Abstract

This paper presents a simple and efficient algorithm for approximate dictionary matching designed for similarity measures such as cosine, Dice, Jaccard, and overlap coefficients. We propose this algorithm, called CPMerge, for the $\tau$-overlap join of inverted lists. First we show that this task is solvable exactly by a $\tau$-overlap join. Given inverted lists retrieved for a query, the algorithm collects fewer candidate strings and prunes unlikely candidates to efficiently find strings that satisfy the constraint of the $\tau$-overlap join. We conducted experiments of approximate dictionary matching on three large-scale datasets that include person names, biomedical names, and general English words. The algorithm exhibited scalable performance on the datasets. For example, it retrieved strings in 1.1 ms from the string collection of Google Web1T unigrams (with cosine similarity and threshold 0.7).

1 Introduction

Languages are sufficiently flexible to be able to express the same meaning through different dictionary matching designed for similarity measures such as cosine, Dice, Jaccard, and overlap coefficients. We propose this algorithm, called CPMerge, for the $\tau$-overlap join of inverted lists. First we show that this task is solvable exactly by a $\tau$-overlap join. Given inverted lists retrieved for a query, the algorithm collects fewer candidate strings and prunes unlikely candidates to efficiently find strings that satisfy the constraint of the $\tau$-overlap join. We conducted experiments of approximate dictionary matching on three large-scale datasets that include person names, biomedical names, and general English words. The algorithm exhibited scalable performance on the datasets. For example, it retrieved strings in 1.1 ms from the string collection of Google Web1T unigrams (with cosine similarity and threshold 0.7).

1. We show that the problem of approximate dictionary matching is solved exactly by a $\tau$-overlap join (Sarawagi and Kirpal, 2004) of inverted lists. Then we present CPMerge, which is a simple and efficient algorithm for the $\tau$-overlap join. In addition, the algorithm is easily implemented.

2. We demonstrate the efficiency of the algorithm on three large-scale datasets with person names, biomedical concept names,
and general English words. We compare the algorithm with state-of-the-art algorithms, including Locality Sensitive Hashing (Ravichandran et al., 2005; Andoni and Indyk, 2008) and DivideSkip (Li et al., 2008). The proposed algorithm retrieves strings the most rapidly, e.g., in 1.1 ms from Google Web1T unigrams (with cosine similarity and threshold 0.7).

2 Proposed Method

2.1 Necessary and sufficient conditions

In this paper, we assume that the features of a string are represented arbitrarily by a set. Although it is important to design a string representation for an accurate similarity measure, we do not address this problem: our emphasis is not on designing a better representation for string similarity but on establishing an efficient algorithm.

The most popular representation is given by n-grams: all substrings of size n in a string. We use trigrams throughout this paper as an example of string representation. For example, the string “methyl sulphone” is expressed by 17 elements of letter trigrams1, {‘$’$’$’m’, ‘$’$’m’’, ‘met’, ‘eth’, ‘thy’, ‘hy’l’, ‘yl’, ‘s’$’’, ‘su’, ‘ulp’, ‘iph’, ‘pho’, ‘hon’, ‘one’, ‘ne$’, ‘e$$’}. We insert two $s$ before and after the string to denote the start or end of the string. In general, a string x consisting of |X| letters yields (|x| + n − 1) elements of n-grams. We call |x| and |X| the length and size, respectively, of the string x.

Let X and Y denote the feature sets of the strings x and y, respectively. The cosine similarity between the two strings x and y is,

$$\text{cosine}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X||Y|}}.$$  (2)

By integrating this definition with Equation 1, we obtain the necessary and sufficient condition for approximate dictionary matching,

$$\left[\alpha \sqrt{|X||Y|}\right] \leq |X \cap Y| \leq \min\{|X|, |Y|\}.$$  (3)

This inequality states that two strings x and y must have at least \( \tau = \left[\alpha \sqrt{|X||Y|}\right] \) features in common. When ignoring \( |X \cap Y| \) in the inequality, we have an inequality about \( |X| \) and \( |Y| \),

$$\left[\alpha^2 |X|\right] \leq |Y| \leq \left[\frac{|X|}{\alpha^2}\right].$$  (4)

This inequality presents the search range for retrieving similar strings; that is, we can ignore strings whose feature size is out of this range. Other derivations are also applicable to similarity measures, including Dice, Jaccard, and overlap coefficients. Table 1 summarizes the conditions for these similarity measures.

We explain one usage of these conditions. Let query string x = “methyl sulphone” and threshold for approximate dictionary matching \( \alpha = 0.7 \) with cosine similarity. Representing the strings with letter trigrams, we have the size of x, |X| = 17. The inequality 4 gives the search range of \( |Y| \) of the retrieved strings, 9 ≤ |Y| ≤ 34. Presuming that we are searching for strings of \( |Y| = 16 \), we obtain the necessary and sufficient condition for the approximate dictionary matching from the inequality 3, \( \tau = 12 \leq |X \cap Y| \). Thus, we need to search for strings that have at least 12 letter trigrams that overlap with X. When considering a string y = “methyl sulfone”, which is a spelling variant of y (ph → f), we confirm that the string is a solution for approximate dictionary matching because \( |X \cap Y| = 13 \geq \tau \). Here, the actual similarity is \( \text{cosine}(X, Y) = 13/\sqrt{17 \times 16} = 0.788 \geq \alpha \).

2.2 Data structure and algorithm

Algorithm 1 presents the pseudocode of the approximate dictionary matching based on Table 1.
Given a query string $x$, a collection of strings $V$, and a similarity threshold $\alpha$, the algorithm computes the size range (line 3) given by Table 1. For each size $l$ in the range, the algorithm computes the minimum number of overlaps $\tau$ (line 4). The function overlapjoin (line 5) finds similar strings by solving the following problem ($\tau$-overlap join): given a list of features of the query string $X$ and the minimum number of overlaps $\tau$, enumerate strings of size $l$ in the collection $V$ such that they have at least $\tau$ feature overlaps with $X$.

To solve this problem efficiently, we build an inverted index that stores a mapping from the features to their originating strings. Then, we can perform the $\tau$-overlap join by finding strings that appear at least $\tau$ times in the inverted lists retrieved for the query features $X$.

Algorithm 2 portrays a naïve solution for the $\tau$-overlap join (AllScan algorithm). In this algorithm, function $\text{get}(V, l, q)$ returns the inverted list of strings (of size $l$) for the feature $q$. In short, this algorithm scans strings in the inverted lists retrieved for the query features $X$, counts the frequency of occurrences of every string in the inverted lists, and returns the strings whose frequency of occurrences is no smaller than $\tau$.

This algorithm is inefficient in that it scans all strings in the inverted lists. The number of scanned strings is large, especially when some query features appear frequently in the strings, e.g., ‘s$$’ (words ending with ‘s’) and ‘pre’ (words with substring ‘pre’). To make matters worse, such features are too common for characterizing string similarity. The AllScan algorithm is able to maintain numerous candidate strings in $M$, but most candidates are not likely to qualified because they have few overlaps with $X$.

To reduce the number of the candidate strings, we refer to signature-based algorithms (Arasu et al., 2006; Chaudhuri et al., 2006):

**Property 1** Let there be a set (of size $h$) $X$ and a set (of any size $Y$). Consider any subset $Z \subseteq X$ of size $(h - \tau + 1)$. If $|X \cap Y| \geq \tau$, then $Z \cap Y \neq \phi$.

We explain one usage of this property. Let query string $x$ = “methyl sulphone” and its trigram set $X$ be features (therefore, $|X| = h = 17$). Presuming that we seek strings whose trigrams are size 16 and have 12 overlaps with $X$, then string $y$ must have at least one overlap with any subset of size 6 ($= 17 - 12 + 1$) of $X$. We call the subset signatures. The property leads to an algorithmic design by which we obtain a small set of candidate strings from the inverted lists for signatures, $(|X| - \tau + 1)$ features in $X$, and verify whether each candidate string satisfies the $\tau$ overlap with the remaining $(\tau - 1)$-grams.

Algorithm 3 presents the pseudocode employing this idea. In line 1, we arrange the features in $X$ in ascending order of the number of strings in their inverted lists. We denote the $k$-th element in the ordered features as $X_k$ ($k \in \{0, \ldots, |X| - 1\}$), where the index number begins with 0. Based on this notation, $X_0$ and $X_{|X|-1}$ are the most uncommon and the most common features in $X$, respectively.

In lines 2−7, we use $(|X| - \tau + 1)$ features
X₀, ..., X_{|X|−τ} to generate a compact set of candidate strings. The algorithm stores the occurrence count of each string s in M[s]. In lines 9–21, we increment the occurrence counts if each of X_{|X|−τ+1}, ..., X_{|X|−1} inverted lists contain the candidate strings. For each string s in the candidates (line 10), we perform a binary search on the inverted list (line 11), and increment the overlap count if the string s exists (line 12). If the overlap counter of the string reaches \( τ \) (line 14), then we append the string s to the result list \( R \) and remove s from the candidate list (lines 15–16). We prune a candidate string (lines 17–18) if the candidate is found to be unreachable for \( τ \) overlaps even if it appears in all of the unexamined inverted lists.

3 Experiments

We report the experimental results of approximate dictionary matching on large-scale datasets with person names, biomedical names, and general English words. We implemented various systems of approximate dictionary matching.

- **Proposed**: CPMerge algorithm.
- **Naive**: Naive algorithm that computes the cosine similarity \( |V| \) times for every query.
- **AllScan**: AllScan algorithm.
- **Signature**: CPMerge algorithm without pruning: this is equivalent to Algorithm 3 without lines 17–18.
- **DivideSkip**: our implementation of the algorithm (Li et al., 2008)².
- **Locality Sensitive Hashing (LSH)** (Andoni and Indyk, 2008): This baseline system follows the design of previous work (Ravichandran et al., 2005). This system approximately solves Equation 1 by finding dictionary entries whose LSH values are within the (bit-wise) hamming distance of \( θ \) from the LSH value of a query string. To adapt the method to approximate dictionary matching, we used a 64-bit LSH function computed with letter trigrams. By design, this method does not find an exact solution to Equation 1; in other words, the method can miss dictionary entries that are actually similar to the query strings. This system has three parameters, \( θ, q \) (number of bit permutations), and \( B \) (search width), to control the tradeoff between retrieval speed and recall³. Generally speaking, increasing these parameters improves the recall, but slows down the speed. We determined \( θ = 24 \) and \( q = 24 \) experimentally⁴, and measured the performance when \( B \in \{16, 32, 64\} \).

The systems, excluding LSH, share the same implementation of Algorithm 1 so that we can specifically examine the differences of the algorithms for \( τ \)-overlap join. The C++ source code of the system used for this experiment is available⁵.

We ran all experiments on an application server running Debian GNU/Linux 4.0 with Intel Xeon 5140 CPU (2.33 GHz) and 8 GB main memory.

²We tuned parameter values \( μ \in \{0.01, 0.02, 0.04, 0.1, 0.2, 0.4, 1, 2, 4, 10, 20, 40, 100\} \) for each dataset. We selected the parameter with the fastest response.
³We followed the notation of the original paper (Ravichandran et al., 2005) here. Refer to the original paper for definitions of the parameters \( θ, q \), and \( B \).
⁴q was set to 24 so that the arrays of shuffled hash values are stored in memory. We chose \( θ = 24 \) from \( \{8, 16, 24\} \) because it showed a good balance between accuracy and speed.
⁵http://www.chokkan.org/software/simstring/
3.1 Datasets
We used three large datasets with person names (IMDB actors), general English words (Google Web1T), and biomedical names (UMLS).

- **IMDB actors**: This dataset comprises actor names extracted from the IMDB database\(^6\). We used all actor names (1,098,022 strings; 18 MB) from the file actors.list.gz. The average number of letter trigrams in the strings is 17.2. The total number of trigrams is 42,180. The system generated index files of 83 MB in 56.6 s.

- **Google Web1T unigrams**: This dataset consists of English word unigrams included in the Google Web1T corpus (LDC2006T13). We used all word unigrams (13,588,391 strings; 121 MB) in the corpus after removing the frequency information. The average number of letter trigrams in the strings is 10.3. The total number of trigrams is 301,459. The system generated index files of 601 MB in 551.7 s.

- **UMLS**: This dataset consists of English names and descriptions of biomedical concepts included in the Unified Medical Language System (UMLS). We extracted all English concept names (5,216,323 strings; 212 MB) from MRCONSO.RRF.aa.gz and MRCONSO.RRF.ab.gz in UMLS Release 2009AA. The average number of letter trigrams in the strings is 43.6. The total number of trigrams is 171,596. The system generated index files of 1.1 GB in 1216.8 s.

For each dataset, we prepared 1,000 query strings by sampling strings randomly from the dataset. To simulate the situation where query strings are not only identical but also similar to dictionary entries, we introduced random noise to the strings. In this experiment, one-third of the query strings are unchanged from the original (sampled) strings, one-third of the query strings have one letter changed, and one-third of the query strings have two letters changed. When changing a letter, we randomly chose a letter position from a uniform distribution, and replaced the letter at the position with an ASCII letter randomly chosen from a uniform distribution.

3.2 Results
To examine the scalability of each system, we controlled the number of strings to be indexed from 10%–100%, and issued 1,000 queries. Figure 1 portrays the average response time for retrieving strings whose cosine similarity values are no smaller than 0.7. Although LSH (\(B=16\)) seems to be the fastest in the graph, this system missed many true positives\(^7\); the recall scores of approximate dictionary matching were 15.4% (IMDB), 13.7% (Web1T), and 1.5% (UMLS). Increasing the parameter \(B\) improves the recall at the expense of the response time. LSH (\(B=64\))\(^8\). It not only ran slower than the proposed method, but also suffered from low recall scores, 25.8% (IMDB), 18.7% (Web1T), and 7.1% (UMLS). LSH was useful only when we required a quick response much more than recall.

The other systems were guaranteed to find the exact solution (100% recall). The proposed algorithm was the fastest of all exact systems on all datasets: the response times per query (100% index size) were 1.07 ms (IMDB), 1.10 ms (Web1T), and 20.37 ms (UMLS). The response times of the Naïve algorithm were too slow, 32.8 s (IMDB), 236.5 s (Web1T), and 416.3 s (UMLS).

The proposed algorithm achieved substantial improvements over the AllScan algorithm: the proposed method was 65.3 times (IMDB), 227.5 times (Web1T), and 13.7 times (UMLS) faster than the Naïve algorithm. We observed that the Signature algorithm, which is Algorithm 3 without lines 17–18, did not perform well: The Signature algorithm was 1.8 times slower (IMDB), 2.1 times faster (Web1T), and 135.0 times slower (UMLS) than the AllScan algorithm. These results indicate that it is imperative to minimize the number of candidates to reduce the number of binary-search operations. The proposed algorithm was 11.1–13.4 times faster than DivideSkip.

Figure 2 presents the average response time

\(^6\)ftp://ftp.fu-berlin.de/misc/movies/database/

\(^7\)Solving Equation 1, all systems are expected to retrieve the exact set of strings retrieved by the Naïve algorithm.

\(^8\)The response time of LSH (\(B=64\)) on the IMDB dataset was 29.72 ms (100% index size).
of the proposed algorithm for different similarity
measures and threshold values. When the similar-
ity threshold is lowered, the algorithm runs slower
because the number of retrieved strings |Y| in-
creases exponentially. The Dice coefficient and
cosine similarity produced similar curves.

Table 2 summarizes the run-time statistics of
the proposed method for each dataset (with co-
sine similarity and threshold 0.7). Using the
IMDB dataset, the proposed method searched for
strings whose size was between 8.74 and 34.06;
it retrieved 4.63 strings per query string. The
proposed algorithm scanned 279.7 strings in 4.6
inverted lists to obtain 232.5 candidate strings.
The algorithm performed a binary search on 4.3
inverted lists containing 7,561.8 strings in all.
In contrast, the AllScan algorithm had to scan
16,155.1 strings in 17.7 inverted lists and con-
sidered 9,788.7 candidate strings, and found only
4.63 similar strings.

This table clearly demonstrates three key con-
tributions of the proposed algorithm for efficient
approximate dictionary matching. First, the pro-
posed algorithm scanned far fewer strings than did
the AllScan algorithm. For example, to obtain
candidate strings in the IMDB dataset, the pro-
posed algorithm scanned 279.7 strings, whereas
the AllScan algorithm scanned 16,155.1 strings.
Therefore, the algorithm examined only 1.1%–
3.5% of the strings in the entire inverted lists in
the three datasets. Second, the proposed algo-

9These values are 4.6 + 4.3, 3.1 + 2.9, and 14.3 + 17.4.
Table 2: Run-time statistics of the proposed algorithm for each dataset

<table>
<thead>
<tr>
<th>Averaged item</th>
<th>IMDB</th>
<th>Web1T</th>
<th>UMLS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min (</td>
<td>y</td>
<td>)</td>
<td>8.74</td>
<td>5.35</td>
</tr>
<tr>
<td>max (</td>
<td>y</td>
<td>)</td>
<td>34.06</td>
<td>20.46</td>
</tr>
<tr>
<td>(\tau)</td>
<td>14.13</td>
<td>9.09</td>
<td>47.77</td>
<td>minimum number of overlaps required/sufficient per query</td>
</tr>
<tr>
<td>(</td>
<td>D</td>
<td>)</td>
<td>4.63</td>
<td>3.22</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td>— averaged for each query and target size:</td>
</tr>
<tr>
<td># inverted lists</td>
<td>17.7</td>
<td>11.2</td>
<td>61.1</td>
<td>number of inverted lists retrieved for a query</td>
</tr>
<tr>
<td># strings</td>
<td>16 155.1</td>
<td>52 557.6</td>
<td>49 561.4</td>
<td>number of strings in the inverted list</td>
</tr>
<tr>
<td># unique strings</td>
<td>9 788.7</td>
<td>44 834.6</td>
<td>17 457.5</td>
<td>number of unique strings in the inverted list</td>
</tr>
<tr>
<td><strong>Candidate stage</strong></td>
<td></td>
<td></td>
<td></td>
<td>— averaged for each query and target size:</td>
</tr>
<tr>
<td># inverted lists</td>
<td>4.6</td>
<td>3.1</td>
<td>14.3</td>
<td>number of inverted lists scanned for generating candidates</td>
</tr>
<tr>
<td># strings</td>
<td>279.7</td>
<td>522.7</td>
<td>1 756.3</td>
<td>number of strings scanned for generating candidates</td>
</tr>
<tr>
<td># candidates</td>
<td>232.5</td>
<td>523.7</td>
<td>1 149.7</td>
<td>number of candidates generated for a query</td>
</tr>
<tr>
<td><strong>Validation stage</strong></td>
<td></td>
<td></td>
<td></td>
<td>— averaged for each query and target size:</td>
</tr>
<tr>
<td># inverted lists</td>
<td>4.3</td>
<td>2.9</td>
<td>17.4</td>
<td>number of inverted lists examined by binary search for a query</td>
</tr>
<tr>
<td># strings</td>
<td>7 561.8</td>
<td>19 843.6</td>
<td>20 443.7</td>
<td>number of strings targeted by binary search</td>
</tr>
</tbody>
</table>

4 Related Work

Numerous studies have addressed approximate dictionary matching. The most popular configuration uses \(n\)-grams as a string representation and the edit distance as a similarity measure. Grassano et al. (1998; 2001) presented various filtering strategies, e.g., count filtering, position filtering, and length filtering, to reduce the number of candidates. Kim et al. (2005) proposed two-level \(n\)-gram inverted indices (\(n\)-Gram/2L) to eliminate the redundancy of position information in \(n\)-gram indices. Li et al. (2007) explored the use of variable-length grams (VGRAMs) for improving the query performance. Lee et al. (2007) extended \(n\)-grams to include wild cards and developed algorithms based on a replacement semi-lattice. Xiao et al. (2008) proposed the Ed-Join algorithm, which utilizes mismatching \(n\)-grams.

Several studies addressed different paradigms for approximate dictionary matching. Bocek et al. (2007) presented the Fast Similarity Search (FastSS), an enhancement of the neighborhood generation algorithms, in which multiple variants of each string record are stored in a database. Wang et al. (2009) further improved the technique of neighborhood generation by introducing partitioning and prefix pruning. Huynh et al. (2006) developed a solution to the \(k\)-mismatch problem in compressed suffix arrays. Liu et al. (2008) stored string records in a trie, and proposed a framework called TITAN. These studies are specialized for the edit distance measure.

A few studies addressed approximate dictionary matching for similarity measures such as cosine and Jaccard similarities. Chaudhuri et al. (2006) proposed the SSJoin operator for similarity joins with several measures including the edit distance and Jaccard similarity. This algorithm first generates signatures for strings, finds all pairs of strings whose signatures overlap, and finally outputs the subset of these candidate pairs that satisfy the similarity predicate. Arasu et al. (2006) addressed signature schemes, i.e., methodologies for obtaining signatures from strings. They also presented an implementation of the SSJoin operator in SQL. Although we did not implement this algorithm in SQL, it is equivalent to the Signature algorithm in Section 3.

Sarawagi and Kirpal (2004) proposed the MergeOpt algorithm for the \(\tau\)-overlap join to approximate string matching with overlap, Jaccard, and cosine measures. This algorithm splits inverted lists for a given query \(A\) into two groups, \(S\) and \(L\), maintains a heap to collect candidate strings on \(S\), and performs a binary search on \(L\) to verify the condition of the \(\tau\)-overlap join for each candidate string. Their subsequent work includes an efficient algorithm for the top-\(k\) search of the overlap join (Chandel et al., 2006).

Li et al. (2008) extended this algorithm to the SkipMerge and DivideSkip algorithms. The SkipMerge algorithm uses a heap to compute the \(\tau\)-overlap join on entire inverted lists \(A\), but has an additional mechanism to increment the fron-
tier pointers of inverted lists efficiently based on the strings popped most recently from the heap. Consequently, SkipMerge can reduce the number of strings that are pushed to the heap. Similarly to the MergeOpt algorithm, DivideSkip splits inverted lists \( A \) into two groups \( S \) and \( L \), but it applies SkipMerge to \( S \). In Section 3, we reported the performance of DivideSkip.

Charikar (2002) presented the Locality Sensitive Hash (LSH) function (Andoni and Indyk, 2008), which preserves the property of cosine similarity. The essence of this function is to map strings into \( N \)-bit hash values where the bitwise hamming distance between the hash values of two strings approximately corresponds to the angle of the two strings. Ravichandran et al. (2005) applied LSH to the task of noun clustering. Adapting this algorithm to approximate dictionary matching, we discussed its performance in Section 3.

Several researchers have presented refined similarity measures for strings (Winkler, 1999; Cohen et al., 2003; Bergsma and Kondrak, 2007; Davis et al., 2007). Although these studies are sometimes regarded as a research topic of approximate dictionary matching, they assume that two strings for the target of similarity computation are given; in other words, it is out of their scope to find strings in a large collection that are similar to a given string. Thus, it is a reasonable approach for an approximate dictionary matching to quickly collect candidate strings with a loose similarity threshold, and for a refined similarity measure to scrutinize each candidate string for the target application.

5 Conclusions

We present a simple and efficient algorithm for approximate dictionary matching with the cosine, Dice, Jaccard, and overlap measures. We conducted experiments of approximate dictionary matching on large-scale datasets with person names, biomedical names, and general English words. Even though the algorithm is very simple, our experimental results showed that the proposed algorithm executed very quickly. We also confirmed that the proposed method drastically reduced the number of candidate strings considered during approximate dictionary matching. We believe that this study will advance practical NLP applications for which the execution time of approximate dictionary matching is critical.

An advantage of the proposed algorithm over existing algorithms (e.g., MergeSkip) is that it does not need to read all the inverted lists retrieved by query \( n \)-grams. We observed that the proposed algorithm solved \( \tau \)-overlap joins by checking approximately half of the inverted lists (with cosine similarity and threshold \( \alpha = 0.7 \)). This characteristic is well suited to processing compressed inverted lists because the algorithm needs to decompress only half of the inverted lists. It is natural to extend this study to compressing and decompressing inverted lists for reducing disk space and for improving query performance (Behm et al., 2009).

Acknowledgments

This work was partially supported by Grants-in-Aid for Scientific Research on Priority Areas (MEXT, Japan) and for Solution-Oriented Research for Science and Technology (JST, Japan).

References


