

Efficient Batch Top-k Search for Dictionary-based Entity Recognition

Amit Chandel
achandel@cse.iitb.ac.in
IIT Bombay

P. C. Nagesh
nagesh@it.iitb.ac.in
IIT Bombay

Sunita Sarawagi
sunita@iitb.ac.in
IIT Bombay

Abstract

We consider the problem of speeding up Entity Recognition systems that exploit existing large databases of structured entities to improve extraction accuracy. These systems require the computation of the maximum similarity scores of several overlapping segments of the input text with the entity database. We formulate a Batch-Top-K problem with the goal of sharing computations across overlapping segments. Our proposed algorithm performs a factor of three faster than independent Top-K queries and only a factor of two slower than an unachievable lower bound on total cost. We then propose a novel modification of the popular Viterbi algorithm for recognizing entities so as to work with easily computable bounds on match scores, thereby reducing the total inference time by a factor of eight compared to state-of-the-art methods.

1 Introduction

The extraction of structured entities from unstructured text is a challenging problem encountered in many applications such as data warehousing, web data integration and bio-informatics. For example, during warehouse data cleaning a common operation is to extract from address strings structured attributes like street names, city names and addresses [2, 1]. A lot of work has been done in this area starting from early rule-based systems [4] to the current highly flexible and powerful conditional graphical models [18].

Typical NER¹ systems rely on various properties of the word and regular expression pattern surrounding it to find the entity of interest. Another valuable resource is match with an existing database of the entity of interest. For example, as Citeseer attempts to identify journal and author names from html pages, match of a text string with author, journal and title entities of existing structured databases like DBLP² and Bibtex servers³, can provide strong evidence

¹NER- Named Entity Recognition [3, 15, 6]

²<http://dblp.uni-trier.de>

³<http://citeseer.ist.psu.edu>

about the entity type of the string. Several independent experimental studies have found that the accuracy of extraction can improve significantly when coupled with an external database [3, 6, 1]. However match with an external database is hard as the form of a name in unstructured text varies substantially from its database version. Recently, this problem has been addressed in [6], that proposes a more continuous notion of match through text similarity measures like TF-IDF-based cosine similarity [20]. Though such continuous match scores have been shown to improve accuracy, they come at the cost of increased extraction time. The time overhead becomes more noticeable as applications that depend on automatic extraction start getting deployed on a large scale and database sizes increase.

In this paper we show how to improve the performance of models that rely on similarity scores with existing structured databases to identify entities in unstructured text. We first present an overview of dictionary-based entity recognition systems and then outline our main contributions in this paper.

2 Dictionary-based entity recognition

We are given an unstructured text string \mathbf{x} consisting of a sequence of tokens $x_1 \dots x_n$ where each token is either a word or a delimiter. Let Y denote the set of entity types (such as title, person-names and city-names) to be recognized from \mathbf{x} . The entity recognition task is to segment \mathbf{x} into a sequence \mathbf{s} of segments $s_1 \dots s_p$ where each segment s_j is either labeled with an entity type from Y or a special label `other` denoting none of entities. For ease of notation, we assume Y includes the `other` label. Thus, each segment s_j is associated with a start position t_j , an end position u_j and a label $y_j \in Y$. Further, since adjacent segments about t_j and u_j always satisfy $1 \leq t_j \leq u_j \leq |\mathbf{x}|$, $t_{j+1} = u_j + 1$, $u_p = |\mathbf{x}|$, and $t_1 = 1$.

Intuitively, we recognize entities in unstructured text based on various simultaneous clues in and around the proposed segment such as capitalization patterns and delimiters. Other clues like the implicit ordering of labels and matches with known list of entities are also utilized.

A number of methods have been proposed [14, 10, 17, 2, 11, 18, 6, 15] to exploit all these clues in a combined manner to extract the entity of interest. We describe a state-of-the-art method for extraction called a semi-Markov model that is elegant, provides high accuracy and is flexible enough to include a diverse set of clues. In this model each segment $s_j = (t_j, u_j, y)$ is associated with a set of features that capture various properties of the segment when it is assigned a label y and its previous segment is assigned label y' . Thus, features are real-valued functions of the form: $g(y, y', \mathbf{x}, t_j, u_j)$ and each segment is associated with several such feature functions $g^1 \dots g^v$. During training each feature g^k is assigned a weight w^k that intuitively captures the importance of the feature. Details of the training process are not relevant to this work.

Examples of features: We present some examples of common features.

First word of author names is capitalized: $g^7(y, y', \mathbf{x}, t, u) = \llbracket x_t \text{ is capitalized} \rrbracket \cdot \llbracket y = \text{Author} \rrbracket$

The token following titles is a dot:

$g^8(y, y', \mathbf{x}, t, u) = \llbracket x_{u+1} \text{ is dot} \rrbracket \cdot \llbracket y = \text{Title} \rrbracket$

The Title segment is just before the Journal segment:

$g^{11}(y, y', \mathbf{x}, t, u) = \llbracket y \text{ is Journal} \rrbracket \cdot \llbracket y' = \text{Title} \rrbracket$

Author segments consist of two capitalized initials and a capitalized word: $g^{12}(y, y', \mathbf{x}, 3, 5) = \llbracket x_3 x_4 x_5 \approx X. X. Xx+ \rrbracket \cdot \llbracket y_i = \text{Author} \rrbracket$

The maximum TF-IDF match of a author segment with an entry in the personDB database: $g^{18}(y, y', \mathbf{x}, 3, 5) = \max_{r \in \text{personDB}} \text{TF-IDF}(x_3 x_4 x_5, r) \cdot \llbracket y = \text{Author} \rrbracket$

In this set the first four features are boolean whereas the last one returns continuous values between 0 and 1. More importantly, the first four are typical entity recognition features that can be computed efficiently by simply considering a small neighbourhood of tokens around the segment boundary (t u). In contrast, the last one requires us to find the person name entry that has the maximum TF-IDF similarity score over the database of people names. If the database, gets large, we can expect the computation of this feature to require significantly more time than the other four features. In general, one can define features corresponding to Top-K such matches and not just the maximum.

Best segmentation The goal during entity recognition is to break the input \mathbf{x} into a sequence of labeled segments such that the weighted sum of features that are fired is maximized. That is, we need to find a s^* such that

$$s^* = \underset{s: (y_j, t_j, u_j)}{\operatorname{argmax}} \sum_{s: (y_j, t_j, u_j)} \mathbf{W} \cdot \mathbf{g}(y_j, y_{j-1}, \mathbf{x}, t_j, u_j)$$

A brute force approach would require the enumeration of

all possible segmentation — an exponential number. Fortunately, the special form of the function, makes it possible to design an efficient dynamic programming algorithm as follows:

Let L be an upper bound on segment length. Let $s_{i:y}$ denote the set of all partial segmentation starting from 1 (the first index of the sequence) to i , such that the last segment has the label y and ending position i . Let $\delta(i, y)$ denote the largest value of the sum of segment scores for any $s' \in s_{i:y}$. We can now find the optimal segmentation by solving the famous Viterbi equations defined as follows[eq. 1]:

$$\delta(i, y) = \begin{cases} \max_{y', i-L < i' \leq i} \{ \delta(i' - 1, y') \\ + \mathbf{W} \cdot \mathbf{g}(y, y', \mathbf{x}, i', i) \} & \text{if } i > 0 \\ 0 & \text{if } i = 0 \\ -\infty & \text{if } i < 0 \end{cases} \quad (1)$$

The algorithm makes a forward scan of the input tokens, and for each token position i and entity label y computes the best segmentation from 1 to i by taking the maximum over all possible segment lengths of the last segment ending at i and all possible labels of the segment before the last. The best segmentation s^* then corresponds to the path traced by $\max_y \delta(|\mathbf{x}|, y)$. This requires $O(nm^2L)$ time where $m = |Y|$ and $O(mnL)$ space. Since L and m are small integers, this makes the algorithm linear in the length of the input and has been traditionally considered to be a fast algorithm.

However, as we attempt to exploit matches with large structured databases as one of the features of segments, an important concern becomes the time taken and number of Top-K queries that this entails. For an input string of n tokens, for each structured database column this will involve nL Top-K queries. Since most other features in typical extraction systems are lightweight functions of the input segment, the match features start dominating the total inference cost. For example, an entire Viterbi run on a citation record takes 0.8 seconds without dictionary features. If we add features based on lookup in a DBLP database with 100,000 *title* entries, the time for best segmentation goes to 4.2 seconds — a factor of five blowup.

In Section 3, we propose algorithms for batching up Top-K searches so that the time required is significantly smaller than doing independent Top-K queries for each segment. Later, in Section 4 we modify the Viterbi algorithm so that we do not require the computation of exact match scores for all input segments.

3 Batched dictionary lookups (Batch Top-k)

In this section we address the problem of finding the Top-K match scores for each possible segment of length no more than L of an input token sequence \mathbf{x} . These scores are used for designing match features that are needed during both training and testing of the extraction models. We measure similarity using the popular TF-IDF [eq.2] similarity score, which has been found to be highly effective in text searches.

The TF-IDF similarity between two text records r_1 and r_2 is defined as follows:

$$\begin{aligned} \text{TF-IDF}(r_1, r_2) &= \sum_{t \in r_1 \cap r_2} V(t, r_1) V(t, r_2) \quad (2) \\ V(t, r) &= \frac{V'(t, r)}{\sum_{t' \in r} V'(t', r)^2} \\ V'(t, r) &= \log(\text{TF}(t, r) + 1) \log(\text{IDF}(t)) \end{aligned}$$

In the above, the IDF term makes the weight of a token inversely proportional to its frequency in the database and the TF term makes it proportional to its frequency in the record. Intuitively, this assigns low scores to frequent tokens (stop-tokens) and high scores to rare tokens.

We use a threshold ϵ to limit matches only to values greater than ϵ because features with very low scores are not useful and unnecessarily increase running time and have even been found to reduce accuracy in some cases. Even distinctly dissimilar record pairs have non-zero positive similarity scores due to the presence of stop-tokens.

The basic Top-K search problem with the TF-IDF similarity measure is extensively researched [7, 5, 19, 20] in the information retrieval and database literature. Thus, a simple mechanism of solving our Batch-Top-K problem is to invoke the basic Top-K algorithm for each and every segment of length less than or equal to L in the input sequence. However, since segments have overlapping tokens there is scope for significant gains by batching their computations. The state-of-the-art Top-K algorithms are highly optimized to balance the number of tidlist⁴ fetches with record fetches via lower and upper bound score estimates. Utilizing these optimizations along with effectively sharing the partial and final results of one sub-query in the evaluation of another is a challenging problem.

We first present a highly optimized Top-K search algorithm that combines ideas from several Top-K and range search papers [7, 5, 19, 16]. We then present our algorithm for the batch Top-K search problem.

3.1 Simple Top-k

Given a relation R , and an input query string x of tokens, the simple Top-K search problem seeks to find the records in R with the k largest TF-IDF matches with x provided the match score is greater than a threshold ϵ .

As in all existing methods [5, 19], we assume an inverted index on R where for each token t we maintain a list of ordered pairs $(r, V(t, r))$ of all record identifier (rid) r in R that contain the token t . We call this the tidlist of token t . Since in general, the database can be quite large we assume that the index is stored in a relational database. The inverted index is a relation with records consisting of a token-id t ,

⁴tidlist refers to list of tuple-ids containing a given token. This is analogous to document-id list in a inverted-word index used in IR applications.

INPUT Query String $Q = q_1 q_2 \dots q_n$, k and ϵ

- 1: $X = x_1 x_2 \dots x_n$ = tokens of Q sorted in decreasing order of $V(q_i, Q)$
- 2: $\max V(t) = \max_r \{V(t, r)\}$
//tidlist(t)[cursor]=value in tidlist(t) at position=cursor
- 3: find min p s.t. $\sum_{p <= i <= n} V(x_i, X) * \max V(x_i) < \epsilon$
- 4: $\forall t \in S$, fetch tidlist(t) where $S = \{x_i | 1 <= i < p\}$
- 5: $\text{bestOffset} = \sum_{\forall t \notin S} V(t, X) * \max V(t)$
{ process the tidlists in S , row-wise }
- 6: **for** row = 1 to $\max_{\forall t \in S} \{\text{tidlist}(t)\}$ **do**
- 7: $\text{curMax} = \sum_{\forall t \in S} V(t, X) * \text{tidlist}(t)[\text{cursor}]$
- 8: **if** $\text{curMax} + \text{bestOffset} < \max(\epsilon, \text{topK}[k])$ **then**
- 9: **break**
- 10: **end if**
- 11: fetch $(r, V(t, r)) = \text{tidlist}(t)[\text{cursor}]$, $\forall t \in S$, and update $\text{score}(r) += V(t, X) * V(t, r)$
- 12: UpdateTopK(r , $\text{score}(r)$) and add r to CandidateSet
- 13: **end for**
- 14: **if** $\text{bestOffset} > 0$ OR NOT(all tidlists were fully merged) **then**
- 15: best estimate(r) = $\text{score}(r) + \text{curMax} + \text{bestOffset}$
- 16: $\forall r \in \text{CandidateSet}$ s.t best estimate(r) $> \max(\epsilon, \text{topK}[k])$ fetch r from database, find Similarity Score (r) and add to finalSet
- 17: **else**
- 18: set CandidateSet to final set
- 19: **end if**
- 20: return the Top-K records from the final set.
{**Procedure: UpdateTopK(record id r, score)**
//This function maintains an ordered list topK of top K values among the currently seen items}

Algorithm 1. Simple Top-k

a rid r and its score $V(t, r)$. We rely on suitably defined database indices to get for a given token t its tidlist in decreasing order of $V(t, r)$ values. For each token t , we maintain in a separate table aggregate statistics about the token's IDF value and the maximum $V(t, r)$ of any record containing the token. This table is small and can be assumed to be cached in the application.

The simple Top-K algorithm is described in Algorithm 1. We first sort tokens of x in decreasing order of their IDF values and then divide them into two parts 'strong' and 'weak' (as suggested in [16]) where the weak set consists of the largest suffix of the sorted list whose maximum sum of scores is less than ϵ . Intuitively, this ensures that all rids with match $> \epsilon$ appear at least in one strong list. We expect most of the stop-tokens to go to the weak set. We then open cursors on tidlists of each of the n' tokens in the 'strong' set in sorted order of $V(t, r)$ values. This creates n' lists of pairs $(r, V(t, r))$ and we merge them in a manner similar to Fagin's rank merging algorithm [7] to get a set of candidate records. The merging algorithm exploits the sorted nature of the tidlists to estimate upper bounds on scores of unseen records so that we can stop advancing on cursors if the estimated upper bound score is less than the Top-K partial scores of seen records or ϵ . Similarly, any record from the candidate set with upper bound score less than ϵ is removed. The remaining records are then fetched from

database (point query), their final similarity scores computed, and Top-K among them are returned.

Our approach allows us to effectively utilize the distribution of scores in tidlists and limit the size of candidate set. Though total tidlist fetch time is not affected, the total time spent in point queries is significantly reduced. Also we do not make a complete parallel sorted access on all columns as in Fagin’s algorithm, as we do not fetch tidlists from ‘weak’ segment.

3.2 Batch Top-K algorithm

In the batch Top-K setting our goal is to find the Top-K match for each possible subsequence of tokens in a text record $\mathbf{x} = x_1 \dots x_n$. Given a maximum segment length L , for each segment sub-query $Q = q_i q_{i+1} \dots q_{i+j}$ where $1 \leq i \leq n - L + 1$ and $0 \leq j \leq L - 1$, we need to find the Top-K matches for Q with a given relation R provided the match-score is $\geq \epsilon$.

The easiest way to share work across multiple subqueries is to cache data fetches. Tidlist fetches are cached because a token occurs in many sub-queries. We also cache records fetched during point queries because it is common for the same record to appear in the candidate set of overlapping segments. We call this algorithm that shares only data fetches but otherwise executes each Top-K independently our baseline **Iterative** algorithm.

The sharing of computation, in particular, the list merges is more challenging because the Top-K algorithm above crucially depends on token scores $V(t, Q)$ which change as Q changes. Therefore, the bounds of one query might make the results of merging a set of tidlists unusable for another.

We can maximize reuse of list merges if we ignore bounds and perform full merges as proposed in the progressive algorithm below.

3.3 Progressive Algorithm

Let $L_1 \dots L_n$ denote the n tidlists for the tokens in \mathbf{x} in the order in which the tokens appear in \mathbf{x} . When merging a contiguous set s of tidlists we generate another tidlist L_s in the same decreasing score order. Let F_i denote the frontier set, $F_i = \{L_s | s = \{j, j+1, \dots, i\} \forall j \in [i-L+1, i]\}$. The progressive algorithm fetches each L_i in sequence and merges it with each of the lists in the frontier set F_{i-1} to form a new frontier F_i as shown in Figure 1 for $n = 5, L = 4$ and $i = 4$. The algorithm stops after outputting the Top-K from frontier F_n .

The total number of list merges is $O(nL)$ which is much smaller than the $O(nL^2)$ merges performed when executing each Top-K independently. However, empirical results show that the progressive algorithms is significantly slower than the iterative algorithm primarily because of the various

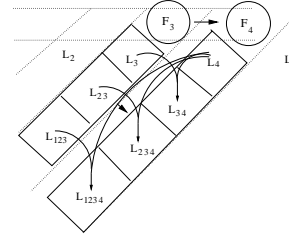


Figure 1. Moving Frontier

optimizations in the underlying Top-K, including pruning of lists based on upper bounds and the elimination of weak tidlists. Some of the tidlists may be arbitrary large, almost half the size of the given relation, in that case most of the time is spent in retrieving the tidlist from the database and merging these long lists. We next propose an algorithm that avoids these deficiencies.

3.4 Segmented Algorithm

Given a string \mathbf{x} , we use the criterion of simpleTop-k above to mark the tokens in a sub-query Q of \mathbf{x} as *weak* or *strong*. A token is globally strong if it is strong in any of the subqueries, otherwise it is globally weak. We use L_i to denote the tidlist of a token x_i if it is weak, otherwise we denote it by S_i . The strong and weak tokens might be arbitrarily interleaved in \mathbf{x} . Our strategy is to merge the lists for strong tokens completely for each sub-query $Q_{ji} = \{x_j, \dots, x_i\}$ progressively. Note that this differs from Progressive Algorithm 3.3, in that weak tokens are ignored while merging. Figure 2 shows an example of the complete merged lists for $n=7$ and $L=7$, where x_2, x_3, x_5 and x_6 are the strong tokens. As these are short tokens, a complete merge will not take much time compared to the time taken for partial merge in case of simpleTopK. But we gain since we can reuse the complete merge for other subqueries, where as we could not reuse the partial merged lists in Iterative algorithm. In this case, since we perform complete merges we scan the tidlists in rid order so as to allow for faster merges.

Then we iterate over each sub-query Q_{ji} . If all the tokens of Q are strong, we output top k records of the merged list $Z_{j,i}$, where $Z_{j,i}$ denotes the complete merger of all strong tokens from j to i . If all the tokens are weak, there will not be any result $\geq \epsilon$ for Q_{ji} . If Q_{ji} contains both weak and strong tokens, we find the complete merged list of all the strong tokens of Q_{ji} (Step 10 of Algorithm 2). Then we filter it using the best estimated contribution from weak tokens, to obtain candidate list and find the actual scores for records in the candidate list by point queries against the database. To make point queries efficient, we also cache

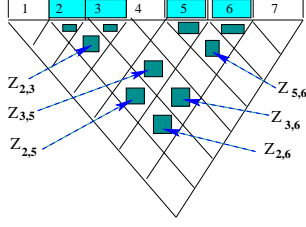


Figure 2. Complete Merged Lists in Segmented Batch TopK

the result of the point queries. Segmented Batch-Top-K is a efficient, yet simple to implement algorithm.

INPUT: $X = x_1 x_2 \dots x_n, k, \epsilon, L$

OUTPUT: Batch Top-K

- 1: Mark each x_i as global weak or global strong w.r.t. query X
- 2: $S =$ Set of all strong tokens
- 3: Moving from left, merge the lists of strong token progressively
- 4: $Z_{j,i}$ = Merged list of strong tokens of subquery $x_j \dots x_i$
- 5: **for** each pair (j,i) in $\{(u,v) : v - u < L, 1 \leq u \leq v \leq n\}$ **do**
- 6: **if** $\{x_j, \dots, x_i\} \subseteq S$ **then**
- 7: output top-k from $Z_{j,i}$
- 8: **else**
- 9: **if** NOT $(\{x_j \dots x_i\} \subseteq X - S)$ **then**
- 10: $\alpha = \min\{\gamma : x_\gamma \in S, j \leq \gamma \leq i\}$
- 11: $\beta = \max\{\gamma : x_\gamma \in S, j \leq \gamma \leq i\}$
- 12: $C_{\alpha,\beta}$ = candidate set filtered from $Z_{\alpha,\beta}$
- 13: Do point queries for each $r \in C_{\alpha,\beta}$ and output top-k
- 14: **end if**
- 15: **end if**
- 16: **end for**

Algorithm 2. Segmented Batch Top-k

4 Optimizing Viterbi with dictionary lookups

We show how to modify the Viterbi algorithm for finding the best segmentation so as to not require *exact* dictionary match scores for all possible input segments. Our goal is to exploit cheaper non-dictionary features and bounds on match scores to reduce the set of segments for which we need to get exact similarity.

Consider the Viterbi Equation 1. Suppose now the feature function $g(y, y', \mathbf{x}, i', i)$ instead of necessarily returning the exact feature value, returns a lower bound $g^l(y, y', \mathbf{x}, i', i)$ and an upper bound $g^u(y, y', \mathbf{x}, i', i)$ within which the actual value lies. For features that are cheap to compute the lower and upper bounds are the same. For the more expensive dictionary features, it is relatively cheap to get lower and upper bounds via aggregate token statistics or partial merges of the strong lists, but the exact

values require expensive point queries or complete tidlist merges.

We therefore start with bounds on the exact values and refine the bounds only when the existing tight features cannot uniquely find the best segmentation.

For ease of exposition we partition features into one of two types: *state features* that are independent of the previous label y' and *transition features* that depend on the transition features; and assume that the expensive dictionary features are all state features.

Given an input \mathbf{x} , for each of its segment $s = (t, u, y)$ with start boundary t , end boundary u ($u < t + L$) and label y , we maintain the following quantities:

- $\text{suffixUB}(u, y)$: an upper bound to the total score over all segmentation from $u + 1$ until the end position n with the previous label y . This is the maximum score we can obtain over all segmentation starting with $u + 1$ where the previous label is y .
- $\text{suffixLB}(u, y)$: a lower bound to the total score over all segmentation from $u + 1$ to n where the previous label is y . This is the score we are guaranteed to obtain for the best segmentation starting from $u + 1$ with previous label y .
- segmentUB , segmentLB : the upper bound and lower bound on the sum of scores of the state features of segment s .
- inexact-F : the set of features yet to be refined for segment s
- prefixCurrent : the current best score of a segmentation from the start position 1 and ending in segment s , excluding the scores of the state features of s .

There are two main phases of the algorithm. A backward Viterbi phase when we compute the suffixUB , suffixLB , segmentLB and segmentUB values for all segments. This is followed by a forward phase where we depend on a branch and bound algorithm to refine the match scores of only the minimum number of segments needed to find the optimal segmentation.

4.1 Backward Viterbi pass

In the reverse Viterbi pass we find the suffixUB and suffixLB values for each position i ($1 \leq i < n$) and label y . During this pass we also store for each segment, the upper bound segmentUB and lower bounds segmentLB on the state scores of each segment while also recording the features inexact-F which need refinement. For each segment $s = (t, u, y)$ the segmentUB and segmentLB values

are computed as follows:

$$\begin{aligned} \text{segmentUB}(t, u, y) &= \sum_{w_j \geq 0} w_j \cdot g_j^u(y, \mathbf{x}, t, u) \\ &\quad + \sum_{w_k < 0} w_k \cdot g_k^l(y, \mathbf{x}, t, u) \\ \text{segmentLB}(t, u, y) &= \sum_{w_j \geq 0} w_j \cdot g_j^l(y, \mathbf{x}, t, u) \\ &\quad + \sum_{w_k < 0} w_k \cdot g_k^u(y, \mathbf{x}, t, u) \end{aligned}$$

Now using the following backward Viterbi we compute the suffixUB and suffixLB values for each position i and label y . Assume \mathbf{g}' includes only transition features.

$$\begin{aligned} \text{suffixUB}(i, y) &= \max_{y', i < i' \leq i+L} \{ \text{segmentUB}(i+1, i', y') + \\ &\quad \mathbf{W} \cdot \mathbf{g}'(y', y, i+1, i') + \text{suffixUB}(i', y') \} \\ \text{suffixLB}(i, y) &= \max_{y', i < i' \leq i+L} \{ \text{segmentLB}(i+1, i', y') + \\ &\quad \mathbf{W} \cdot \mathbf{g}'(y', y, i+1, i') + \text{suffixLB}(i', y') \} \end{aligned}$$

An important issue is how to assign the initial lower bound g^l and upper bound g^u for the TF-IDF similarity features. There are several options: One option is to assign the trivial bounds of 0 and 1 respectively — this requires no work but would produce weak values of suffixUB and suffixLB causing the followup forward refinement to do several refinements. A second option is to compute the upper and lower bounds from the aggregate token statistics. This gives tighter bounds at negligible overheads since the token statistics are cached in memory. A third option is to perform partial tidlist fetch and merges, for example following the Segmented batch topk algorithm we perform all the strong segment merges and perform the point queries only when asked to refine to the exact score. As we perform increasing amount of work during the initialization, we run into the likelihood of most of the work being wasted and not useful for finding the best segmentation. We found the second option to provide the right tradeoff between the tightness of the initial bound and the time spent in its computation.

4.2 Forward selective refinement phase

We perform a branch and bound search strategy to order segments based on their promise of being part of the optimal segmentation and refine the inexact features in that order. For each segment s we maintain upper and lower bounds on the best possible path passing through the segment as: $\text{pathUB} = \text{suffixUB}(u, y) + \text{segmentUB}(s) + \text{prefixCurrent}(s)$ and $\text{pathLB} = \text{suffixLB}(u, y) + \text{segmentLB}(s) + \text{prefixCurrent}(s)$. Initially, prefixCurrent is negative infinity for all but the set of segments starting

at 1 for which it is 0. We use a priority queue to maintain the segments sorted on their pathUB values. A lower bound on the best possible segmentation is the maximum of the pathLB values over all segments. Let bestLB denote this lower bound. This is the least score that the best segmentation is guaranteed to have. All segments with pathUB less than bestLB can be removed from the priority queue. The algorithm proceeds by dequeuing the segment s with the highest pathUB value. If $s.\text{inexact-F}$ is not empty, we refine the features in inexact-F exactly and remove them from inexact-F . We adjust segmentUB and segmentLB values of s by the refined amount and propagate the change to pathLB and pathUB and bestLB values. If s is an ending segment, that is, $s.u = n$, then ensure s is fully refined ($\text{segmentLB} = \text{segmentUB}$ for s) and put it back to queue, after setting $s.u = n + 1$. Note that $s.u = n + 1$ is an indicator that the path through this segment has exact score and if it is dequeued, then the path traced from s backwards gives the best possible segmentation. Otherwise, for all segments s' starting after s (i.e., with $s'.t = s.u + 1$), if $s'.\text{prefixCurrent}$ is less than $s.\text{prefixCurrent} + s.\text{segmentLB} + \mathbf{W} \cdot \mathbf{g}'(s'.y, s.y, s'.t, s'.u)$, we update the prefixCurrent value of s' , make s the previous segment of s' , and add s' to the priority queue with updated weights provided $\text{suffixUB}(s')$ is greater than bestLB . (The \mathbf{g}' vector only includes the transition features and the third term can also be cached during the backward Viterbi pass.)

The algorithm is guaranteed to return the optimal solution when it terminates (the proof is similar to standard proofs used in Top-K [7] and A-star [8] search algorithms) but with fewer feature refinements than required by the forward Viterbi pass. The storage requirement of this algorithm is $O(nLY)$ — this is comparable to the requirements of normal Viterbi. The priority queue is also limited to be no more than $O(nLY)$ size since for each segment we have only a single entry in the queue. The time taken in the backward pass $O(nLY^2)$ is the same as for normal forward Viterbi except that the expensive dictionary features are replaced by fast bounds. During the forward pass, the main cost is the feature refinements which in the worst case can be the same as in normal Viterbi but in most practical cases significantly less as we will see in the experimental section.

5 Experimental results

In this section we demonstrate the efficiency of our BatchTopK algorithms and show the time improvement in entity recognition with the optimized Viterbi algorithm. We report experiments on two unstructured data sources.

Cora: Cora is a popular citations benchmark [13] that consists of references collected from the reference section of several academic papers. This dataset consists of 500 entries.

Articles: This dataset consists of 100 journal articles collected by one of our authors from Cite-seer and therefore formatted slightly differently from the Cora dataset. Also, unlike Cora it consists only of journal entries. The dataset is available at <http://www.it.iitb.ac.in/~sunita/data/personalBib.tar.gz>.

In both cases our goal was to extract title, author names and journal names from the unstructured text. The reference database for defining similarity features was DBLP consisting of 360,000 titles and 280,000 authors. The average length in DBLP of title and author field was 8 and 3 respectively. As in most real life datasets, we observe that tokens exhibit a zipfian distribution. There are many stop tokens whose tidlist sizes range in tens of thousands, that are spread out in the tail of the distribution. The $\max_t |\text{tidlist}(t)|$ was 108284 and 102259 for DBLP title and author databases respectively.

All experiments were run with default values of $\epsilon = 0.5$, top $k = 10$ and full database size. We also show graphs with changing values of these parameters. The maximum segment size was 20 for the Title dictionary match and 6 for the authors dictionary match. The average query length for articles dataset was 34 and 28 for cora dataset.

The database and inverted index was stored in Postgres version 7.4.2. The Batch Top K and Viterbi algorithms were implemented in Java and interacted with Postgres via JDBC. All our experiments were performed on an IBM X220 server with 2 GB of main memory and Intel PIII processor rated at 1.2 GHz.

5.1 BatchTopK experimental results

We first study the performance of the following batch Top-K algorithms. We compare our proposed Segmented Batch-Top-K with the Iterative Batch-Top-K algorithm that issues independent Top-K searches for each segment but caches tidlist fetches and results of point queries. We also compare the Batch-Top-K algorithm with a lower bound on the most optimized Batch-Top-K algorithm possible given a simple Top-K algorithm. For computing the lower bound we partition the query string into non-overlapping segments, and sum up the time taken for the most optimized simple Top-K for each segment. This is a lower bound since we are solving a subset of all possible simple Top K searches of Batch-Top-K and there is no sharing possible amongst disjoint segments. We choose a partitioning that produces a tight lower bound by picking the most expensive non-overlapping segments (Algorithm 3).

In Figure 3 we report running times for the above three methods on a set of 100 queries selected at random from the unstructured records of the dataset. For queries of article dataset, we perform the experiment on title and author fields, with corresponding full size dictionaries (360K titles,

```

LowerBound( $x_1, x_n$ )
for every sub segment  $x_i, \dots, x_j$  required by Batch Top-K do
    call simple Top-K on  $x_i, \dots, x_j$  using the best implementation of simple Top-K and measure time taken
end for
let  $x_a \dots x_b$  be the segment, that took maximum time, T
return T + LowerBound( $x_1, x_{a-1}$ ) + LowerBound( $x_{b+1}, x_n$ )

```

Algorithm 3. Lower bound on Batch-Top-K

and 280K author names) and for Cora only report numbers with the title field. Though there is no natural ordering in the set of queries, we sort them in increasing order of the time taken by the Iterative method. We observe that for some of the long running queries, the Iterative algorithm can be almost a factor of ten worse than the Segmented Batch-Top-K algorithm. When averaged over the 100 queries, on both datasets we found Segmented to take one-third the time of Iterative for Titles and half the time of Iterative for Authors. Also, Segmented takes no more than twice the time of the lower bound, that includes significantly fewer segments.

The performance improvement in Segmented over Iterative is due to savings in repeated tidlist merges. We observe that this time improvement is greater for queries over titles, than authors. While querying over authors database we have very few strong tokens, as compared to title database. This is because of the shorter length of the author field (average=3) and a shorter collection of tokens that could find a good match over author database. In contrast, titles fields are longer (average=8) and there are far more tokens that match with the titles database. Therefore, Batch-TopK over authors yields fewer records and have lesser running times. Fewer strong tokens over authors database implies fewer tidlist merges and thereby reduced gain of Segmented over Iterative.

We have not included numbers for the Progressive algorithm (Section 3.3) because it takes more than a factor of ten times the running time of Iterative. This is mainly due to uniform treatment of all tokens. Stop-words (weak tokens) have very lengthy tidlists which take longer time for fetch and merges.

5.1.1 Effect of database size and epsilon

In Figure 4(a) we plot the running times of various Batch-Top-K algorithms for different dictionary sizes on the Articles dataset with epsilon fixed at 0.5. The time reported is averaged over 100 queries. As expected the relative improvement over Iterative increases as database sizes increases. In Figure 4(b) we report results with varying values of ϵ for the full database size. Top-K algorithms are sensitive to epsilon and running times increase rapidly for lower values of epsilon. The number of strong tokens in a query increases as epsilon decreases. This leads to rapid increase

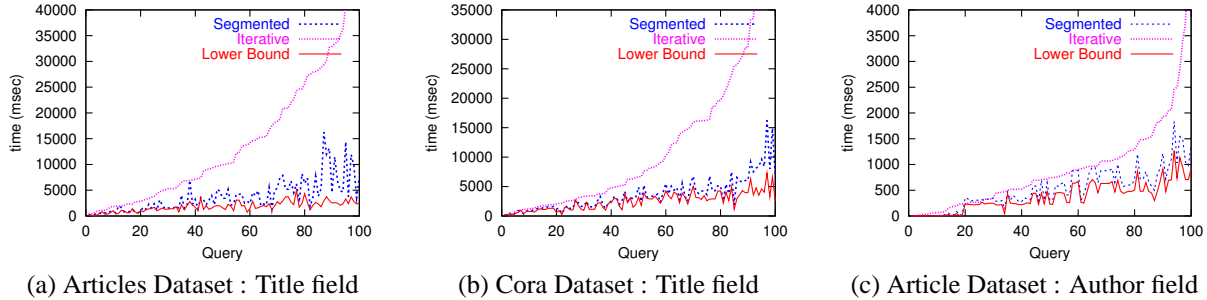
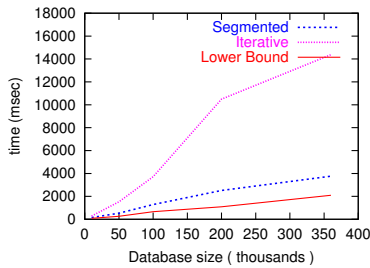
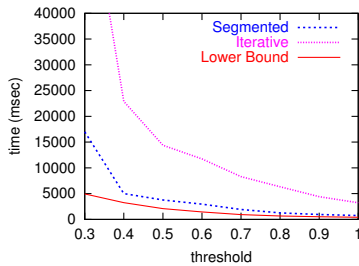


Figure 3. Running times (in milliseconds) of different Batch Top-K algorithms



(a) DB size variation



(b) Epsilon variation

Figure 4. Effect of varying DB size and epsilon on running times (in milliseconds) of different Batch Top-K algorithms

in running times of Iterative as the total number of tidlist merges increase. With Segmented however, the re-usage of tidlist merges, assauges the increase in the running times.

ϵ	0.3	0.5	0.7	0.9
accuracy	76.0	76.3	70.12	69.46

Table 1. Effect of epsilon on accuracy of the model

5.2 Inferring algorithms

In this section we show the running times of our optimized inference algorithm (Optimized-Viterbi) and compare with the normal Viterbi algorithm running with the Segmented Batch-Top-K algorithm and Iterative Batch-Top-K algorithm. Note that the accuracy of entity recognition is unaffected by the different inferring techniques, as they all return the optimum segmentation.

We trained two models of NER, one on 150 examples from the cora dataset and the other on 50 examples from the articles dataset. The training process assigned weights to the various features including the dictionary match features for the title and author columns. We fixed the value of epsilon to be 0.5 for all training and inferring processes. This value was observed to provide sufficiently high accuracy for reasonable running times.

We show an example of the variation of accuracy with epsilon for the articles dataset on the authors column of DBLP in Table 1. We set ϵ to different values, re-train our model, and measure accuracy. We observe that accuracy drops for higher values of ϵ . For lower values of ϵ , the training and inferring times are very high. We choose a value of 0.5 as the right trade off for accuracy and running time. The $\epsilon = 1$ case corresponds to not using the database. This illustrates the importance of external databases which boost accuracy from 69 to 76.

Overall comparisons In Table 2 we present an overall comparison of the three methods for Viterbi-based inferring and for reference also show the running time of Viterbi without any dictionary-based similarity feature. We observe that compared to Viterbi without any dictionary similarity features, Viterbi with a simple Iterative TopK algorithm could take upto a factor of twenty higher running time. This is reduced by more than a factor of half with the Segmented Batch-Top-K algorithm. The improvement for Cora-author is smaller because in this case even the Iterative algorithm took only 20% more time. Also optimized Viterbi for Authors performed worse because the overhead of the two pass

DataSet	Dictionary	Average Inference Time (ms)			
		Viterbi with			Optimized Viterbi
		no dictionary	Iterative BTK	Segmented BTK	
Articles	Title	715	16398	8609	2230
Cora	Title	559	12520	5090	1582
Articles	Author	640	1900	880	1192
Cora	Author	404	564	467	763

Table 2. Overall result of the Inferencing Algorithms

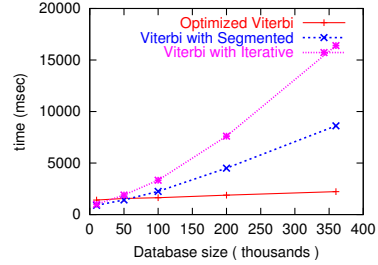
Viterbi is more than the savings due to partial segment refinement. The reason is because the Cora dataset mostly includes machine learning papers and these have little overlap with the DBLP database. Optimized Viterbi for Titles reduced running time by more than a factor of three over Segmented Batch-Top-K.

Changing database size We wish to study the effect of costly features on the running times of our inferencing algorithms. To vary the cost of our distance feature, we simply choose dictionaries of larger size, as this directly increases the cost of evaluating the distance feature. We observe in Figure 5 that for smaller dictionary sizes Optimized-Viterbi is slightly slower than Viterbi with Segmented Batch-Top-K. This is because, for small dictionaries, the optimized Batch-Top-K algorithms are fast, and the savings in dictionary lookups due to Optimized-Viterbi are offset by the two stages of Viterbi that it requires. But as the dictionary size grows, the savings in dictionary lookups start making a significant impact on running time and for the largest database size there is a factor of four reduction even over the optimized Segmented Batch-Top-K algorithm and a factor of eight reduction over the independent Batch-Top-K algorithm.

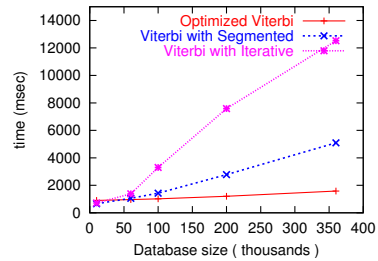
The superior performance of the Optimized Viterbi algorithm does not make the Segmented Batch-Top-K algorithm redundant because during training we do not just require just the best segmentation but need to sum over the scores of all possible segmentation. In summary, these experiments show that when the database size is large, both the Segmented Batch-Top-K and the Optimized Viterbi algorithm proposed in this paper can significantly improve the performance of entity recognition systems.

6 Related work

The work presented here is most related to the Top-K search problem, for which a number of algorithms have been proposed in the database and IR literature [7, 5, 19, 20, 16] and these are already discussed in Section 3.1. We build upon this work to design our segmented Batch-Top-K algorithm. We know of no earlier work on the specific



(a) Articles Dataset



(b) Cora Dataset

Figure 5. Comparison of average inferencing time of Viterbi and Optimized Viterbi

Batch-Top-K problem that we solve in this paper.

Our ideas for extending Viterbi to a A-star like search algorithm for reducing costs of expensive features is related to the factored A-star algorithm of [9]. However, the setting there is different in that features are decomposed into disjoint sets and the suffix upper bound is computed for each set separately and summed up to get the total upper bound. Thus, there may not be any single path that corresponds to the upper bound. In our case, we have bounds associated with the values of features and the algorithm attempts to minimize expensive refinements of these bounds.

While there is prior work on the role of dictionaries to improve extraction accuracy [12, 2, 1, 6], we are not aware of any work addressing the performance issues associated with exploiting large databases.

7 Conclusion

A number of applications now depend on statistical models for the automatic extraction of structured entities from unstructured text. These models combine clues from a number of different sources including match with existing databases of structured entities. As the size of the database increase, the computation of match scores with successive segments of the unstructured text, becomes a bottleneck in the extraction process. In this paper we proposed two ways to address this performance problem. We first formulated a Batch-Top-K problem to batch up the computation of similarity scores for overlapping segments and developed an algorithm that is a factor of two to three faster than independent Top-K searches with data caching. Our algorithm combines the optimization tricks from earlier Top-K search algorithms such that it is at most a factor of two worse than an unachievable lower bound. Next, we modified the standard Viterbi algorithm used to find the best entity segments so as to not require exact similarity computation for all segments. This further reduces the running time by a factor of three for large databases.

References

- [1] E. Agichtein and V. Ganti. Mining reference tables for automatic text segmentation. In *Proc. of ACM SIGKDD, Seattle, USA*, 2004.
- [2] V. R. Borkar, K. Deshmukh, and S. Sarawagi. Automatic text segmentation for extracting structured records. In *Proc. of ACM SIGMOD*, Santa Barbara, USA, 2001.
- [3] A. Borthwick, J. Sterling, E. Agichtein, and R. Grishman. Exploiting diverse knowledge sources via maximum entropy in named entity recognition. In *Sixth Workshop on Very Large Corpora New Brunswick, New Jersey. Association for Computational Linguistics.*, 1998.
- [4] M. Califf and R. Mooney. Relational learning of pattern-match rules for information extraction. *Working Papers of the ACL-97 Workshop in Natural Language Learning*, pages 9–15, 1997.
- [5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of ACM SIGMOD*, 2003.
- [6] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: Combining semi-markov extraction processes and data integration methods. In *Proc. of ACM SIGKDD, Seattle, USA*, 2004.
- [7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66:614,656, Sept. 2001.
- [8] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
- [9] D. Klein and C. D. Manning. Factored a* search for models over sequences and trees. In *Proc. of International Joint Conference on Artificial Intelligence*, 2003.
- [10] J. Kupiec. Robust part of speech tagging using a hidden Markov model. *Computer Speech and Language*, 6:225–242, 1992.
- [11] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of ICML*, Williams, MA, 2001.
- [12] I. Mansuri and S. Sarawagi. A system for integrating unstructured data into relational databases. To appear in ICDE, 2006.
- [13] F. Peng and A. McCallum. Accurate information extraction from research papers using conditional random fields. In *HLT-NAACL*, pages 329–336, 2004.
- [14] A. Ratnaparkhi. Learning to parse natural language with maximum entropy models. *Machine Learning*, 34, 1999.
- [15] S. Sarawagi and W. W. Cohen. Semi-markov conditional random fields for information extraction. In *NIPs*, 2004.
- [16] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proc. of ACM SIGMOD*, 2004.
- [17] K. Seymore, A. McCallum, and R. Rosenfeld. Learning Hidden Markov Model structure for information extraction. In *Papers from the AAAI-99 Workshop on Machine Learning for Information Extraction*, pages 37–42, 1999.
- [18] F. Sha and F. Pereira. Shallow parsing with conditional random fields. In *Proc. of HLT-NAACL*, 2003.
- [19] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proc. of VLDB*, pages 648–659, 2004.
- [20] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, 1999.