Entity Annotation based on Inverse Index Operations

Ganesh Ramakrishnan, Sreeram Balakrishnan, Sachindra Joshi
IBM India Research Labs
IIT Delhi, Hauz Khas,
New Delhi, India
{ganramkr, sreevb, jsachind}@in.ibm.com

Abstract

Entity annotation involves attaching a label such as 'name' or 'organization' to a sequence of tokens in a document. All the current rule-based and machine learning-based approaches for this task operate at the document level. We present a new and generic approach to entity annotation which uses the inverse index typically created for rapid key-word based searching of a document collection. We define a set of operations on the inverse index that allows us to create annotations defined by cascading regular expressions. The entity annotations for an entire document corpus can be created purely of the index with no need to access the original documents. Experiments on two publicly available data sets show very significant performance improvements over the document-based annotators.

1 Introduction

Entity Annotation associates a well-defined label such as 'person name', 'organization', 'place', *etc*., with a sequence of tokens in unstructured text. The dominant paradigm for annotating a document collection is to annotate each document separately. The computational complexity of annotating the collection in this paradigm, depends linearly on the number of documents and the cost of annotating each document. More precisely, it depends on the total number of tokens in the document collection. It is not uncommon to have millions of documents in a collection. Using this paradigm, it can take hours or days to annotate such big collections even with highly parallel server farms. Another drawback of this paradigm is that the entire document collection needs to be reprocessed whenever new annotations are required.

In this paper, we propose an alternative paradigm for entity annotation. We build an index for the tokens in the document collection first. Using a set of operators on the index, we can generate new index entries for sequences of tokens that match any given regular expression. Since a large class of annotators (*e.g.*, GATE (Cunningham et al., 2002)) can be built using cascading regular expressions, this approach allows us to support annotation of the document collection purely from the index.

We show both theoretically and experimentally that this approach can lead to substantial reductions in computational complexity, since the order of computation is dependent on the size of the indexes and not the number of tokens in the document collection. In most cases, the index sizes used for computing the annotations will be a small fraction of the total number of tokens.

In (Cho and Rajagopalan, 2002) the authors develop a method for speeding up the evaluation of a regular expression 'R' on a large text corpus by use of an optimally constructed multi-gram index to filter documents that will match 'R'. Unfortunately, their method requires access to the document collection for the final match of 'R' to the filtered document set, which can be very time consuming. The other bodies of related prior work concern indexing annotated data (Cooper et al., 2001; Li and Moon, 2001) and methods for document level annotation (Agichtein and Gravano, 2000; McCallum et al., 2000). The work on indexing annotated data is not directly relevant, since our method creates the index to the annotations directly as part of the algorithm for computing the annotation. (Eikvil, 1999) has a good survey of existing document level IE methods. The relevance to our work is that only a certain class of annotators can be implemented using our method: namely anything that can be implemented using cascading weighted regular expressions. Fortunately, this is still powerful enough to enable a large class of highly effective entity annotators.

The rest of the paper is organized as follows. In Section 2, we present an overview of the proposed approach for entity annotation. In Section 3, we construct an algorithm for implementing a deterministic finite automaton (DFA) using an inverse index of a document collection. We also compare the complexity of this approach against the direct approach of running the DFA over the document collection, and show that under typical conditions, the index-based approach will be an order of magnitude faster. In Section 4, we develop an alternative algorithm which is based on translating the original regular expression directly into an ordered AND/OR graph with an associated set of index level operators. This has the advantage of operating directly on the much more compact regular expressions instead of the equivalent DFA (which can become very large as a result of the NFA to DFA conversion and epsilon removal steps). We provide details of our experiments on two publicly available data sets in Section 5. Finally we present our conclusions in Section 6.

2 Overview

Figure 1 shows the process for entity annotation presented in the paper. A given document collection $\mathcal{D}$ is tokenized and segmented into sentences. The tokens are stored in an inverse index I. The inverse index I has an ordered list $\mathcal{U}$ of the unique tokens u_1, u_2, $..u_W$ that occur in the collection, where W is the number of tokens in I. Additionally, for each unique token u_i, I has a postings list $L(u_i) =< l_1, l_2, \ldots l_{cnt(u_i)} >$ of locations in $\mathcal{D}$ at which u_i occurs. $cnt(u_i)$ is the length of $L(u_i)$. Each entry l_k, in the postings list $L(u_i)$, has three fields: (1) a sentence identifier, $l_k.sid$, (2) the begin position of the particular occurrence of u_i, $l_k.first$ and (3) the end position of the same occurrence of u_i, $l_k.last$.

We require the input grammar to be the same as that used for named entity annotations in GATE (Cunningham et al., 2002). The GATE architecture for text engineering uses the Java Annotations Pattern Engine (JAPE) (Cunningham, 1999) for its information extraction task. JAPE is a pattern matching language. We support two classes of properties for tokens that are required by grammars such as JAPE: (1) orthographic properties such as an uppercase character followed by lower case characters, and (2) gazetteer (dictionary) containment properties of tokens and token sequences such as ‘location’ and ‘person name’. The set of tokens along with entity types specified by either of these two properties are referred to as *Basic Entities*. The instances of basic entities specified by orthographic properties must be single tokens. However, instances of basic entities specified using gazetteer containment properties can be token sequences.

The module (1) of our system shown in Figure 1, identifies postings lists for each basic entity type. These postings lists are entered as index entries in I for the corresponding types. For example, if the input rules require tokens/token sequences that satisfy *Capsword* or *Location Dictionary* properties, a postings list is created for each of these basic types. Constructing the postings list for a basic entity type with some orthographic property is a fairly straightforward task; the postings lists of tokens satisfying the orthographic properties are merged (while retaining the sorted order of each postings list). The mechanism for generating the postings list of basic entities with gazetteer properties will be developed in the following sections. A rule for NE annotation may require a token to satisfy multiple properties such as *Location Dictionary* as well as *Capsword*. The posting list for tokens that satisfy multiple properties are determined by performing an operation $parallelint(L, L')$ over the posting lists of the corresponding basic entities. The $parallelint(L, L')$ operation returns a posting list such that each entry in the returned list occurs in both L as well as L'. The module (2) of our system shown in Figure 1 identifies instances of each annotation type, by performing index-based operations on the postings lists of *basic entity* types and other tokens.

3 Annotation using Cascading Regular Expressions

Regular expressions over basic entities have been extensively used for NE annotations. The Common Pattern Specification Language (CSPL)[1] specifies a standard for describing Annotators that can be implemented by a series of cascading regular expression matches.

Consider a regular expression R over an alphabet Σ of basic entities, and a token sequence

[1] `http://www.ai.sri.com/~appelt/TextPro`

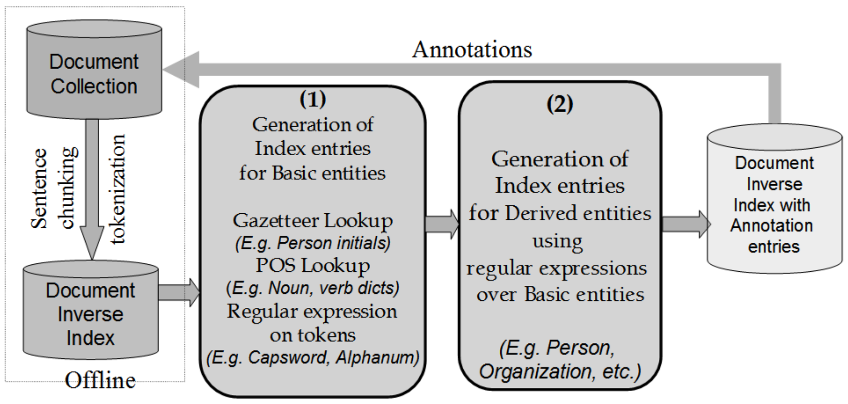

Annotations
Document Collection
Sentence chunking
tokenization
Document Inverse Index
Offline
(1)
Generation of Index entries for Basic entities
Gazetteer Lookup
(E.g. Person initials)
POS Lookup
(E.g. Noun, verb dicts)
Regular expression on tokens
(E.g. Capsword, Alphanum)
(2)
Generation of Index entries for Derived entities using regular expressions over Basic entities
(E.g. Person, Organization, etc.)
Document Inverse Index with Annotation entries

Figure 1: Overview of the entity annotation process described in this paper

$T = \{t_1, \ldots, t_W\}$. The annotation problem aims at determining all matches of regular expression R in the token sequence T. Additionally, NE annotations do not span multiple sentences. We will therefore assume that the length of any annotated token sequence is bounded by Δ, where Δ can be the maximum sentence length in the document collection of interest. In practice, Δ can be even smaller.

3.1 Computing Annotations using a DFA

Given a regular expression R, we can convert it into a deterministic finite automate (DFA) D_R. A DFA is a finite state machine, where for each pair of state and input symbol, there is one and only one transition to a next state. D_R starts processing of an input sequence from a start state s_R, and for each input symbol, it makes a transition to a state given by a transition function Φ_R. Whenever D_R lands in an accept state, the symbol sequence till that point is accepted by D_R. For simplicity of the document and index algorithms, we will ignore document and sentence boundaries in the following analysis.

Let $@t_{i,i+\Delta}, 1 \leq i \leq W - \Delta$ be a subsequence of T of length Δ. On a given input $@t_{i,i+\Delta}$, D_R will determine all token sequences originating at t_i that are accepted by the regular expression grammar specified through D_R. Figure 2 outlines the algorithm *findAnnotations* that locates all token sequences in T that are accepted by D_R.

Let D_R have $\{S_1, \ldots, S_N\}$ states. We assume that the states have been topologically ordered so that S_1 is the start state. Let α be the time taken to consume a single token and advance the DFA to the next state (this is typically implemented as a table or hash look-up). The time taken by the algorithm *findAnnotations* can be obtained by summing up the number of times each state is visited as the input tokens are consumed. Clearly, the state S_1 is visited W times, W being the total number of symbols in the token sequence T. Let $cnt(S_i)$ give the total number of times the state S_i has been visited. The complexity of this method is:

$$C_D = \alpha \sum_{i=1}^{i=N} cnt(S_i) = \alpha \left[W + \sum_{i=2}^{i=N} cnt(S_i) \right] \quad (1)$$

```
**findAnnotations**($T, D_R$)
Let $T = \{t_1, \ldots, t_W\}$
**for** $i$ = 1 to $W - \Delta$ **do**
    let $@t_{i,i+\Delta}$ be a subsequence of length $\Delta$ starting from $t_i$ in $T$
    use $D_R$ to annotate $@t_{i,i+\Delta}$
**end for**
```

Figure 2: The algorithm for finding all the occurrences of R in a token sequence T.

3.2 Computing Regular Expression Matches using Index

In this section, we present a new approach for finding all matches of a regular expression R in a token sequence T, based on the inverse index I of T. The structure of the inverse index was presented in Section 2. We define two operations on postings lists which find use in our annotation algorithm.

1. $merge(L, L')$: Returns a postings list such that each entry in the returned list occurs either in L or L' or both. This operation takes $O(|L| + |L'|)$ time.

2. $consint(L, L')$: Returns a postings list such that each entry in the returned list points to a token sequence which consists of two consecutive

subsequences $@sa$ and $@sb$ within the same sentence, such that, L has an entry for $@sa$ and L' has an entry for $@sb$. There are several methods for computing this depending on the relative size of L and L'. If they are roughly equal in size, a simple linear pass through L and L', analogous to a merge, can be performed. If there is a significant difference in sizes, a more efficient modified binary search algorithm can be implemented. The details are shown in Figure 3. The

```
consint($L, L'$)
Let $M$ elements of L be $l_1 \cdots l_M$
Let $N$ elements of L' be $l'_1 \cdots l_N$
if $M < N$ then
  set $j = 1$
  for $i = 1$ to $M$ do
    set $k = 1$, keep doubling $k$ until
    $l'_j.first \leq l_i.last < l'_{j+k}.first$
    binary search the $L'$ in the interval $j \cdots k$
    to determine the value of $p$ such that
    $l'_p.first \leq l_i.last < l'_{p+1}.first$
    if $l'_p.first = l_i.last$ a match exists, copy to output
    set $j = p + 1$
  end for
else
  Same as above except $l$ and $l'$ are reversed
end if
```

Figure 3: The modified binary search algorithm for consint

complexity of this algorithm is determined by the size q_i of the interval required to satisfy $l'_j.first \leq l_i.last < l'_{j+q_i}.first$ (assuming $|L| < |L'|$). It will take an average of $\log_2(q_i)$ operations to determine the size of interval and $\log_2(q_i)$ operations to perform the binary search, giving a total of $2\log_2(q_i)$. Let $q_1 \cdots q_M$ be the sequence of intervals. Since the intervals will be at most two times larger than the actual interval between the nearest matches in L' to L, we can see that $|L'| \leq \sum_{i=1}^{M} q_i \leq 2 * |L'|$. Hence the worst case will be reached when $q_i = 2|L'|/|L|$ with a time complexity given by $2|L|\left(\log_2(|L'|/|L|) + 1\right)$, assuming $|L| < |L'|$.

To support annotation of a token sequence that matches a regular expression only in the context of some regular expression match on its left and/or right, we implement simple extensions to the $consint(L_1, L_2)$ operator. Details of the extensions are left out from this paper owing to space constraints.

3.3 Implementing a DFA using the Inverse Index

In this section, we present a method that takes a DFA D_R and an inverse index I of a token sequence T, to compute a postings list of subsequences of length at most Δ, that match the regular expression R.

Let the set $S = \{S_1, \ldots, S_N\}$ denote the set of states in D_R, and let the states be topologically ordered with S_1 as the start state. We associate an object $list_{s,k}$ with each state $s \in S$ and $\forall 1 \leq k \leq \Delta$. The object $list_{s,k}$ is a posting list of all token sequences of length exactly k that end in state s. The $list_{s,k}$ is initialized to be empty for all states and lengths. We iteratively compute $list_{s,k}$ for all the states using the algorithm given in Figure 4. The function $dest(S_i)$ returns a set of states, such that for each $s \in dest(S_i)$, there is an arc from state S_i to state s. The function $label(S_i, S_j)$ returns the token associated with the edge (S_i, S_j).

```
for $k$ = 1 to $\Delta$ do
  for $i$ = 1 to $N$ do
    for $s \in dest(S_i)$ do
      if $i == 1$ then
        $t = L(label(S_i, s))$
      else
        $t = consint(list_{S_i,k-1}, L(label(S_i, s)))$
      end if
      $list_{s,k} = merge(list_{s,k}, t)$
    end for
  end for
end for
```

Figure 4: The algorithm for building the index to all token sequences in T that match R.

At the end of the algorithm, all token sequences corresponding to postings lists $list_{s,i}, s \in S, 1 \leq i \leq \Delta$ are sequences that are matched by the regular expression R.

3.4 Complexity Analysis for the Index-based Approach

The complexity analysis of the algorithm given in Figure 4 is based on the observation that, $\sum_{k=1}^{k=\Delta} |list_{S_i,k}| = cnt(S_i)$. This holds, since $list_{S_i,k}$ contains an entry for all sequences that visit the state S_i and are of length exactly k. Summing the length of these lists for a particular state S_i across all the values of k will yield the total number of sequences of length at most Δ that visit the state S_i.

For the algorithm in Figure 3, the time taken by one $consint$ operation is given by $2\beta(|list_{S_i,k}| * (\log(\rho_{ijk}) + 1))$ where β is a constant that varies with the lower level implementation. $\rho_{ijk} = \frac{|L(label(S_i,S_j))|}{|list_{S_i,k}|}$ is the ratio of the postings list size of the label associated with the arc from S_i to S_j to the list size of S_i at step k. Note that $\rho_{ijk} \geq 1$. Let $prev(S_i)$ be the list of predecessor states to S_i. The time taken by all the $merge$ operations for a state S_i at step k is given by $\gamma(\log(|prev(S_i)|)|list_{S_i,k}|)$ Assuming all the merges are performed simultaneously, $\gamma(log(|prev(S_i)|)$ is the time taken to create each entry in the final merged list, where γ is a constant that varies with the lower level implementation. Note this scales as the log of the number of lists that are being merged.

The total time taken by the algorithm given in Figure 4 can be computed using the time spent on $merge$ and $consint$ operations for all states and all lengths. Setting $\bar{\rho_{is}} = \max_k \rho_{isk}$, the total time C_I can be given as:

$$C_I = \sum_{i=2}^{i=N} \left[\gamma \log(|prev(S_i)|) + 2\beta \sum_{s \in dest(S_i)} \log(\bar{\rho_{is}}) \right] cnt(S_i) \tag{2}$$

Note that in deriving Equation 2, we have ignored the cost of merging $list(S_a, k)$ for $k = 1 \cdots \Delta$ for the accept states.

3.5 Comparison of Complexities

To simplify further analysis, we can replace $cnt(S_i)$ with $fcnt(S_i)$ where $fcnt(S_i) = cnt(S_i)/W$. If we assume that the token distribution statistics of the document collection remain constant as the number of documents increases, we can also assume that $fcnt(S_i)$ is invariant to W. Since ρ_{ijk} is given by a ratio of list sizes, we can also consider it to be invariant to W. We now assume $\alpha \approx \beta \approx \gamma$ since these are implementation specific times for similar low level compute operations. With this assumptions from Equations 1 and 2, the ratio C_D/C_I can be approximated by:

$$\frac{1 + \sum_{i=2}^{N} fcnt(S_i)}{\sum_{i=2}^{N} \left[\sum_{s \in dest(S_i)} 2\log(\bar{\rho_{is}}) + \log(|prev(S_i)|) \right] fcnt(S_i)} \tag{3}$$

The overall ratio of C_D to C_I is invariant to W and depends on two key factors $fcnt(S_i)$ and $\sum_{s \in dest(S_i)} \log(\bar{\rho_{is}})$. If $fcnt(S_i) \ll 1$, the ratio will be large and the index-based approach will be much faster. However, if either $fcnt(S_i)$ starts approaching 1 or $\sum_{s \in dest(S_i)} \log(\bar{\rho_{is}})$ starts getting very large (caused by a large fan out from S_i), the direct match using the DFA may be more efficient.

Intuitively, this makes sense since the main benefit of the index is to eliminate unnecessary hash lookups for tokens do not match the arcs of the DFA. As $fcnt(S_i)$ approaches 1, this assumption breaks down and hence the inherent efficiency of the direct DFA approach, where only a single hash lookup is required per state regardless of the number of destination states, becomes the dominant factor.

3.6 Comparison of Complexities for Simple Dictionary DFA

To illustrate the potential gains from the index-based annotation, consider a simple DFA D_R with two states S_1 and S_2. Let the set of unique tokens A be $\{a, b, c \cdots z\}$. Let E be the dictionary $\{a, e, i, o, u\}$. Let D_R have five arcs from S_1 to S_2 one for each element in E. The DFA D_R is a simple acceptor for the dictionary E, and if run over a token sequence T drawn from A, it will match any single token that is in E. For this simple case $fcnt(S_2)$ is just the fraction of tokens that occur in E and hence by definition $fcnt(S_2) \leq 1$. Substituting into 3 we get

$$\frac{C_D}{C_I} = \frac{1 + fcnt(S_2)}{2\log(5) fcnt(S_2)} \tag{4}$$

As long as $fcnt(S_2) < 0.27$, this ratio will always be greater than 1.

4 Inverse Index-based Annotation using Regular Expressions

A DFA corresponding to a given regular expression can be used for annotation, using the inverse index approach as described in Section 3.3. However, the NFA to DFA conversion step may result in a DFA with a very large number of states. We develop an alternative algorithm that translates the original regular expression directly into an ordered AND/OR graph. Associated with each node in the graph is a regular expression and a postings list that points to all the matches for the node's regular expression in the document collection. There are two node types: AND nodes where the output list is computed from the $consint$ of the postings lists of two children nodes and OR nodes where the output list is formed by merging the posting lists of all the children nodes. Additionally, each node has two binary properties: isOpt and selfLoop. The first property is set if the regular expression being matched is of the form ‘R?’, where ‘?’ denotes that the regular expression R is optional. The second property is set if the regular expression is of the form ‘R+’, where ‘+’ is the Kleen operator denoting one or more occurrences. For the case of ‘R*’, both properties are set.

The AND/OR graph is recursively built by scanning the regular expression from left to right and identifying every sub-regular expression for which a sub-graph can be built. We use capital letters R, X to denote regular expressions and small letters a, b, c, *etc.*, to denote terminal symbols in the symbol set Σ. Figure 5 details the algorithm used to build the AND/OR graph. Effectively, the AND/OR graph decomposes the computation of the postings list for R into a ordered set of $merge$ and $consint$ operations, such that the output $L(v)$ for node v become the input to its parents. The graph specifies the ordering, and by evaluating all the nodes in dependency order, the root node will end up with a postings list that corresponds to the desired regular expression.

```
if $R$ is empty then
  Return $NULL$
else if $R$ is a symbol $a \in \Sigma$ then
  Return $createNode(\text{name} = a)$
else
  Decompose $R$ such that $R \to R'$ $<regexp>$
  if $<regexp>$ is empty then
    if $R' == (X)$ or $X+$ or $X*$ or $X?$ then
      $node = createGraph(X)$
      if $R' == X+$ or $X*$ then
        $node.selfLoop = 1$
      end if
      if $R' == X?$ or $X*$ then
        $node.isOpt = 1$
      end if
    else if $R' == (X_1|X2|..|X_k)$ then
      $node = createNode(\text{name} = R)$
      $node.nodetype = OR$
      for $i = 1$ to $k$ do
        $node.children[i] = createGraph(X_i)$
      end for
    end if
  else
    $node = createNode(\text{name} = R)$
    $node.nodetype = AND$
    $node.children[1] = createGraph(R')$
    $node.children[2] = createGraph(<regexp>)$
  end if
  Return $node$
end if
```

Figure 5: $createGraph(R)$

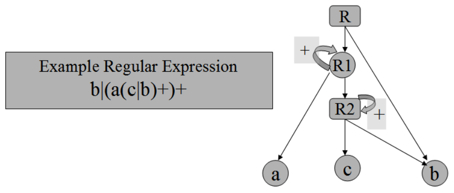


Figure 6: An example regular expression and corresponding AND/OR graph

4.1 Handling ‘?’ and Kleen Operators

The isOpt and selfLoop properties of a node are set if the corresponding regular expression is of the form $R?$, $R+$ or $R*$. To handle the $R?$ case we associate a new property $isOpt$ with the output list $L(v)$ from node v, such that $L(v).isOpt = 1$ if the $v.isOpt = 1$. We also define two operations $consint_\epsilon$ in Figure 7 and $merge_\epsilon$ which account for the $isOpt$ property of their argument lists. For $consint_\epsilon$, the generated list has its $isOpt$ set to 1 if and only if both the argument lists have their $isOpt$ property set to 1. The $merge_\epsilon$ operation remains the same as $merge$, except that the resultant list has $isOpt$ set to 1 if any of its argument lists has $isOpt$ set to 1. The worst case time taken by $consint_\epsilon$ is bounded by 1 $consint$ and 2 $merge$ operations.

To handle the $R+$ case, we define a new operator $consint_\epsilon(L, +)$ which returns a postings list L', such that each entry in the returned list points to a token sequence consisting of all $k \in [1, \Delta]$ consecutive subsequences $@s_1, @s_2 \ldots @s_k$, each $@s_i, 1 \leq i \leq k$ being an entry in L. A simple linear pass through L is sufficient to obtain $consint(L, +)$. The time complexity of this operation is linear in the size of L'. The $isOpt$ property of the result list L' is set to the same value as its argument list L.

Figure 6 shows an example regular expression and its corresponding AND/OR graph; AND nodes are shown as circles whereas OR nodes are shown as square boxes. Nodes having isOpt and selfLoop properties are labeled with $+$, $*$ or $?$. Any AND/OR graph thus constructed is acyclic. The edges in the graph represent dependency between computing nodes. The main regular expression is at the root node of the graph. The leaf nodes correspond to symbols in Σ. Figure 8 outlines the algorithm for computing the postings list of a regular expression by operating bottom-up on the AND/OR graph.

```
$consint_\epsilon(L, L')$
if ((L.isOpt == 0) and (L'.isOpt == 0)) then
    Return $consint(L, L')$
end if
if ((L.isOpt == 0) and (L'.isOpt == 1)) then
    Return $merge(L, consint(L, L'))$
end if
if ((L.isOpt == 1) and (L'.isOpt == 0)) then
    Return $merge(consint(L, L'), L')$
end if
if ((L.isOpt == 1) and (L'.isOpt == 1)) then
    $t = merge(consint(L, L'), L')$
    Return $merge(t, L)$
end if
```

Figure 7: $consint_\epsilon$

```
for Each node $v$ in the reverse topological sorting of $G_R$ do
    if $v.nodetype == AND$ then
        Let $v_1$ and $v_2$ be the children of $v$
        $L(v) = consint_\epsilon(L(v_1), L(v_2))$
    else if $v.type == OR$ then
        $L(v) = merge_\epsilon(L(v.child1), \cdots, L(v.childn))$
    end if
    if $v.selfLoop == 1$ then
        $L(v) = consint_\epsilon(L(v), +)$
    end if
    if $v.isOpt == 1$ then
        $L(v).isOpt = 1$
    end if
end for
```

Figure 8: The algorithm for computing postings list of a regular expression R using the inverse index I and the corresponding AND/OR graph G_R

5 Experiments and Results

In this section, we present empirical comparison of performance of the index-based annotation technique (Section 4) against annotation based on the 'document paradigm' using GATE. The experiments were performed on two data sets, *viz.*, (i) the enron email data set[2] and (ii) a combination of Reuters-21578 data set[3] and the 20 Newsgroups data set[4]. After cleaning, the former data set was 2.3 GB while the latter was 93 MB in size. Our code is entirely in Java. The experiments were performed on a dual 3.2GHz Xeon server with 4 GB RAM. The code for creation of the index was custom-built in Java. Prior to indexing, the sentence segmentation and tokenization of each data set was performed using in-house Java versions of standard tools[5].

5.1 Rule Specification using JAPE

JAPE is a version of CPSL[6] (Common Pattern Specification Language). JAPE provides finite state transduction over annotations based on regular expressions. The JAPE grammar requires information from two main resources: (i) a tokenizer and (ii) a gazetteer.

(1) *Tokenizer:* The tokenizer splits the text into very simple tokens such as numbers, punctuation and words of different types. For example, one might distinguish between words in uppercase and lowercase, and between certain types of punctuation. Although the tokenizer is ca pable of much deeper analysis than this, the aim is to limit its work to maximise efficiency, and enable greater flexibility by placing the burden on the grammar rules, which are more adaptable. A rule has a left hand side (LHS) and a right hand side (RHS). The LHS is a regular expression which has to be matched on the input; the RHS describes the annotations to be added to the Annotation Set. The LHS is separated from the RHS by '$>$'. The following four operators can be used on the LHS: '$|$', '?', '$*$' and '$+$'. The RHS uses ';' as a separator between statements that set the values of the different attributes. The following tokenizer rule identifies each character sequence that begins with a letter in upper case and is followed by 0 or more letters in lower case:

```
"UPPERCASELETTER" "LOWERCASELETTER"*
>>> Token; orth=upperInitial; kind=word;
```

Each such character sequence will be annotated as type "Token". The attribute "orth" (orthography) has the value "upperInitial"; the attribute "kind" has the value "word".

(2) *Gazetteer:* The gazetteer lists used are plain text files, with one entry per line. Each list represents a set of names, such as names of cities, organizations, days of the week, *etc.* An index file is used to access these lists; for each list, a major type is specified and, optionally, a minor type. These lists are compiled into finite state machines. Any text tokens that are matched by these machines will be annotated with features specifying the major and minor types. JAPE grammar rules

[2]http://www.cs.cmu.edu/~enron/

[3]http://www.daviddlewis.com/resources/testcollections/reuters21578/

[4]http://people.csail.mit.edu/jrennie/20Newsgroups/

[5]http://l2r.cs.uiuc.edu/~cogcomp/tools.php

[6]A good description of the original version of this language is in Doug Appelt's TextPro manual: http://www.ai.sri.com/~appelt/TextPro.

then specify the types to be identified in particular circumstances.

The JAPE Rule: Each JAPE rule has two parts, separated by “–>”. The LHS consists of an annotation pattern to be matched; the RHS describes the annotation to be assigned. A basic rule is given as:

```
Rule::=
<rule> <ident> ( <priority> <integer> )?
LeftHandSide ">>>" RightHandSide
```

(1) *Left hand side:* On the LHS, the pattern is described in terms of the annotations already assigned by the tokenizer and gazetteer. The annotation pattern may contain regular expression operators (*e.g.* $*$, ?, $+$). There are 3 main ways in which the pattern can be specified:

1. *value:* specify a string of text, *e.g.* {Token.string == “of”}

2. *attribute:* specify the attributes (and values) of a token (or any other annotation), *e.g.* {Token.kind == number}

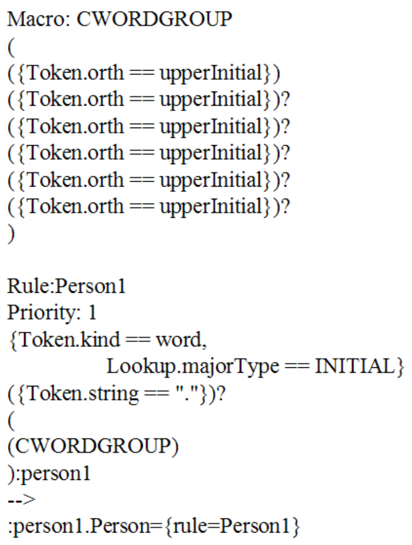
3. *annotation:* specify an annotation type from the gazetteer, *e.g.* {Lookup.minorType == month}

(2) *Right hand side:* The RHS consists of details of the annotations and optional features to be created. Annotations matched on the LHS of a rule may be referred to on the RHS by means of labels that are attached to pattern elements. Finally, attributes and their corresponding values are added to the annotation. An example of a complete rule is:

```
Rule: NumbersAndUnit
(({Token.kind=="number"})+:numbers
{Token.kind=="unit"})
>>>
:numbers.Name={rule="NumbersAndUnit"}
```

This says ‘match sequences of numbers followed by a unit; create a *Name* annotation across the span of the numbers, and attribute rule with value *NumbersAndUnit*’.

Use of context: Context can be dealt with in the grammar rules in the following way. The pattern to be annotated is always enclosed by a set of round brackets. If preceding context is to be included in the rule, this is placed before this set of brackets. This context is described in exactly the same way as the pattern to be matched. If context following the pattern needs to be included, it is placed after the label given to the annotation. Context is used where a pattern should only be recognised if it occurs in a certain situation, but the context itself does not form part of the pattern to be annotated.



```
Macro: CWORDGROUP
(
({Token.orth == upperInitial})
({Token.orth == upperInitial})?
({Token.orth == upperInitial})?
({Token.orth == upperInitial})?
({Token.orth == upperInitial})?
({Token.orth == upperInitial})?
)

Rule:Person1
Priority: 1
{Token.kind == word,
             Lookup.majorType == INITIAL}
({Token.string == "."})?
(
(CWORDGROUP)
):person1
-->
:person1.Person={rule=Person1}
```

Figure 9: An example JAPE rule used in the experiments

For example, the following rule for ‘email-id’s (assuming an appropriate regular expression for “EMAIL-ADD”) would mean that an email address would only be recognized if it occurred inside angled brackets (which would not themselves form part of the entity):

```
Rule: Emailaddress1
({Token.string=="<"})
(
{Token.kind==EMAIL-ADD}
)
:email
({Token.string==">"})
>>>
:email.Address={kind="email",
rule="Emailaddress1"}
```

5.2 Results

In our first experiment, we performed annotation of the two corpora for 4 annotation types using 2 JAPE rules for each type. The 4 annotation types were ‘Person name’, ‘Organization’, ‘Location’ and ‘Date’. A sample JAPE rule for identifying person names is shown in Figure 9. This rule identifies a sequence of words as a person name when each word in the sequence starts with an alphabet in upper-case and when the sequence is immediately preceded by a word from a dictionary of ‘INITIAL’s. Example words in the ‘INITIAL’ dictionary are: ‘Mr.’, ‘Dr.’, ’Lt.’, *etc.*

Table 1 compares the time taken by the index-based annotator against that taken by GATE for the 8 JAPE rules. The index-based annotator performs 8-13 times faster than GATE. Table 2 splits the time mentioned for the index-based annotator in Table 1 into the time taken for the task of computing postings lists for basic entities and derived entities (*c.f.* Section 2) for each of the data sets. We can also observe that a greater speedup is achieved for the larger corpus.

Data set	GATE	Index-based
Enron	4974343	374926
Reuters	752287	92238

Table 1: Time (in milliseconds) for computing annotations using the two techniques

Data set	Orthographic entity types	Gazetteer entity types	Derived entity types
Enron	38285	105870	230771
Reuters	28493	21531	42214

Table 2: Time (in milliseconds) for computing postings lists of entity types

An important advantage of performing annotations over the inverse index is that index entries for basic entity types can be preserved and reused for annotation types as additional rules for annotation are specified by users. For instance, the index entry for ‘Capsword’ might find reuse in several annotation rules. As against this, a document-based annotator has to process each document from scratch for every newly introduced annotation rule. To verify this, we introduced 1 additional rule for each of the 4 named entity types. In Table 3, we compare the time required by the index-based annotator against that required by GATE for annotating the two corpora using the 4 additional rules. We achieve a greater speedup factor of 23-37 for incremental annotation.

Data set	GATE	Index-based
Enron	1479954	62227
Reuters	661157	17929

Table 3: Time (in milliseconds) for computing annotations using the two techniques for the additional 4 rules

6 Conclusions

In this paper we demonstrated that a suitably constructed inverse index contains all the necessary information to implement entity annotators that use cascading regular expressions. The approach has the key advantage of not requiring access to the original unstructured data to compute the annotations. The method uses a basic set of operators on the inverse index to construct indexes to all matches for a regular expression in the tokenized data set. We showed theoretically, that for a DFA implementation, the index approach can be much faster if the index sizes corresponding to the labels on the DFA are a small fraction of the total number of tokens in the data set. We also provided a more efficient index-based implementation that is directly computed from the regular expressions without the need of a DFA conversion and experimentally demonstrated the gains.

References

Eugene Agichtein and Luis Gravano. 2000. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the Fifth ACM International Conference on Digital Libraries*.

Junghoo Cho and Sridhar Rajagopalan. 2002. A fast regular expression indexing engine. In *Proceedings of the* 18^{th} *International Conference on Data Engineering*.

Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. 2001. A fast index for semistructured data. In *The VLDB Conference*, pages 341–350.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications.

H. Cunningham. 1999. Jape – a java annotation patterns engine.

Line Eikvil. 1999. Information extraction from world wide web - a survey. Technical Report 945, Norweigan Computing Center.

Quanzhong Li and Bongki Moon. 2001. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370.

Andrew McCallum, Dayne Freitag, and Fernando Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. In *Proc. 17th International Conf. on Machine Learning*, pages 591–598. Morgan Kaufmann, San Francisco, CA.