‖d̂_c^{<j}‖ < t
28:            Scale dot-product in A[d_c] according to Eq. 3
29:            if A[d_c] ≥ ϵ then
30:                N ← N ∪ (d_q, d_c, A[d_c])
31: return N
```

As in our cosine APSS method [33], after each accumulation operation, in both the CG and CV stages of the algorithm, we check an additional bound, based on the Cauchy–Schwarz inequality. The objects cannot be neighbors if the accumulated suffix dot-product, added to the upper bound $\|\hat{\mathbf{d}}_q^{<j}\|\|\hat{\mathbf{d}}_c^{<j}\|$ of their prefix dot-product, cannot meet the threshold t. We have tested a number of additional candidate verification bounds described in the literature based on vector number of nonzeros, prefix lengths, or prefix sums of the vector feature values, but have found them to be less efficient to compute and in general less effective than our described cosine pruning in a variety of datasets. The interested reader may consult [28,32–34] for details on additional verification bounds for cosine similarity.

3.4 New Tanimoto similarity bounds

Up to this point, we have used pruning bounds based on the lengths of the un-normalized vectors and prefix ℓ^2-norms of the normalized vectors to either ignore outright or stop

considering (prune) those objects that cannot be neighbors for a given query. We will now present new Tanimoto-specific bounds which combine the two concepts to effect additional pruning. First, we will describe a bound on the prefix length of an un-normalized candidate vector, which we use during candidate generation. Then, we will introduce a bound for the length of the un-normalized candidate vector that relies on cosine similarity estimates we compute in our method.

A bound on the prefix length of an un-normalized candidate vector

Recall that the dot-product of a query with a candidate vector can be decomposed as the sum of its prefix and suffix dot-products, which can be written as a function of the respective normalized vector dot-products as,

$$\begin{aligned}\langle \mathbf{d}_q, \mathbf{d}_c \rangle &= \langle \mathbf{d}_q^{\leq p}, \mathbf{d}_c \rangle + \langle \mathbf{d}_q^{>p}, \mathbf{d}_c \rangle \\ &= \langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle \|\mathbf{d}_q^{\leq p}\| \|\mathbf{d}_c\| + \langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \rangle \|\mathbf{d}_q^{>p}\| \|\mathbf{d}_c\|.\end{aligned}$$

For an object that has not yet become a candidate ($A[d_c] = 0$), $\langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \rangle = 0$, simplifying the expression to,

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle = \langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle \|\mathbf{d}_q^{\leq p}\| \|\mathbf{d}_c\|.$$

From the expression $\mathrm{T}(d_c, d_q) \geq \epsilon$, substituting the Tanimoto formula in Eq. 1, we can derive,

$$\begin{aligned}\langle \mathbf{d}_q, \mathbf{d}_c \rangle &\geq \frac{\epsilon}{1+\epsilon} \left(\|\mathbf{d}_q\|^2 + \|\mathbf{d}_c\|^2 \right) \\ \|\mathbf{d}_q^{\leq p}\| &\geq \frac{\epsilon}{1+\epsilon} \frac{\|\mathbf{d}_q\|^2 + \|\mathbf{d}_c\|^2}{\|\mathbf{d}_c\| \langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle} \\ \|\mathbf{d}_q^{\leq p}\| &\geq \frac{\epsilon}{1+\epsilon} \frac{\|\mathbf{d}_q\|^2 + \|\mathbf{d}_1\|^2}{\|\mathbf{d}_{q-1}\| \; ps_q^{\leq j}} \end{aligned} \tag{8}$$

Equation 8 replaces the prefix dot-product $\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle$ with the ps upper bound, which represents the dot-product of the query with any potential candidate. Furthermore, taking advantage of the pre-defined object processing order in our method, we replace the numerator candidate length by that of the object with minimum length (the first processed object, d_1) and the denominator candidate length with that of the object with maximum length (the last processed object, d_{q-1}). Since $\|\mathbf{d}_1\|^2 \leq \|\mathbf{d}_c\|^2$, $\|\mathbf{d}_{q-1}\| \geq \|\mathbf{d}_c\|$, and $ps_q^{\leq j} \geq \langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \rangle$, the inequality holds.

We use the bound in Eq. 8 during the candidate generation stage of our method as a potentially more restrictive condition for accepting new candidates. It complements the ps bound in line 20 in Algorithm 2, which checks whether new candidates can still be neighbors based only on the prefix of the normalized query vector. Once the prefix length of the query un-normalized vector falls below the bound in Eq. 8, objects that have not already been encountered in the index can no longer be similar enough to the query.

A tighter bound for the un-normalized candidate vector length

Let $\beta = \|\mathbf{d}_c\| / \|\mathbf{d}_q\|$, and, for notation simplicity, $s = \langle \hat{\mathbf{d}}_q, \hat{\mathbf{d}}_c \rangle = \mathrm{C}(d_i, d_j)$. Given $\mathrm{T}(d_q, d_c) \geq \epsilon$, and the pre-imposed object processing order (i.e., $\|\mathbf{d}_q\| \geq \|\mathbf{d}_c\|$), we derive β as a function of the cosine similarity of the objects, starting from Eq. 3,

$$\begin{aligned}\mathrm{T}(d_q, d_c) &= \frac{\mathrm{C}(d_q, d_c)}{\frac{\|\mathbf{d}_q\|^2 + \|\mathbf{d}_c\|^2}{\|\mathbf{d}_q\| \|\mathbf{d}_c\|} - \mathrm{C}(d_q, d_c)} \geq \epsilon \\ \frac{s\|\mathbf{d}_q\| \|\mathbf{d}_c\|}{\|\mathbf{d}_q\|^2 + \|\mathbf{d}_c\|^2 - s\|\mathbf{d}_q\| \|\mathbf{d}_c\|} &\geq \epsilon \\ \epsilon\|\mathbf{d}_c\|^2 - s(1+\epsilon)\|\mathbf{d}_c\| \|\mathbf{d}_q\| - \epsilon\|\mathbf{d}_q\|^2 &\leq 0 \\ \epsilon\beta^2 - s(1+\epsilon)\beta - \epsilon &\leq 0 \\ \beta = \frac{s(1+\epsilon)}{2\epsilon} + \sqrt{\left(\frac{s(1+\epsilon)}{2\epsilon}\right)^2 - 1} \\ = \frac{s}{t} + \sqrt{\left(\frac{s}{t}\right)^2 - 1}\end{aligned} \tag{9}$$

Replacing s with any of the upper bounds on the cosine similarity we described in Sect. 3.3, the bound in Eq. 9 allows us to prune any candidate whose length is less than $\|\mathbf{d}_q\|/\beta$. Note that, for $s = 1$, which is the upper limit of the cosine similarity of nonnegative real-valued vectors, $\beta = \alpha$, which is the bound introduced by Kryszkiewicz [35] for length-based pruning of candidate vectors. In the presence of an upper bound estimate of the cosine similarity for two vectors, our bound provides a more accurate estimate of the minimum length a candidate vector must have to potentially be a neighbor for the query.

In Algorithm 3, we present pseudocode for the `TAPNN` method, which includes all the pruning strategies we described in Sect. 3. The symbol *EQ*8 in line 12 refers to checking the query prefix vector length, according to Eq. 8. While our bound β for the un-normalized candidate vector length could be checked each time we have a better estimate of the cosine similarity of two vectors, after each accumulation operation, it is more expensive to compute than the simpler prefix ℓ^2-norm cosine bound. We thus check it only twice for each candidate object, first after computing the cosine estimate based on the candidate ps bound (line 17) and again after accumulating the first un-indexed feature in the candidate (line 22). We have found this strategy works well in practice.

Algorithm 3 TAPNN algorithm

function TAPNN(D, ϵ)
  Lines 2–10 in Algorithm 2
  for each $q = 1, \ldots, |D|$ s.t. $\|\mathbf{d}_c\| \leq \|\mathbf{d}_q\|\ \forall c \leq q$ **do**
    Find label d_{max} of last object that can be ignored
    for each $j = m, \ldots, 1$ s.t. $\hat{d}_{q,j} > 0$ **do** ▷ CG
      for each $k = S[j], \ldots, |I_j|$ **do**
        $(d_c, d_{c,j}) \leftarrow I_j[k]$
        if $d_c \leq d_{max}$ **then**
          $S[j] \leftarrow S[j] + 1$
        else if $d_c \geq d_q$ **then**
          break
        else if $A[d_c] > 0$ or $[ps_q^{\leq j} \geq t$ **and** $EQ8]$ **then**
          $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j} \times \hat{d}_{c,j}$
          Prune if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\| \|\hat{\mathbf{d}}_c^{<j}\| < t$
    for each d_c s.t. $A[d_c] > 0$ **do** ▷ CV
      Prune if $A[d_c] + ps_c^{<} < t$
      Compute β given $s = A[d_c] + ps_c^{<}$
      Prune if $\|\mathbf{d}_c\| \times \beta < \|\mathbf{d}_q\|$
      Find first j s.t. $\hat{d}_{c,j}^{\leq} > 0$ and $d_{q,j} > 0$
      $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j} \times \hat{d}_{c,j}$
      Prune if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\| \|\hat{\mathbf{d}}_c^{<j}\| < t$
      Compute β given $s = A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\| \|\hat{\mathbf{d}}_c^{<j}\|$
      Prune if $\|\mathbf{d}_c\| \times \beta < \|\mathbf{d}_q\|$
      for each $j = \ldots, 1$ s.t. $\hat{d}_{c,j}^{\leq} > 0$ and $d_{q,j} > 0$ **do**
        $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j} \times \hat{d}_{c,j}$
        Prune if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\| \|\hat{\mathbf{d}}_c^{<j}\| < t$
      Scale dot-product in $A[d_c]$ according to Eq. 3
      if $A[d_c] \geq \epsilon$ **then**
        $N \leftarrow N \cup (d_q, d_c, A[d_c])$
return N

4 Materials

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

4.1 Datasets

We evaluate each method using several real-world and benchmark text and chemical compound corpora. Their characteristics, including number of rows (n), columns (m), nonzeros (nnz), and mean row/column length (μ_r/μ_c), are detailed in Table 2.

1. **Patents-8.8M** is a random subset of 8.8M patent documents from all US utility patents.[1] Each document contains the patent title, abstract, and body. **Patents-4M**, **Patents-2M**, **Patents-1M**, **Patents-500K**, **Patents-250K**, and **Patents-100K** are random subsets of 4E+6, 2E+6, 1E+6, 5E+5, 2.5E+5, and 1E+5 patents, respectively, from the **Patents-8.8M** dataset. Most of our experiments used the Patents-100K dataset, which we had readily available. We later processed the larger Patents datasets and included them in our scaling experiments.
2. **RCV1** is a standard text processing benchmark corpus containing over 800,000 newswire stories from Reuters, Ltd [38].
3. **MLSMR** [39] (Molecular Libraries Small Molecule Repository) is a collection of structures of compounds accepted into the repository of PubChem, a database of small organic molecules and their biological activity curated by the National Center for Biotechnology Information (NCBI). We used the December 2008 version of the Structure Data Format (SDF) database.[2]
4. **SC-11.5M** contains compounds from the SureChEMBL [40] database, which includes a large set of chemical compounds automatically extracted from text, images, and attachments of patent documents. **SC-5M**, **SC-1M**, **SC-500K**, and **SC-100K** are random subsets of 5E+6, 1E+6, 5E+5, and 1E+5 compounds, respectively, from the **SC-11.5M** dataset.

Table 2 Dataset statistics

Dataset	n	m	nnz	μ_r	μ_c
RCV1	804K	46K	61.5M	77	1348
Patents-8.8M	8820K	16,591K	4277.1M	485	258
Patents-4M	4000K	8187K	1791.0M	448	219
Patents-2M	2000K	4146K	837.4M	419	202
Patents-1M	1000K	3215K	464.5M	465	145
Patents-500K	500K	2156K	233.0M	466	108
Patents-250K	250K	1403K	116.1M	464	83
Patents-100K	100K	759K	46.3M	464	61
MLSMR	325K	20K	56.1M	173	2803
SC-11.5M	11,519K	7,415	1,784.5M	155	262,669
SC-5M	5000K	7415	699.9M	155	103,063
SC-1M	1000K	6752	154.9M	155	22,949
SC-500K	500K	6717	77.5M	155	11,533
SC-100K	100K	6623	15.5M	155	2336

In the table, n represents the number of objects (*rows*), m is the number of features in the vector representation of the objects (*columns*), nnz is the number of nonzero values, and μ_r and μ_c are the mean number of nonzeros in each row and column, respectively. The first eight lines in the table represent text datasets, while the rest of the lines are chemical compound datasets

4.1.1 Text data processing

We used standard text processing methods to encode documents as sparse vectors. Each document was first tokenized,

[1] http://www.uspto.gov/.

[2] https://mlsmr.evotec.com/MLSMR_HomePage/pdf/MLSMR_Collection_20081201.zip.

removing punctuation, making text lowercased, and splitting the document into a set of words. Each word was then stemmed using the Porter stemmer [41], reducing different versions of the same word to a common token. Within the space of all tokens, a document is then represented by the sparse vector containing the frequency of each token present in the document.

4.1.2 Chemical compound processing

We encode each chemical compound as a sparse frequency vector of the molecular fragments it contains, represented by GF [42] descriptors extracted using the AFGen v. 2.0 [43] program.[3] AFGen represents molecules as graphs, with vertices corresponding to atoms and edges to bonds in the molecule. GF descriptors are the complete set of unique size-bounded subgraphs present in each compound. Within the space of all GF descriptors for a compound dataset, a compound is then represented by the sparse vector containing the frequency of each GF descriptor present in the compound. We used a minimum length of 3 and a maximum length of 5 and ignored hydrogen atoms when generating GF descriptors (AFGen settings *fragtype=GF, lmin=3, lmax=5, fmin=1, noh: yes*). Before running AFGen on each chemical dataset, we used the Open Babel toolbox [44] to remove compounds with incomplete descriptions.

4.2 Baseline approaches

We compare our methods against the following baselines.

- `IdxJoin` [33] is a straight-forward baseline that does not use any pruning when computing similarities. `IdxJoin` uses an accumulator data structure to simultaneously compute the dot-products of a query object with all prior processed objects, iterating through the inverted lists corresponding to features in the query. While in [33] the method was used to compute dot-products of normalized vectors, we apply the method on the un-normalized vectors. Resulting Tanimoto similarities are computed according to Eq. 1, using previously stored vector norms. Then, those similarities below ϵ are removed.
- `L2AP` [33] solves the all-pairs problem for the cosine similarity, rather than the Tanimoto coefficient. As shown in Sect. 3.3, the Tanimoto all-pairs result is a subset of the cosine all-pairs result. After executing the `L2AP` algorithm, we use Eq. 3 and previously stored vector norms to compute the Tanimoto coefficient of all resulting object pairs and filter out those below ϵ.
- `MMJoin` [34] is a filtering-based approach to solving the all-pairs problem for the Tanimoto coefficient. It relies on efficiently solving the cosine similarity all-pairs problem using pruning bounds based on vector lengths and the number of nonzero features in each vector.
- `MK-Join` is a method we designed using the Tanimoto similarity pruning bounds described by Kryszkiewicz in [35] and [36]. `MK-Join` uses an accumulator to compute similarities of each query against all candidates found in the inverted lists associated with features present in the query. However, `MK-Join` processes inverted lists in a different order, in non-increasing order of the query feature values. By following this order, Kryszkiewicz has shown that the method can *safely* stop accepting new candidates once the squared norm of the partially processed query vector (i.e., setting values of unprocessed features to 0) falls below $t = 1 - (\frac{2\epsilon}{1+\epsilon})^2$. A candidate is also ignored if its length $\|\mathbf{d}_c\|$ falls outside the range $[(1/\alpha)\|\mathbf{d}_q\|,\ \alpha\|\mathbf{d}_q\|]$, where α is defined as in Eq. 2.
- We also implemented `MK-Join2`, a version of `MK-Join` that further incorporates a tighter bound on candidate lengths described by Kryszkiewicz in Theorem 5 of [37]. The bound is equivalent to our Eq. 8 with $s = \sqrt{1 - \sum_{i \in L} \hat{d}_{q,i}}$, given the set L of query features that are not also candidate features. However, finding this set requires traversing both the query and candidate sparse vectors, which reduces the benefit obtained by pruning candidates. As an example, Table 3 shows the results of executing `MK-Join` and `MK-Join2` for $\epsilon \in \{0.6, 0.7, 0.8, 0.9, 0.99\}$ on the MLSMR and RCV1 datasets. The execution environment details for this experiment are provided in Sect. 4.4. While executing as little as 1/5 of the dot-product computations that `MK-Join` executes, `MK-Join2` was slower than `MK-Join` in our experiments. As a result, in order to reduce clutter in our figures, we only include the results for `MK-Join` in Sect. 5.

Table 3 Comparison of `MK-Join` and `MK-Join2`

MLSMR			RCV1		
ϵ	dps	time	ϵ	dps	time
0.6	0.9019	1.0169	0.6	0.6088	1.0044
0.7	0.8348	1.0350	0.7	0.5890	1.0161
0.8	0.7303	1.0339	0.8	0.5942	1.0141
0.9	0.5681	1.0329	0.9	0.6202	1.0054
0.99	0.2275	1.0027	0.99	0.5115	1.0067

The table shows the ratio of the number of dot-products computed by `MK-Join2` and `MK-Join` (*dps* column), and the ratio of the time taken by `MK-Join2` and that of `MK-Join` (*time* column) for two datasets and five different ϵ values

[3] http://glaros.dtc.umn.edu/gkhome/afgen/download.

4.3 Performance measures

We compare the search performance of different methods in terms of CPU runtime, which is measured in seconds. I/O time needed to load the dataset into memory or write output to the file system should be the same for all methods and is ignored. Between a method A and a baseline B, we report speedup as the ratio of B's execution time and that of A's.

We use the *number of candidates* and the *number of full similarity computations* as an architecture- and programming language-independent way to measure similarity search cost [33,45,46]. A naïve method may compute up to $n(n-1) = O(n^2)$ similarities to solve the APSS problem. However, all of our comparison methods take advantage of the commutative property of the Tanimoto similarity and at most compare $\frac{n(n-1)}{2}$ candidate object pairs and compute as many similarities. We thus report the percent of compared candidates (*candidate rate*) and computed full dot-products (*scan rate*) as opposed to this upper limit.

4.4 Execution environment

Our method[4] and all baselines are single-threaded, serial programs, implemented in C, and compiled using gcc 5.1.0 with the -O3 optimization setting enabled. Each method was executed on its own node in a cluster of HP Linux servers. Each server is a dual-socket machine, equipped with 24 GB RAM and two four-core 2.6 GHz Intel Xeon 5560 (Nehalem EP) processors with 8 MB Cache. We executed each method a minimum of three times for $\epsilon \in \{0.6, 0.7, 0.8, 0.9, 0.99\}$ and report the best execution time in each case. Processing the full SC-11.5M (1.78B nonzeros, 14 GB on disk) and Patents-8.8M (4.28B nonzeros, 28 GB on disk) datasets requires more than the available RAM on the Nehalem machines; thus, we executed data scaling experiments on a different server, equipped with 64 GB RAM and two 12-core 2.5 Ghz Intel Xeon (Haswell E5-2680v3) processors with 30 MB Cache. All datasets except the full Patents-8.8M dataset could be processed using 64 GB RAM. The Patents dataset experiments were executed on a high-memory machine with the same Haswell processors and 256 GB RAM. As all tested methods are serial, only one core was used on each server during the execution.

5 Results and discussion

Our experiment results are organized along several directions. First, we analyze statistics of the input data and output neighborhood graphs for some of the datasets we use in our experiments, and the effectiveness of our new Tanimoto bounds at pruning the similarity search space. Then, we compare the efficiency of our method against existing state-of-the-art baselines, demonstrating up to an order of magnitude improvement. Finally, we analyze the scaling characteristics of our method when dealing with increasing amounts of data.

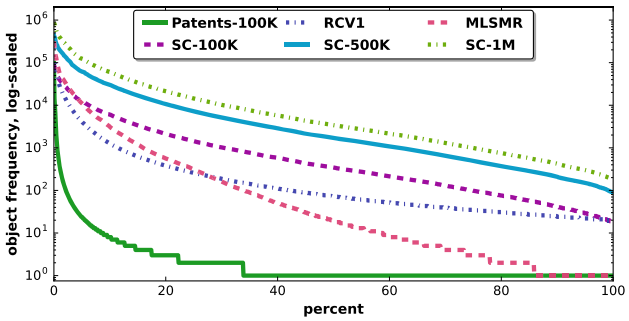

Fig. 5 Object frequency distributions for dataset features

5.1 Neighborhood graph statistics

The efficiency of similarity search methods for input objects represented as a sparse matrix is highly dependent on the characteristics of those data. Consider, for example, a banded sparse matrix of width k. Each object would have to be compared against at most $2k$ other objects. On the other hand, almost all pairwise similarities must be computed if the nonzeros are randomly distributed in the matrix. In many real-world datasets, the object frequency of features (the number of objects that have a nonzero value for a feature) displays a power-law distribution, with a small number of features present in many objects and the majority of features present in few objects. Thus, even though these datasets are sparse, the features at the head of the distribution will cause most objects to be compared against most other objects when computing pairwise similarities.

Our chosen datasets have diverse object frequency distributions. Figure 5 shows these distributions for six of the datasets in Table 2. Note that the frequency counts are log-scaled to better distinguish differences between the distributions. The graph shows that more than 60% of the 759,044 features in the Patents-100K dataset can only be found in one object, yet the top 1% of features can each be found in at least 490 objects. Similarly, almost 15% of the 20,021 features in the MLSMR dataset are only present in one object, but 200 features are present in at least 63,646 of the 325,164 objects in the dataset. In the RCV1 and SC datasets, all features are present in at least ten objects.

While sparsity and feature distributions play a big role in the number of objects that must be compared to solve the APSS problem exactly, the number of computed similarities is also highly dependent on the threshold ϵ. We studied

[4] Source code available at http://davidanastasiu.net/software/tapnn/.

Table 4 Neighborhood graph statistics

ϵ	μ	ρ	μ	ρ
	Patents-100K		RCV1	
0.1	3412	3.412e−02	21,655	2.692e−02
0.2	445	4.452e−03	2707	3.366e−03
0.3	82	8.208e−04	881	1.095e−03
0.4	15	1.535e−04	417	5.196e−04
0.5	2.6	2.615e−05	199	2.484e−04
0.6	0.47	4.716e−06	85	1.062e−04
0.7	0.15	1.513e−06	34	4.300e−05
0.8	0.09	8.660e−07	12	1.616e−05
0.9	0.06	6.006e−07	5.2	6.428e−06
0.99	0.04	3.818e−07	1.2	1.433e−06
	MLSMR		SC-100K	
0.1	281,509	8.657e−01	67,121	6.712e−01
0.2	212,894	6.547e−01	45,229	4.523e−01
0.3	126,620	3.894e−01	26,198	2.620e−01
0.4	61,067	1.878e−01	12,950	1.295e−01
0.5	23,482	7.222e−02	5300	5.300e−02
0.6	6569	2.020e−02	1688	1.688e−02
0.7	1184	3.644e−03	397	3.976e−03
0.8	127	3.924e−04	72	7.270e−04
0.9	10	3.358e−05	11	1.128e−04
0.99	0.28	8.495e−07	0.09	8.900e−07
	SC-500K		SC-1M	
0.1	336,815	6.736e−01	673,156	6.732e−01
0.2	226,917	4.538e−01	453,149	4.531e−01
0.3	131,385	2.628e−01	262,292	2.623e−01
0.4	64,982	1.300e−01	129,752	1.298e−01
0.5	26,590	5.318e−02	53,152	5.315e−02
0.6	8442	1.688e−02	16,914	1.691e−02
0.7	1963	3.927e−03	3953	3.953e−03
0.8	349	6.996e−04	710	7.104e−04
0.9	54	1.085e−04	110	1.109e−04
0.99	0.47	9.351e−07	0.95	9.453e−07

The table shows the average neighborhood size (μ) and neighborhood graph density (ρ) for six of the test datasets and ϵ ranging from 0.1 to 0.99

properties of the output graph to understand how the input threshold can affect the efficiency of search algorithms. Each nonzero 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 exact APSS algorithm will take a long time to solve the problem, no matter how effectively it can prune the search space. Table 4 shows the average neighborhood size (μ) and neighborhood graph density (ρ) for six of the test datasets and ϵ ranging from 0.1 to 0.99. 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. 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 $\epsilon = 0.99$. However, the average number of neighbors and graph density increase disproportionally for the different datasets as ϵ increases. The Patents-100K dataset has less than 100 neighbors on average for each of the objects even at $\epsilon = 0.3$, while the chemical datasets have hundreds of neighbors on average even at $\epsilon = 0.8$. The density of the similarity graphs for the chemical datasets increases rapidly as ϵ decreases. For $\epsilon = 0.1$, these graphs contain more than 67% of the edges in the complete graph.

To put things in perspective, the **673.16 billion** edges of the SC-1M neighborhood graph for $\epsilon = 0.1$ take up **16.7 Tb** of hard drive space, and more than half of those represent similarities below 0.5, which are somewhat distant neighbors. Nearest neighbor-based classification or recommender systems methods often rely on a small number (generally less than 100) of each object's nearest neighbor to complete their task. This analysis suggests that different ϵ thresholds may be appropriate for the analysis of different datasets. Searching the Patents-100K dataset using $\epsilon = 0.3$, the RCV1 dataset using $\epsilon = 0.6$, the MLSMR dataset using $\epsilon = 0.8$, and the SC-1M dataset using $\epsilon = 0.9$ would provide enough nearest neighbors on average to complete the required tasks.

Figure 6 gives a more detailed picture of the distribution of neighborhood sizes for the similarity graphs in Table 4 and a subset of the ϵ thresholds. 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 the Patents-100K dataset, 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. As the figure shows, the potential for savings is less than half on chemical datasets for $\epsilon = 0.5$ and shrinks to almost nothing at $\epsilon = 0.1$. On the other hand, text datasets show a much higher potential for savings, even at low ϵ thresholds.

5.2 Pruning effectiveness

We now study the effectiveness of our method, along several directions. First, we analyze the performance of our method with regard to the number of ignored or pruned object pairs in different stages of the similarity search and the effectiveness of the partial indexing strategy described in Sect. 3.3. Then, we compare the amount of pruning in our method to that in other state-of-the-art filtering methods. Finally, we consider the effect of our Tanimoto-specific pruning on the efficiency on our method.

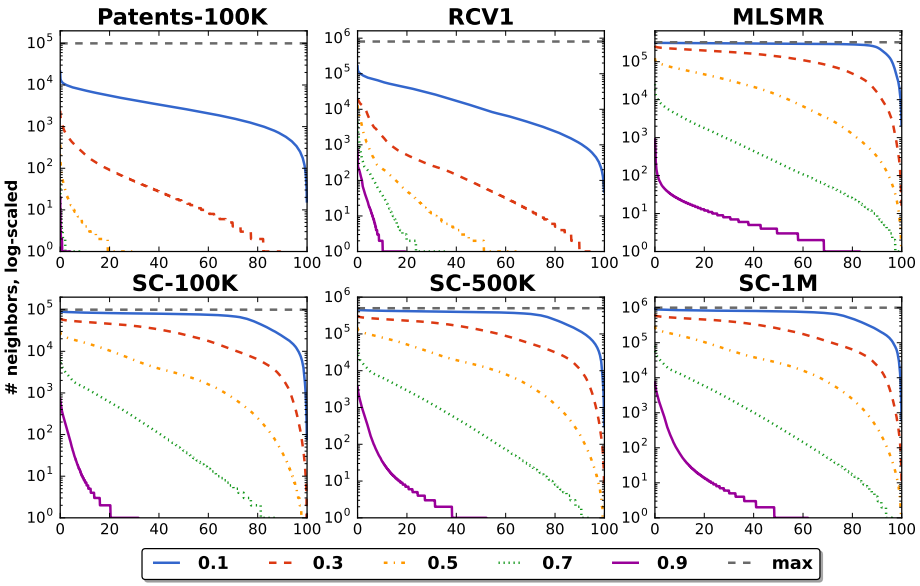

Fig. 6 Neighbor count distributions for several values of ϵ

5.2.1 Effectiveness of pruning the search space

As described in Sect. 3 and shown in Fig. 2, our method works by taking advantage of sparsity in the input data, the length of the input vectors, and even the angle between vectors to prune the search space. In order to measure the effectiveness of our method, we instrumented our code to count the number of object pairs that were pruned as a result of each of these strategies. We first show the pruning effected by TAPNN prior to generating candidates, by taking advantage of sparsity, vector lengths, and partial indexing based on vector angles. Note that TAPNN does not compute any part of the similarity for these pruned object pairs.

Table 5 shows the cumulative percent of the pairwise similarity search space pruned by these strategies for six of the test datasets and ϵ ranging from 0.1 to 0.99. Percent values are computed with respect to the number of object similarities considered by a naïve algorithm while taking advantage of the commutative property of Tanimoto similarity, i.e., $\frac{n(n-1)}{2}$. As the results show, TAPNN is very effective at high similarity thresholds, pruning up to 99.995% of the search space in the case of the RCV1 dataset and $\epsilon = 0.99$. However, for small ϵ values, when the output graph is no longer sparse (see Table 4), the amount of pruning effected by TAPNN prior to generating candidates dwindles. Angle- and length-based pruning are most effective in our method, accounting for 90–100% of the pruning effectiveness across datasets and thresholds. While our datasets are very sparse (their nonzero densities range between 6.10E−4 and 2.34E−2), the distributions of the features in the data cause the majority of objects to be potential neighbors. However, the length- and angle-based pruning in TAPNN effectively reduces the number of object pairs that must be compared to solve the problem.

The *idx* column in Table 5 shows the percent of the input dataset nonzeros that are indexed by our method. Indexing fewer nonzeros increases the efficiency in our method by allowing it to traverse shorter inverted index lists during the candidate generation stage, and it leads to more pruning. We see this correlation by comparing the *idx* column with the percent of the pruning effected by the partial indexing (the *angle* column minus the *sparsity* and *length* columns) in the table. The comparison reveals a Pearson correlation ranging from 0.9314 for the Patents-100K dataset and 0.9993 for the SC datasets. At high values of ϵ, our method indexes few features, which in turn leads to many potential candidates being implicitly ignored because they have no features in common with the indexed part of the query vector. On the other hand, at low similarity thresholds, the majority of the input nonzeros are indexed, leading to fewer objects being pruned.

While TAPNN prunes some of the search space before candidate generation, it also continues the pruning process once an object becomes a candidate. Table 6 compares the percent of pairwise object pairs that become candidates in our method (candidate rate—*cand* column) versus those whose similarity is fully computed by our method (scan rate—*dps* column) and those who are actually neighbors (*nbr* column), given ϵ ranging from 0.1 to 0.99 and six different datasets. The *cand* column represents the un-pruned object pairs whose similarities we actually start computing, and is equivalent to 100% minus the *angle* column in Table 5. Our method actually computes the similarity in full for a much smaller number of object pairs, shown in the *dps* column. It is also interesting to note that the percent of object pairs whose similarity we compute in full is actually very close to the number of true neighbors, irrespective of similarity threshold, highlighting the effectiveness of our filtering framework.

Table 5 Search space pruning in TAPNN prior to candidate generation

ϵ	sparsity	length	angle	idx	sparsity	length	angle	idx	sparsity	length	angle	idx
	Patents-100K				RCV1				MLSMR			
0.1	0.00	1.69	1.792	93.47	10.49	10.65	22.808	97.00	0.16	0.18	0.707	99.59
0.2	0.00	7.28	7.763	82.45	10.49	11.39	34.061	90.61	0.16	0.39	1.885	98.26
0.3	0.00	14.67	16.353	71.42	10.49	13.10	46.510	81.69	0.16	0.92	3.931	96.26
0.4	0.00	22.08	27.328	62.11	10.49	14.82	61.385	71.11	0.16	1.88	8.421	92.10
0.5	0.00	28.09	40.411	53.59	10.49	15.50	75.678	59.79	0.16	3.55	16.504	85.24
0.6	0.00	31.52	54.684	45.47	10.49	14.99	86.699	48.17	0.16	6.12	28.782	75.50
0.7	0.00	31.39	69.110	37.50	10.49	13.71	93.974	36.73	0.16	8.86	45.490	62.92
0.8	0.00	27.16	82.428	29.54	10.49	12.25	97.922	25.84	0.16	10.44	65.557	47.41
0.9	0.00	17.22	93.655	20.48	10.49	11.07	99.608	15.12	0.16	8.53	87.216	28.04
0.99	0.00	2.05	99.815	7.51	10.49	10.51	99.995	3.49	0.16	1.10	99.749	4.27
	SC-100K				SC-500K				SC-1M			
0.1	1.53	2.44	3.996	99.32	1.44	2.32	3.860	99.32	1.45	2.33	3.872	99.32
0.2	1.53	5.43	8.208	97.74	1.44	5.23	8.025	97.74	1.45	5.26	8.056	97.73
0.3	1.53	9.47	13.935	95.69	1.44	9.22	13.701	95.70	1.45	9.26	13.742	95.69
0.4	1.53	13.84	21.251	92.43	1.44	13.61	21.023	92.45	1.45	13.65	21.062	92.44
0.5	1.53	18.38	30.443	87.52	1.44	18.18	30.200	87.56	1.45	18.23	30.237	87.55
0.6	1.53	22.51	41.667	80.48	1.44	22.35	41.437	80.52	1.45	22.40	41.478	80.51
0.7	1.53	25.11	55.183	70.36	1.44	25.00	55.006	70.42	1.45	25.03	55.050	70.40
0.8	1.53	24.75	71.018	56.00	1.44	24.73	70.878	56.08	1.45	24.74	70.905	56.08
0.9	1.53	18.52	88.246	35.75	1.44	18.46	88.190	35.78	1.45	18.47	88.193	35.78
0.99	1.53	3.47	99.734	6.60	1.44	3.38	99.734	6.59	1.45	3.39	99.733	6.60

The table shows, in the *sparsity, length*, and *angle* columns, respectively, the cumulative percent of the pairwise similarity search space pruned by taking advantage of sparsity, vector lengths, and partial indexing based on vector angles for six of the test datasets and ϵ ranging from 0.1 to 0.99. The *idx* column shows the percent of the input dataset nonzeros that are indexed by our method

During the similarity search, after an object becomes a candidate for some query object, it can be pruned if its similarity estimate with the query falls below the threshold ϵ based on several theoretic upper bounds described in Sect. 3. Figure 7 shows the percent of candidates pruned by the different bounds, in addition to those candidates whose similarity is computed in full (`dpscore`). Objects can be pruned as soon as they become candidates, in the candidate generation stage, by our ℓ^2-norm-based pruning bound (*l2cg*), or as soon as candidate verification starts, through our *ps* bound. Additional pruning is effected through our tighter bound for the un-normalized candidate vector length β, which here we call the vector length angle bound (*vla*), and our ℓ^2-norm-based pruning during candidate verification (*l2cv*). The results show that the majority of the pruning is done early on, during the candidate generation stage. For text datasets, pruning overshadows the percent of objects whose similarity is computed, and those portions of the bars are not even visible for most ϵ values. Moreover, the Tanimoto-specific candidate pruning (*vla*) makes up a significant portion of the overall pruning, especially for chemical datasets.

5.2.2 *Effectiveness comparison with filtering baselines*

Many of the baseline methods we are comparing against in this paper are also filtering methods. As an architecture- and programming language-independent way to compare the effectiveness of our method against the baselines, we show the candidate rate (*cand* column) and scan rate (*dps* column) for all filtering methods under comparison in Table 7, for four of the datasets and ϵ ranging from 0.3 to 0.9. Bold values represent the smallest candidate and scan rates across methods for each similarity threshold.

The results show that TAPNN is most effective among the compared methods at pruning the search space, which results in the fewest similarity values computed in full. L2AP has the closest scan rates to our method for text-based datasets, but, without Tanimoto-specific pruning, considers many more candidates in general, especially for chemical datasets. While MMJoin prunes much of the search space, it lags behind both TAPNN and L2AP. With its vector length-based pruning, MK-Join is able to ignore many objects without starting to compute their similarity. At high thresholds, its candidate rate is often lower than both that of MMJoin and L2AP. However,

Table 6 Pruning performance during filtering in TAPNN

ϵ	*cand*	*dps*	*nbr*	*cand*	*dps*	*nbr*	*cand*	*dps*	*nbr*
	Patents-100K			RCV1			MLSMR		
0.1	98.208	3.75195	3.41222	77.192	5.12196	2.69207	99.293	86.98722	86.57500
0.2	92.237	0.57141	0.44519	65.939	0.89402	0.33658	98.115	65.97607	65.47327
0.3	83.647	0.12117	0.08208	53.490	0.32050	0.10953	96.069	39.52094	38.94078
0.4	72.672	0.02500	0.01535	38.615	0.14907	0.05196	91.579	19.54342	18.78069
0.5	59.589	0.00449	0.00261	24.322	0.07025	0.02484	83.496	7.80633	7.22169
0.6	45.316	0.00076	0.00047	13.301	0.03078	0.01062	71.218	2.34046	2.02025
0.7	30.890	0.00020	0.00015	6.026	0.01216	0.00430	54.510	0.46706	0.36440
0.8	17.572	0.00010	0.00009	2.078	0.00416	0.00162	34.443	0.05425	0.03924
0.9	6.345	0.00006	0.00006	0.392	0.00117	0.00064	12.784	0.00416	0.00336
0.99	0.185	0.00004	0.00004	0.005	0.00018	0.00014	0.251	0.00009	0.00008
	SC-100K			SC-500K			SC-1M		
0.1	96.004	68.00492	67.12217	96.140	68.24212	67.36314	96.128	68.19030	67.30840
0.2	91.792	46.28032	45.23038	91.975	46.44308	45.38356	91.944	46.37475	45.31497
0.3	86.065	27.20557	26.19898	86.299	27.29515	26.27724	86.258	27.24872	26.22932
0.4	78.749	13.88424	12.95055	78.977	13.93386	12.99662	78.938	13.91150	12.97526
0.5	69.557	5.94892	5.30025	69.800	5.96769	5.31808	69.763	5.96442	5.31528
0.6	58.333	2.01654	1.68805	58.563	2.01728	1.68846	58.522	2.01987	1.69147
0.7	44.817	0.50744	0.39764	44.994	0.50206	0.39275	44.950	0.50491	0.39532
0.8	28.982	0.09779	0.07270	29.122	0.09415	0.06996	29.095	0.09545	0.07104
0.9	11.754	0.01589	0.01128	11.810	0.01515	0.01085	11.807	0.01547	0.01109
0.99	0.266	0.00014	0.00009	0.266	0.00015	0.00009	0.267	0.00015	0.00009

The table shows the percent of pairwise object comparisons considered by our algorithm (*cand* column), the percent of pairwise object pairs whose similarity is fully computed by our method (*dps* column), and the percent of pairwise object pairs that are actually neighbors (*nbr* column), given ϵ ranging from 0.1 to 0.99 and six different datasets

the method seems ineffective at pruning candidates, resulting in very high scan rates.

5.2.3 Effectiveness of Tanimoto bounds

As another way to test the pruning effectiveness of the new Tanimoto length bounds introduced in Sect. 3.4, we compared execution times of TAPNN with two versions of the program which did not take advantage of these bounds. While both programs implement the length-based pruning described in Sect. 3.1, TAPNN-c filters cosine neighbors using the threshold ϵ, while TAPNN-t employs the tighter cosine filtering bound from Eq. 4. Figure 8 shows the log-scaled execution times for the three methods, given ϵ ranging from 0.3 to 0.99.

The results of our experiments indicate that the newly introduced bounds are effective at improving search performance, achieving up to 5.8x speedup against TAPNN-t and 13.3x speedup against TAPNN-c. Chemical datasets exhibit higher performance improvement at high thresholds, but much lower as $\epsilon \rightarrow 0.6$.

5.3 Execution efficiency

The main goal of our method is to efficiently solve the Tanimoto APSS problem. We compared TAPNN against the baselines described in Sect. 4.2, for a wide range of ϵ values. Figure 9 displays our timing results for each method on four datasets. In each quadrant, smaller times indicate better performance. Note that the y-axis has been log-scaled.

The results show that TAPNN significantly outperformed all baselines, by up to an order of magnitude, for all thresholds $\epsilon \geq 0.6$. As discussed in Sect. 5.1, neighborhood graphs for lower similarities are likely too dense and provide less benefit for neighborhood-based analysis. In the range $\epsilon \in [0.6, 0.99]$, speedup of TAPNN versus the next best method was between 3.0x–8.0x for text datasets and 1.2x–12.5x for chemical datasets. Speedup against IdxJoin which is similar to a linear search and does not employ any pruning ranged between 8.3x–3981.4x for text data and 1.5x–519x for chemical data, highlighting the pruning performance of our method, especially for high values of ϵ.

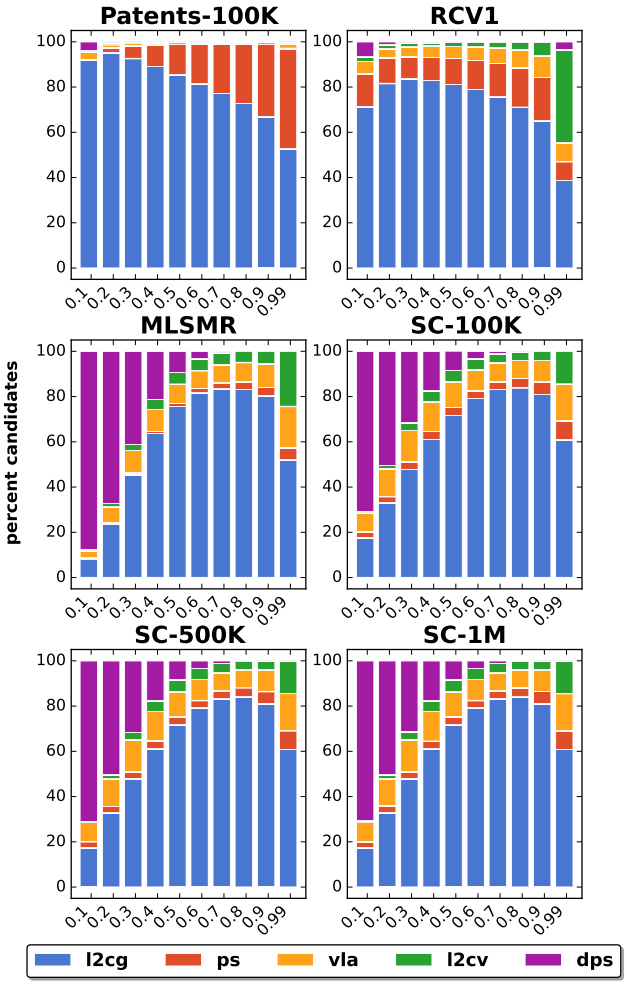

Fig. 7 Percent of candidates pruned by different bounds in TAPNN (Best viewed in color)

TAPNN performed on par with the IdxJoin baseline on the two chemical datasets for $\epsilon = 0.5$, and slightly worse than the IdxJoin and MK-Join baselines for lower thresholds. While our method pruned most of the object pairs in the search space that were not neighbors (see Sect. 5.2), the benefit gained by the pruning did not outweigh the cost of checking filtering bounds at small similarity thresholds. The IdxJoin and MK-Join baselines spend no or little time checking filtering bounds, which is an advantage when the neighborhood graph is fairly dense.

The best performing baseline in general was L2AP, our previous cosine APSS method, which employs similar cosine-based pruning, but does not take advantage of unnormalized vector lengths in its filtering. L2AP was shown in [33] to outperform MMJoin for the cosine APSS task. Our results show that it also outperformed MMJoin for Tanimoto APSS, in all experiments. MK-Join was not competitive against L2AP and MMJoin for $\epsilon \geq 0.8$ for chemical datasets and in general for text datasets. In fact, it performed worse than IdxJoin for the Patents-100K dataset and only slightly better in general. The Patents-100K dataset has a high average vector size (number of nonzeros) and low average index list size, which may have contributed to the poor performance of MK-Join. The results show that the strategy of cosine filtering applied to the Tanimoto APSS problem, which is employed in different ways by TAPNN, L2AP, and MMJoin, works quite well for both text and chemical datasets.

5.4 Scaling

As a way for us to understand the scalability of our method, we measured the execution time when searching for neighbors, given ϵ between 0.5 and 0.99, on three random subsets from the SC-11.5M dataset (100K, 500K, and 1M compounds) and four random subsets of the Patents-8.8M dataset (100K, 250K, 500K, and 1M patents), for TAPNN and the IdxJoin, MK-Join, and MMJoin baselines. Figure 10 shows the results of these experiments for the Patents (left) and SC (right) datasets. In each quadrant of each subfigure, we plot the number of nonzeros in the dataset ($\times 10^8$, x-axis) against the execution time (log-scaled, y-axis).

Overall, the results show that algorithms display similar scaling trends as dataset sizes are increased. However, as ϵ is increased, TAPNN is able to distance itself from baselines, increasing the efficiency gap to outperform them by over an order of magnitude. The SC and Patents datasets are quite different. By construction, the SC dataset has few features (less than 7.5K), which means its inverted lists become quite long (up to 262.7K compounds on average for the full SC-11.5M dataset), and many object pairs are likely to have at least one feature in common. On the other hand, patents use quite diverse terminology, which is evident form the drastic increase in the number of features in the Patents datasets, from 759.0K to 16.6M between the Patents-100K and Patents-8.8M datasets. TAPNN is able to get excellent performance for both types of data by employing effective pruning strategies. Our analysis in Sect. 5.1 also showed that the neighborhood graph for the SC datasets is close to complete at $\epsilon = 0.5$ and below, which means there is little to be gained by filtering in these scenarios. However, filtering is very effective at high similarity thresholds for these datasets, producing dramatic speedups over state-of-the-art baselines.

We also tested TAPNN in a near-duplicate detection scenario on SC subsets ranging from 500K to 11.5M compounds and Patents datasets ranging from 500K to 8.8M compounds, for $\epsilon \in \{0.95, 0.975, 0.99, 0.999\}$. Baseline methods were not able to complete execution for the very large datasets in a reasonable amount of time (96 h), and we do not include them in this result. Figure 11 shows execution times in our experiments for the Patents (left) and SC (right) datasets.

Table 7 Comparison of candidate and scan rates for filtering-based methods

ϵ	TAPNN		L2AP		MMJoin		MK-Join	
	cand	*dps*	*cand*	*dps*	*cand*	*dps*	*cand*	*dps*
	Patents-100K							
0.30	**83.65**	**0.1212**	99.87	1.3527	100.00	8.5759	90.20	76.8965
0.40	**72.67**	**0.0250**	99.38	0.4173	99.67	2.0669	79.26	60.9828
0.50	**59.59**	**0.0045**	97.59	0.1187	95.98	0.4888	67.90	46.2979
0.60	**45.32**	**0.0010**	93.10	0.0267	88.48	0.1460	57.65	34.0969
0.70	**30.89**	**0.0002**	83.93	0.0040	77.31	0.0306	48.86	24.2447
0.80	**17.57**	**0.0001**	67.56	0.0004	60.48	0.0049	41.42	16.2758
0.90	**6.34**	**0.0001**	41.18	0.0001	33.78	0.0005	37.17	10.0336
	RCV1							
0.30	**53.49**	**0.3205**	73.62	0.8375	73.09	15.1487	66.45	63.3365
0.40	**38.61**	**0.1491**	64.23	0.2835	67.11	10.5094	54.47	48.8616
0.50	**24.32**	**0.0703**	52.46	0.1244	52.37	6.0532	43.81	35.9840
0.60	**13.30**	**0.0314**	38.80	0.0611	34.93	2.6000	34.52	25.1845
0.70	**6.03**	**0.0124**	24.57	0.0265	18.99	0.7648	26.92	16.7887
0.80	**2.08**	**0.0042**	11.74	0.0086	8.07	0.1929	20.86	10.4767
0.90	**0.39**	**0.0012**	3.15	0.0017	1.95	0.0307	16.43	5.7592
	MLSMR							
0.30	**96.07**	**39.5209**	98.87	77.0628	98.74	82.1759	98.89	98.0920
0.40	**91.58**	**19.5434**	97.89	59.7118	98.11	74.5478	98.36	96.4064
0.50	**83.50**	**7.8063**	95.97	39.0700	97.11	65.0857	97.59	93.4267
0.60	**71.22**	**3.1128**	91.67	19.9784	93.43	47.2142	96.74	88.5399
0.70	**54.51**	**0.6213**	83.62	6.9207	82.13	22.6891	95.65	80.8921
0.80	**34.44**	**0.0671**	68.34	1.1792	62.17	6.1240	94.01	69.2132
0.90	**12.78**	**0.0044**	40.82	0.0462	32.23	0.5651	92.06	51.0706
	SC-1M							
0.30	**86.26**	**27.2487**	96.11	62.3976	95.73	73.8524	94.27	86.5448
0.40	**78.94**	**13.9115**	94.83	47.6004	94.72	68.0235	92.43	80.0253
0.50	**69.76**	**5.9644**	92.86	31.8095	93.51	60.1836	90.39	72.6842
0.60	**58.52**	**2.0199**	89.69	17.7314	90.29	45.9351	88.33	64.5104
0.70	**44.95**	**0.7568**	84.00	7.2723	82.17	27.4836	86.13	55.3111
0.80	**29.10**	**0.0954**	72.62	1.7795	67.04	11.2794	83.38	44.4693
0.90	**11.81**	**0.0193**	48.71	0.2018	40.29	2.3703	80.71	30.9351

The table shows the candidate and scan rates for the filtering-based methods under comparison, as the result of experiments over four datasets and ϵ ranging from 0.3 to 0.9. Bold values represent the smallest candidate and scan rates across methods for each similarity threshold

In each quadrant of each subfigure, we plot the number of nonzeros in the dataset ($\times 10^9$, x-axis) against the execution time (log-scaled, y-axis). Each line shows an execution of our TAPNN algorithm with the labeled ϵ value. The name (and size) of the dataset the experiment is executed on is also written below the markers of the $\epsilon = 0.999$ line.

The results of this experiment confirm that our method continues its nice scaling characteristics even for very large datasets, and the trend is similar as the ϵ threshold is decreased. As the dataset increases in size, there is more opportunity for pruning, which allows TAPNN to maintain and improve its overall performance.

Given increasing dataset sizes, it would be beneficial to investigate shared memory and distributed extensions of TAPNN. Existing strategies for parallelizing cosine APSS filtering strategies [47–49] are likely to provide similar benefits in the Tanimoto APSS context. While the serial version can find all nearest neighbors for 1M SC compounds with $\epsilon \geq 0.95$ in minutes, a parallel version of the algorithm is needed to achieve similar performance for lower ϵ thresholds.

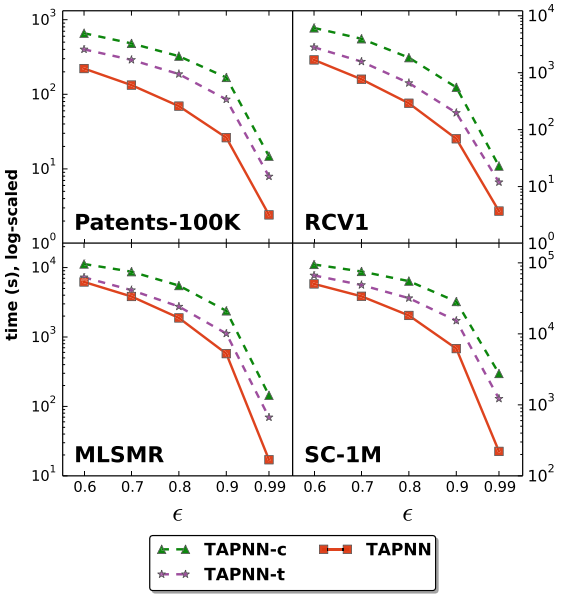

Fig. 8 Effect of Tanimoto bounds on search efficiency

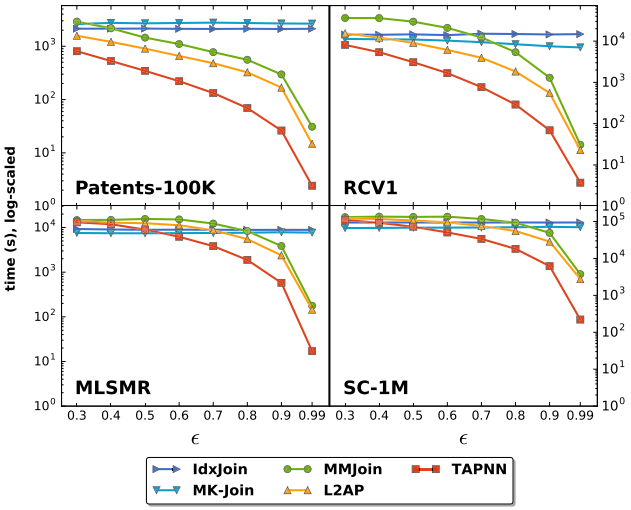

Fig. 9 Efficiency comparison of TAPNN versus baselines

6 Conclusion

We presented TAPNN, a new serial algorithm for solving the Tanimoto all-pairs similarity search problem for objects represented as nonnegative real-valued vectors. Unlike many alternatives, our method solves the problem *exactly*, finding *all* pairs of objects with a Tanimoto similarity of at least some input threshold ϵ. Our method incorporates several filtering strategies based on object vector lengths and the dot-product of their normalized vectors. We have shown how these strategies can be effectively used to reduce the number of object pairs that have to be fully compared, and have introduced additional filtering techniques that combine normalized dot-product estimates with un-normalized vector lengths. We experimentally evaluated our method against several baselines on both chemical and text datasets, and found TAPNN significantly outperformed them, especially for high thresholds. In particular, TAPNN was able to find all near-duplicate pairs among 5M SureChemBL chemical compounds in minutes, using a single CPU core, was up to 12.5x more efficient than the most efficient baseline, and outperformed a linear search baseline by two orders of magnitude in general at $\epsilon = 0.99$.

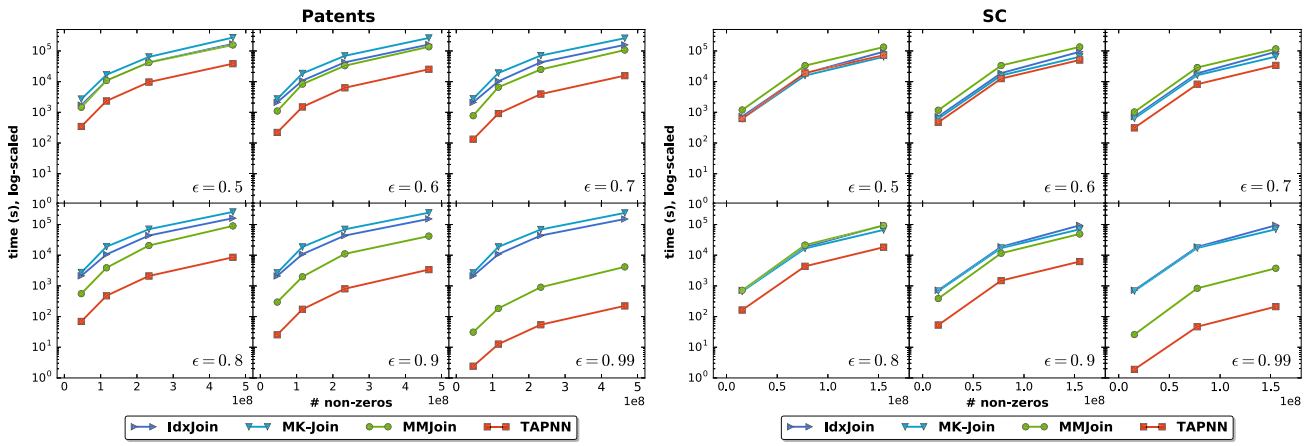

Fig. 10 Scaling characteristics of TAPNN in comparison with baselines at ϵ thresholds ranging from 0.5 to 0.99 over subsets of the Patents-8.8M (left) and SC-11.5M (right) datasets

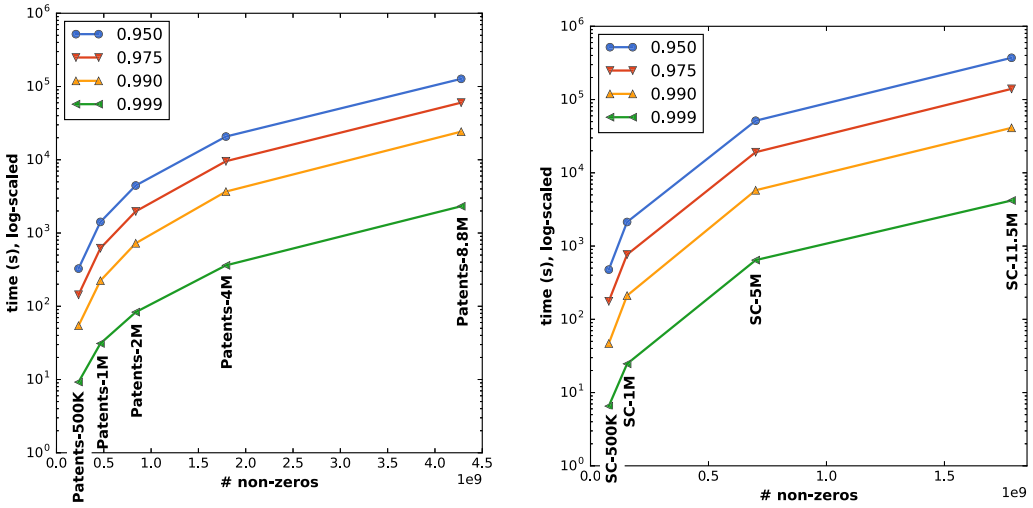

Fig. 11 Scaling characteristics of TAPNN in a near-duplicate detection scenario over the Patents-8.8M (*left*) and SC-11.5M (*right*) datasets

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflicts of interest.

References

1. Anastasiu, D.C., Karypis, G.: Efficient identification of tanimoto nearest neighbors. In: Proceedings of the 3rd IEEE International Conference on Data Science and Advanced Analytics, ser. DSAA '16 (2016)
2. Strehl, A., Ghosh, J.: Relationship-based clustering and visualization for high-dimensional data mining. INFORMS J. Comput. **15**(2), 208–230 (2003)
3. Joydeep, A.S., Strehl, E., Ghosh, J., Mooney, R.: Impact of similarity measures on web-page clustering. In: Workshop on Artificial Intelligence for Web Search (AAAI 2000). Citeseer (2000)
4. Banerjee, A., Ghosh, J.: Scalable clustering algorithms with balancing constraints. Data Min. Knowl. Discov. **13**(3), 365–395 (2006)
5. Huang, A.: Similarity measures for text document clustering. In: Proceedings of the Sixth New Zealand Computer Science Research Student Conference, ser. NZCSRSC2008, Christchurch, New Zealand, pp. 49–56 (2008)
6. Lyon, C., Malcolm, J., Dickerson, B.: Detecting short passages of similar text in large document collections. In: Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing, pp. 118–125 (2001)
7. Bao, J.-P., Malcolm, J.: Text similarity in academic conference papers. In: Proceedings of 2nd International Plagiarism Conference (2006)
8. Alzahrani, S.M., Salim, N., Abraham, A.: Understanding plagiarism linguistic patterns, textual features, and detection methods. Trans. Syst. Man Cybern. Part C **42**(2), 133–149 (2012)
9. Curran, J.R., Moens, M.: Improvements in automatic thesaurus extraction. In: Proceedings of the ACL-02 Workshop on Unsupervised Lexical Acquisition-vol. 9. Association for Computational Linguistics, pp. 59–66 (2002)
10. Strehl, A., Ghosh, J.: A Scalable Approach to Balanced, High-Dimensional Clustering of Market-Baskets. Springer, Berlin (2000)
11. Karypis, G.: 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: ACM, pp. 247–254 (2001)
12. Adam, N.R., Janeja, V.P., Atluri, V.: Neighborhood based detection of anomalies in high dimensional spatio-temporal sensor datasets. In: Proceedings of the 2004 ACM Symposium on Applied Computing, ser. SAC '04. New York, NY, USA: ACM, pp. 576–583 (2004)
13. Geppert, H., Vogt, M., Bajorath, J.: Current trends in ligand-based virtual screening: molecular representations, data mining methods, new application areas, and performance evaluation. J. Chem. Inf. Model. **50**(2), 205–216 (2010)
14. Keiser, M.J., Roth, B.L., Armbruster, B.N., Ernsberger, P., Irwin, B.K., Shoichet, John J.: Relating protein pharmacology by ligand chemistry. Nat. Biotechnol. **25**(2), 197–206 (2007)
15. Stahura, F.L., Bajorath, J.: Virtual screening methods that complement HTS. Comb. Chem. High Throughput Screen **7**(4), 259–269 (2004)
16. Kristensen, T.G.: Transforming tanimoto queries on real valued vectors to range queries in euclidian space. J. Math. Chem. **48**(2), 287–289 (2010)
17. Arif, S.M., Holliday, J.D., Willett, P.: Inverse frequency weighting of fragments for similarity-based virtual screening. J. Chem. Inf. Model. **50**(8), 1340–1349 (2010)
18. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York (2008)
19. Swamidass, S.J., Baldi, P.: Bounds and algorithms for fast exact searches of chemical fingerprints in linear and sublinear time. J. Chem. Inf. Model. **47**(2), 302–317 (2007)
20. Nasr, R., Hirschberg, D.S., Baldi, P.: Hashing algorithms and data structures for rapid searches of fingerprint vectors. J. Chem. Inf. Model. **50**(8), 1358–1368 (2010)
21. Tabei, Y., Tsuda, K.: Sketchsort: fast all pairs similarity search for large databases of molecular fingerprints. Mol. Inform. **30**(9), 801–807 (2011). doi:10.1002/minf.201100050
22. Kristensen, T.G., Nielsen, J., Pedersen, C.N.S.: Algorithms in Bioinformatics: 9th International Workshop, WABI 2009, Philadelphia, PA, USA, Sept 12–13, 2009. Proceedings. Berlin: Springer, 2009, ch. A Tree Based Method for the Rapid Screening of Chemical Fingerprints, pp. 194–205

23. Smellie, A.: Compressed binary bit trees: a new data structure for accelerating database searching. J. Chem. Inf. Model. **49**(2), 257–262 (2009)
24. Kristensen, T.G., Nielsen, J., Pedersen, C.N.S.: Using inverted indices for accelerating lingo calculations. J. Chem. Inf. Model. **51**(3), 597–600 (2011)
25. Thiel, P., Sach-Peltason, L., Ottmann, C., Kohlbacher, O.: Blocked inverted indices for exact clustering of large chemical spaces. J. Chem. Inf. Model. **54**(9), 2395–2401 (2014)
26. Chaudhuri, S., Ganti, V., Kaushik, R.: 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, p. 5 (2006)
27. Moffat, A., Sacks-davis, R., Wilkinson, R., Zobel, J.: Retrieval of partial documents. In: Information Processing and Management, pp. 181–190 (1994)
28. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, ser. WWW '07. New York: ACM, pp. 131–140 (2007)
29. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: Proceedings of the 17th International Conference on World Wide Web, ser. WWW '08. New York: ACM, pp. 131–140 (2008)
30. Xiao, C., Wang, W., Lin, X., Shang, H.: Top-k set similarity joins. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, ser. ICDE '09. Washington, DC: IEEE Computer Society, pp. 916–927 (2009)
31. Ribeiro, L.A., Härder, T.: Generalizing prefix filtering to improve set similarity joins. Inf. Syst. **36**(1), 62–78 (2011)
32. Awekar, A., Samatova, N.F.: 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—Vol. 01, ser. WI-IAT '09. Washington, DC: IEEE Computer Society, pp. 295–300 (2009)
33. Anastasiu, D.C., Karypis, G.: L2ap: fast cosine similarity search with prefix l-2 norm bounds. In: 30th IEEE International Conference on Data Engineering, ser. ICDE '14 (2014)
34. Lee, D., Park, J., Shim, J., Lee, S.-G.: An efficient similarity join algorithm with cosine similarity predicate. In: Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part II, ser. DEXA'10. Berlin, Heidelberg: Springer, pp. 422–436 (2010)
35. Kryszkiewicz, M.: Bounds on lengths of real valued vectors similar with regard to the tanimoto similarity. In: Intelligent Information and Database Systems, ser. Lecture Notes in Computer Science, Selamat, A., Nguyen, N., Haron, H., (eds). Springer, Berlin, 7802, pp. 445–454 (2013)
36. Kryszkiewicz, M.: Using non-zero dimensions for the cosine and tanimoto similarity search among real valued vectors. Fundam. Inform. **127**(1–4), 307–323 (2013)
37. Kryszkiewicz, M.: Using non-zero dimensions and lengths of vectors for the tanimoto similarity search among real valued vectors. In: Intelligent Information and Database Systems. Springer, Berlin, pp. 173–182 (2014)
38. Lewis, D.D., Yang, Y., Rose, T.G., Li, F.: Rcv1: a new benchmark collection for text categorization research. J. Mach. Learn. Res. **5**, 361–397 (2004)
39. Singh, N., Guha, R., Giulianotti, M.A., Pinilla, C., Houghten, R.A., Medina-Franco, J.L.: Chemoinformatic analysis of combinatorial libraries, drugs, natural products, and molecular libraries small molecule repository. J. Chem. Inf. Model. **49**(4), 1010–1024 (2009)
40. Papadatos, G., Davies, M., Dedman, N., Chambers, J., Gaulton, A., Siddle, J., Koks, R., Irvine, S.A., Pettersson, J., Goncharoff, N., Hersey, A., Overington, J.P.: Surechembl: a large-scale, chemically annotated patent document database. Nucleic Acids Res. **44**, D1220–D1228 (2016)
41. Porter, M.F.: An algorithm for suffix stripping. Program **14**(3), 130–137 (1980)
42. Wale, N., Watson, I.A., Karypis, G.: Indirect similarity based methods for effective scaffold-hopping in chemical compounds. J. Chem. Inf. Model. **48**, 730–741 (2008)
43. Wale, N., Karypis, G.: Acyclic subgraph based descriptor spaces for chemical compound retrieval and classification. In: Proceedings of the Sixth International Conference on Data Mining, ser. ICDM '06 (2006)
44. O'Boyle, N.M., Banck, M., James, C.A., Morley, C., Vandermeersch, T., Hutchison, G.R.: Open babel: an open chemical toolbox. J. Cheminform. **3**(1), 1–14 (2011)
45. Dong, W., Moses, C., Li, K.: 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: ACM, pp. 577–586 (2011)
46. Park, Y., Park, S., Lee, S.-G., Jung, W.: Greedy filtering: a scalable algorithm for k-nearest neighbor graph construction. In: Database Systems for Advanced Applications, ser. Lecture Notes in Computer Science. Springer, Berlin 8421, pp. 327–341 (2014)
47. Awekar, A., Samatova, N.F.: Parallel all pairs similarity search. In: Proceedings of the 10th International Conference on Information and Knowledge Engineering, ser. IKE '11 (2011)
48. Anastasiu, D.C., Karypis, G.: Pl2ap: fast parallel cosine similarity search. In: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, in conjunction with SC'15, ser. IA3. New York: ACM, 2015, pp. 1–8 (2015)
49. Anastasiu, D.C., Karypis, G.: Fast parallel cosine k-nearest neighbor graph construction. In: Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms, in Conjunction with SC'16, ser. IA3 2016. New York: ACM (2016)