# Using Suffix Arrays to Compute Term Frequency and Document Frequency for All Substrings in a Corpus

Mikio Yamamoto
University of Tsukuba
1-1-1 Tennodai,
Tsukuba 305-8573, JAPAN
myama@is.tsukuba.ac.jp

Kenneth W. Church
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932, U.S.A
kwc@research.att.com

## Abstract

Mutual Information (MI) and similar measures are often used in corpus-based linguistics to find interesting ngrams. MI looks for bigrams whose term frequency (*tf*) is larger than chance. Residual Inverse Document Frequency (RIDF) is similar, but it looks for ngrams whose document frequency (*df*) is larger than chance. Previous studies have tended to focus on relatively short ngrams, typically bigrams and trigrams. In this paper, we will show that this approach can be extended to arbitrarily long ngrams. Using suffix arrays, we were able to compute *tf*, *df* and RIDF for all ngrams in two large corpora, an English corpus of 50 million words of Wall Street Journal news articles and a Japanese corpus of 216 million characters of Mainichi Shimbun news articles.

## 1 MI and RIDF

Mutual Information (MI), $I(x;y)$, compares the probability of observing word $x$ and word $y$ together (the joint probability) with the probabilities of observing x and y independently (chance).

$$I(x;y) = \log ( P(x,y) / P(x)P(y) )$$

MI has been used to identify a variety of interesting linguistic phenomena, ranging from semantic relations of the doctor/nurse type to lexico-syntactic co-occurrence preferences of the save/from type (Church and Hanks, 1990).

Church and Gale (1995) proposed Residual Inverse Document Frequency (RIDF), the difference between the observed IDF and what would be expected under a Poisson model for a random word or phrase with comparable frequency. RIDF is a variant of IDF, a standard method for weighting keywords in Information Retrieval (IR). Let $D$ be the number of documents, *tf* be the term frequency (what we call "*frequency*" in our field) and *df* be the document frequency (the number of documents which contain the word or phrase at least once). RIDF is defined as:

$$\begin{aligned} \textit{Residual IDF} &\equiv \textit{observed IDF} - \textit{predicted IDF} \\ &= -\log(df/D) + \log(1-\exp(-\theta)) \\ &= -\log(df/D) + \log(1-\exp(-tf/D)). \end{aligned}$$

RIDF is, in certain sense, like MI; both are the log of the ratio between an empirical observation and a chance-based estimate. Words or phrases with high RIDF or MI have distributions that cannot be attributed to chance. However, the two measures look for different kinds of deviations from chance. MI tends to pick out general vocabulary, the kind of words one would expect to find in a dictionary, whereas RIDF tends to pick out good keywords, the kind of words one would not expect to find in a dictionary. This distinction is not surprising given the history of the two measures; MI, as it is currently used in our field, came from lexicography whereas RIDF came from Information Retrieval.

In addition, it is natural to compute RIDF for all substrings. This is generally not done for MI, though there are many ways that MI could be generalized to apply to longer ngrams. In the next section, we will show an algorithm based on suffix

arrays for computing *tf*, *df* and RIDF for all substrings in a corpus in $O(N\log N)$ time.

In section 3, we will compute RIDF's for all substrings in a corpus and compare and contrast MI and RIDF experimentally for phrases in a English corpus and words/phrases in a Japanese corpus. We won't try to argue that one measure is better than the other; rather we prefer to view the two measures as mutually complementary.

## 2 Computing tf and df for all substrings

### 2.1 Suffix arrays

A suffix array is a data structure designed to make it convenient to compute term frequencies for all substrings in a corpus. Figure 1 shows an example of a suffix array for a corpus of $N$=6 words. A suffix array, s, is an array of all $N$ suffixes, pointers to substrings that start at position i and continue to the end of the corpus, sorted alphabetically. The following very simple C function, suffix_array, takes a corpus as input and returns a suffix array.

```
int suffix_compare(char **a, char **b){
    return strcmp(*a, *b); }

/* The input is a string, terminated with a null */
char **suffix_array(char *corpus){
    int i, N = strlen(corpus);
    char **result=(char **)malloc(N*sizeof(char *));
    /* initialize result[i] with the ith suffix */
    for(i=0; i < N; i++) result[i] = corpus + i;
    qsort(result, N, sizeof(char *), suffix_compare);
    return result; }
```

Nagao and Mori (1994) describe this procedure, and report that it works well on their corpus, and that it requires $O(N\log N)$ time, assuming that the sort step requires $O(N\log N)$ comparisons, and that each comparison requires $O(1)$ time. We tried this procedure on our two corpora, and it worked well for the Japanese one, but unfortunately, it can go quadratic for a corpus with long repeated substrings, where strcmp takes $O(N)$ time rather than $O(1)$ time. For our English corpus, after 50 hours of cpu time, we gave up and turned to Doug McIlroy's implementation (*http://cm.bell-labs.com/cm/cs/ who/doug/ssort.c*) of Manber and Myers' (1993) algorithm, which took only 2 hours. For a corpus that would otherwise go quadratic, the Manber and Myers' algorithm is well worth the effort, but otherwise, the procedure described above is simpler, and often a bit faster.

As mentioned above, suffix arrays were designed to make it easy to compute term frequencies (*tf*). If you want the term frequency of "to be," you can do a binary search to find the first and last position in the suffix array that start with this phrase, $i$ and $j$, and then $tf$("to be") = $j$-$i$+1. In this case, i=5 and j=6, and consequently, $tf$("to be")=6-5+1=2. Similarly, $tf$("be")= 2-1+1 = 2, and $tf$("to")=6-5+1=2. This straightforward method of computing tf requires $O(\log N)$ string comparisons, though as before, each string comparison could take $O(N)$ time. There are more sophisticated algorithms that take $O(\log N)$ time, even for corpora with long repeated substrings.

A closely related concept is *lcp* (longest common prefix). Lcp is a vector of length $N$, where $lcp[i]$ indicates the length of the common prefix between the $i$th suffix and the $i+1$st suffix in the suffix array. Manber and Myers (1993) showed how to compute the lcp vector in $O(N\log N)$ time, even for corpora with long repeated substrings, though for many corpora, the complications required to avoid quadratic behavior are unnecessary.

Corpus: "to be or not to be" Alphabet: {to, be, or, not}

| | 1 | 2 | 3 | 4 | ... | lcp |
|---|---|---|---|---|---|---|
| s[1] | be | | | | | 1 |
| s[2] | be | or | not | to | be | 0 |
| s[3] | not | to | be | | | 0 |
| s[4] | or | not | to | be | | 0 |
| s[5] | to | be | | | | 2 |
| s[6] | to | be | or | not | to be | 0 |

Lcp's are denoted by bold vertical lines as well as the lcp table.

**Figure 1**: An example of a Suffix Array with lcp's

### 2.2 Classes of substrings

Thus far we have seen how to compute *tf* for a single ngram, but how do we compute *tf* and *df* for all ngrams? There are $N(N+1)/2$ substrings in a text of size $N$. If every substring has a different *tf* and *df*, the counting algorithm would require at least quadratic time and space. Fortunately many substrings have the same *tf* and the same *df*. We

will cluster the $N(N+1)/2$ substrings into at most $2N-1$ classes and compute *tf* and *df* over the classes. There will be at most $N$ distinct values of RIDF.

Let $\langle i,j \rangle$ be an interval on the suffix array: $\{s[i], s[i+1], ..., s[j]\}$. We call the interval *LCP-delimited* if the lcp's are larger inside the interval than at its boundary:

$$\min(lcp[i], lcp[i+1], ..., lcp[j-1]) > \max(lcp[i-1], lcp[j]) \quad (1)$$

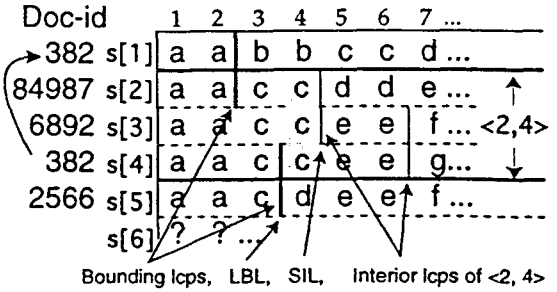
In Figure 1, for example, the interval <5,6> is *LCP-delimited*, and as a result, *tf*("to") = *tf*("to be") = 2, and *df*("to")=*df*("to be").

The interval <5,6> is associated with a class of substrings: "to" and "to be." Classes will turn out to be important because all of the substrings in a class have the same *tf* (*property* 1) and the same *df* (*property* 2). In addition, we will show that classes partition the set of substrings (*property* 3) so that we can compute *tf* and *df* on the classes, rather than substrings. Doing so is much more efficient because there many fewer classes than substrings (*property* 4).

Classes of substrings are defined to be the (not necessarily least) common prefixes in an interval. In Figure 1, for example, both "to" and "to be" are common prefixes throughout the interval <5,6>. That is, every suffix in the interval <5,6> starts with "to," and every suffix also starts with "to be". More formally, we define *class*($\langle i,j \rangle$) as: $\{s[i]_m | LBL < m \leq SIL\}$, where $s[i]_m$ is a substring (the first $m$ characters of $s[i]$), *LBL*(*longest boundary lcp*) is the right hand of (1) and *SIL* (*shortest interior lcp*) is the left hand side*) of (1). In Figure 1, for example, $SIL(\langle 5,6 \rangle) = \min(lcp[5]) = 2$, $LBL(\langle 5,6 \rangle) = \max(lcp[4], lcp[6]) = 0$, and $class(\langle 5,6 \rangle) = \{s[5]_m | 0 < m \leq 2\} = \{$"to", "to be"$\}$.

Figure 2 shows six *LCP-delimited* intervals and the *LBL* and *SIL* of <2,4>. For <2,4>, the bounding lcp's are $lcp[1] = 2$ and $lcp[4]=3$ (*LBL*=3), and the interior lcp's are $lcp[2]=4$ and $lcp[3]=6$ (*SIL*=4). The interval <2,4> is *LCP-delimited*, because *LBL*<*SIL*. $Class(\langle 2,4 \rangle) = \{s[2]_m | 3 < m \leq 4\} = \{aacc\}$. The interval <3,3> is

*) *SIL*($\langle i,i \rangle$) is defined to be infinity, and consequently, all intervals $\langle i,i \rangle$ are *LCP-delimited*, forall *i*.




Vertical lines denote lcps. Gray area denotes endpoints of substrings in class(<2,4>).

| LCP-delimited interval | Class | LBL | SIL | tf |
|---|---|---|---|---|
| <2,4> | {aacc} | 2 | 4 | 3 |
| <3,4> | {aacce, aaccee} | 3 | 6 | 2 |
| <1,1> | {aab, aabb, aabbc, ...} | 2 | infinity | 1 |
| <2,2> | {aaccd, aaccdd, ...} | 4 | infinity | 1 |
| <3,3> | {aacceef, ...} | 6 | infinity | 1 |
| <4,4> | {aacceeg, ...} | 6 | infinity | 1 |

**Figure 2**: Examples of intervals and classes

*LCP-delimited* because *SIL* is infinite and *LBL=6*. The interval <2,3> is not *LCP-delimited* because *SIL* is 4 and *LBL* is 6 (*LBL*>*SIL*).

By construction, the suffixes within the interval $\langle i,j \rangle$ all start with the substrings in $class(\langle i,j \rangle)$, and no suffixes outside this interval start with these substrings. As a result, if $s1$ and $s2$ are two substrings in $class(\langle i,j \rangle)$ then

***Property 1:*** $tf(s1) = tf(s2) = j-i+1$

and

***Property 2:*** $df(s1) = df(s2)$.

The calculation of *df* is more complicated than *tf*, and will be discussed in section 2.4.

It is not uncommon for an *LCP-delimited* interval to be nested within another. In Figure 2, for example, the interval <3,4> is nested within <2,4>. The computation of *df* in section 2.4 will take advantage of a very convenient nesting property. Given two *LCP-delimited* intervals, either one is nested within the other (e.g., <2,4> and <3,4>), or one precedes the other (e.g., <2,2> and <3,4>), but they cannot overlap. Thus, for example, the intervals <1,3> and <2,4> cannot both be *LCP-delimited* because they overlap. Because of this nesting property, it is possible to express the *df* of an interval recursively in terms of its constituents or subintervals.

As mentioned above, we will use the following partitioning property so that we can

compute *tf* and *df* on the classes rather than on the substrings.

***Property 3:*** *the classes partition the set of all substrings in a text.*

There are two parts to this argument: every substring belongs to at most one class (*property* 3a), and every substring belongs to at least one class (*property* 3b).

Demonstration of *property* 3a (proof by contradiction): Suppose there is a substring, *s*, that is a member of two classes: *class*(<*i,j*>) and *class*(<*u,v*>). There are three possibilities: one interval precedes the other, they are property nested or they overlap. The only interesting case is the nesting case. Suppose without loss of generality that <*u,v*> is nested within <*i,j*> as in Figure 3. Because <*u,v*> is *LCP-delimited*, there must be a bounding lcp of <*u,v*> that is smaller than any lcp within <*u,v*>. This bounding lcp must be within <*i,j*>, and as a result, *class*(<*i,j*>) and *class*(<*u,v*>) must be disjoint. Therefore, *s* cannot be in both classes.

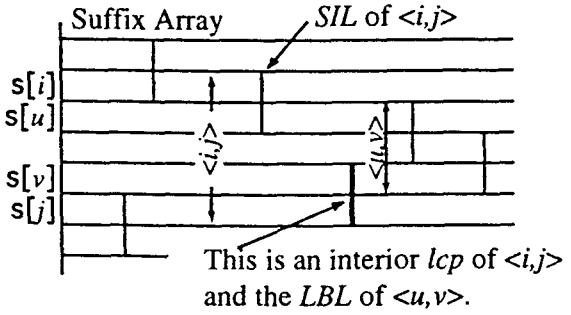



**Figure 3**: An example of nested intervals

Demonstration of *property* 3b (constructive argument): Let *s* be an arbitrary substring in the corpus. There will be at least one suffix in the suffix array that starts with *s*. Let *i* be the first such suffix and let *j* be the last such suffix. By construction, the interval <*i,j*> is *LCP-delimited* (*LBL*(<*i,j*>) < |*s*| and *SIL*(<*i,j*>) ≥ |*s*|), and *s* is an element of *class*(<*i,j*>).

Finally, as mentioned above, computing over classes is much more efficient than computing over the substrings themselves because there are many fewer classes (at most 2*N*-1) than substrings (*N*(*N*+1)/2).

***Property 4:*** *There are N classes with tf=1 and at most N-1 classes with tf > 1.*

The first clause is relatively straightforward. There are *N* intervals <*i,i*>. These are all and only the intervals with *tf*=1. By construction, these intervals are *LCP-delimited.*

To argue the second clause, we will make use of a uniqueness property: an *LCP-delimited* interval <*i,j*> can be uniquely determined by its *SIL* and a representative element *k* (*i*≤*k*<*j*). Suppose there were two distinct intervals, <*i,j*> and <*u,v*>, with the same *SIL*, *SIL*(<*i,j*>)= *SIL*(<*u,v*>), and the same representative, *i*≤*k*<*j* and *u*≤*k*<*v*. Since they share a common representative, *k*, the two intervals must overlap. But since they are distinct, there must be a distinguishing element, *d*, that is in one but not the other. One of these distinguishing elements, *d*, would have to be a bounding lcp in one and an interior lcp in the other. But then the two intervals couldn't both be *LCP-delimited.*

Given this uniqueness property, we can determine the *N*-1 upper bound on the number of *LCP-delimited* intervals by considering the *N*-1 elements in the lcp vector. Each of these elements, *lcp*[*k*], has the opportunity to become the *SIL* of an *LCP-delimited* interval <*i,j*> with a representative *k*. Thus there could be as many as *N*-1 *LCP-delimited* intervals (though there could be fewer if some of the opportunities don't work out). Moreover, there couldn't be any more intervals with *tf*>1, because if there were one, its *SIL* should have been in the lcp vector. (Note that this lcp counting argument excludes intervals with *tf*=1 discussed above, because their *SIL*s need not be in the lcp vector.)

From *property* 4, it follows that there are at most *N* distinct values of RIDF. The *N* intervals <*i,i*> have just one RIDF value since *tf*=*df*=1 for these intervals. The other *N*-1 intervals could have another *N*-1 RIDF values.

In summary, the four *properties* taken collectively make it practical to compute *tf*, *df* and RIDF over a relatively small number of classes; it would have been prohibitively expensive to compute these quantities directly over the *N*(*N*+1)/2 substrings.

## 2.3 Calculating classes using Suffix Array

This section will describe a single pass procedure for computing classes. Since *LCP-delimited* intervals obey a convenient nesting property, the procedure is based on a push-down stack. The procedure outputs 4-tuples, <*s*[*i*],*LBL*,*SIL*,*tf*>, one for each *LCP-delimited* interval. The stack elements are pairs (*x*,*y*), where *x* is an index, typically the left edge of a candidate *LCP-delimited* interval, and *y* is the *SIL* of this candidate interval. Typically, *y*=*lcp*[*x*], though not always, as we will see in Figure 5.

The algorithm sweeps over the suffixes in suffix array *s*[*1..N*] and their *lcp*[*1..N*] (*lcp*[*N*]=*0*) successively. While lcp's of suffixes are monotonically increasing, indexes and lcp's of the suffixes are pushed into a stack. When it finds the *i*-th suffix whose *lcp*[*i*] is less than the lcp on the top of the stack, the index and lcp on the top are popped off the stack. Popping is repeated until the lcp on the top becomes less than the *lcp*[*i*].

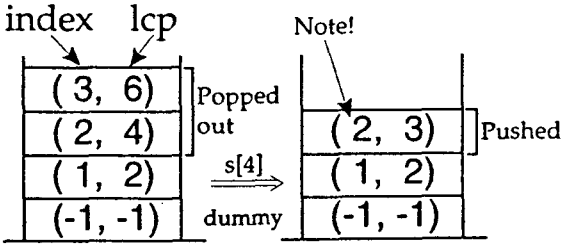
A stack element popped out generates a class. Suppose that a stack element composed of an index *i* and *lcp*[*i*] is popped out by *lcp*[*j*]. *Lcp*[*i*] is used as the *SIL*. The *LBL* is the lcp on the next top element in the stack or *lcp*[*j*]. If the next top lcp will be popped out by *lcp*[*j*], then the algorithm uses the next top lcp as the *LBL*, else it uses the *lcp*[*j*]. *Tf* is the offset between the indexes *i* and *j*, that is, *j-i+1*.

Figure 4 shows the detailed algorithm for computing all classes with *tf* > 1. If classes with *tf* = 1 are needed, we can easily add the line to output those into the algorithm. The expressions, *push*(*x*,*y*) and *pop*(*x*,*y*), operate on the stack in the obvious way, but note that *x* and *y* are inputs for push and outputs for pop. The expression, *top*(*x*,*y*), is equivalent to *pop*(*x*,*y*) followed by *push*(*x*,*y*); it reads the top of the stack without changing the stack pointer.

```
Create and clear stack.
Push (-1, -1) (dummy).
Repeat i = 1, ... , N do
  top (index1, lcp1).
  if lcp[i] > lcp1 then
    push (i, lcp[i]).
  else
    while lcp[i] ≤ lcp1 do
      pop(index1, lcp1)
      top (index2, lcp2)
      if lcp[i] ≤ lcp2 then
        output <s[index1], lcp2, lcp1, i-index1+1>
      else
        output <s[index1], lcp[i], lcp1, i-index1+1>
        push (index1, lcp[i])
        lcp1 = lcp2.
```

**Figure 4**: An algorithm for computing all classes

As mentioned above, the stack elements are typically pairs (*x*,*y*) where *y*=*lcp*[*x*], but not always. Pairs are typically pushed onto the stack by line 6, *push*(*i*, *lcp*[*i*]), and consequently, *y*=*lcp*[*x*], in many cases, but some pairs are pushed on by line 15. Figure 5 (a) shows the typical case with the suffix array in Figure 2. At this point, *i*=3 and the stack contains 4 pairs, a dummy element (-1, -1), followed by three pairs generated by line 6: (1, *lcp*[1]), (2, *lcp*[2]), (3, *lcp*[3]). In contrast, Figure 5 (b) shows an atypical case. In between snapshot (a) and snapshot (b), two *LCP-delimited* intervals were generated, <*s*[3], 4, 6, 2> and <*s*[2], 3, 4, 3>, and then the pair (2, 3) was pushed onto the stack by line 15, *push*(index1, *lcp*[*i*]), to capture the fact that there is a candidate *LCP-delimited* interval starting at index1=2, spanning past the representative element *i*=4, with an *SIL* of *lcp*[*i*=4].




(a) end of processing s[3] (b) end of processing s[4]

**Figure 5**: Snapshots of the stack

## 2.4 Computing df for all classes

This section will extend the algorithm in Figure 4 to include the calculation of *df*. Straightforwardly computing *df* independently for each class would require at least quadratic time, because the program must check document id's for all substrings (*N* at most) in all classes (*N-1* at most). Instead of this, we will take advantage of the nesting property of intervals. The *df* for one

interval can be computed recursively in terms of its constituents (nested subintervals), avoiding unnecessary recomputation.

The stack elements in Figure 5 is augmented with two additional counters: (1) a *df* counter for summing the *df*'s over the nested subintervals and (2) a duplication counter for adjusting for overcounting documents that are referenced in multiple subintervals. The *df* for an interval is simply the difference of these two counters, that is, the sum of the *df*'s of the subintervals, minus the duplication. A C code implementation can be found at

*http://www.milab.is.tsukuba.ac.jp/~myama/tfdf/tfdf.c.*

The *df* counters are relatively straightforward to implement. The crux of the problem is the adjustment for duplication. The adjustment makes use of a document link table, as illustrated in Figure 6. The left two columns indicate that suffixes *s*[101], *s*[104] and *s*[107] are

| Suffix | Document id | Document link (index) |
|---|---|---|
| s[101] | 382 | 66 |
| s[102] | 84987 | 72 |
| s[103] | 6892 | 21 |
| s[104] | 382 | 101 |
| s[105] | 2566 | 12 |
| s[106] | 6892 | 103 |
| s[107] | 382 | 104 |
| s[108] | 84987 | 102 |

**Figure 6**: An example of document link table

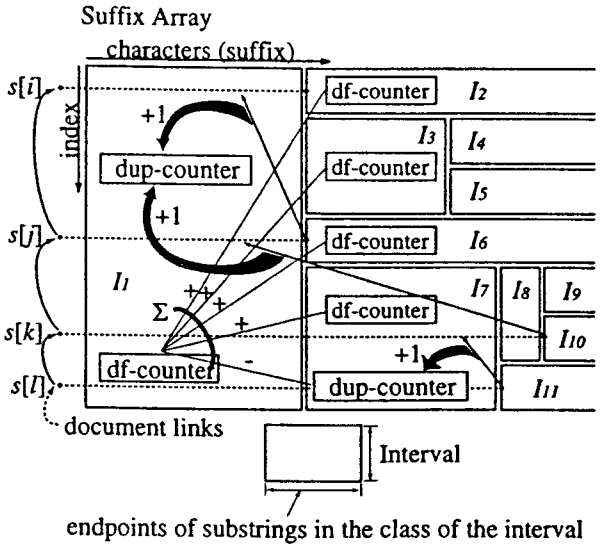



**Figure 7**: *Df* relations among an interval and its constituents

all in document 382, and that several other suffixes are also in the same documents. The third column links together suffixes that are in the same document. Note, for example, that there is a pointer from suffix 104 to 101, indicating that *s*[104] and *s*[101] are in the same document. The suffixes in one of these linked lists are kept sorted by their order in the suffix array. When the algorithm is processing *s*[*i*], the algorithm searches the stack to find the suffix, *s*[*k*], with the largest *k* such $k \leq i$ and *s*[*i*] and *s*[*k*] are in the same document. This search can be performed in $O(\log N)$ time.

Figure 7 shows the *LCP-delimited* intervals in a suffix array and four suffixes included in the same document. *I1* has four immediate constituents of intervals. *S*[*j*] is included in the same document of *s*[*i*]. Count for the document of *s*[*j*] will be duplicated at computing *df* of *I1*. At the point of processing *s*[*j*], the algorithm will increment duplication-counter of *I1* to cancel *df* count of *s*[*j*]. As the same way, *df* count of *s*[*k*] has to canceled at computing *df* of *I1*.

Figure 8 shows a snapshot of the stack after processing *s*[*4*] in Figure 2. Each stack element is a 4-tuple of the index of suffix array, lcp, df-counter and duplication-counter, (*i, lcp, df, dc*). Figure 2 shows *s*[*1*] and *s*[*4*] are in the same document. Looking up the document link table, the algorithm knows *s*[*1*] is the nearest suffix which is in the same document of *s*[*4*]. The duplication-counter of the element of *s*[*1*] is incremented. The duplication of counting *s*[*1*] and *s*[*4*] for the class generated by *s*[*1*] will be avoided using this duplication-counter.

At some processing point, the algorithm uses only a part of the document link table. It

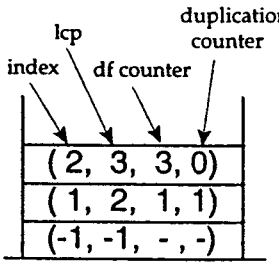



**Figure 8**: A snapshot of the stack in *df* computing

| Doc-id | Nearest index |
|---|---|
| ... | |
| 382 | 4 |
| ... | |
| 6892 | 3 |
| ... | |
| 84987 | 2 |

**Figure 9**: Nearest indexes of documents

needs only the nearest index on the link, but not the whole of the link. So we can compress the link table to dynamic one in which an entry of each document holds the nearest index. Figure 9 shows the nearest index table of document after processing *s*[4].

The final algorithm to calculate all classes with *tf* and *df* takes $O(NlogN)$ time and $O(N)$ space in the worst case.

## 3 Experimental results

### 3.1 RIDF and MI for English and Japanese

We computed all RIDF's for all substrings of two corpora, Wall Street Journal of ACL/DCI in English (about 50M words and 113k articles) and Mainichi News Paper 1991-1995 (CD-Mainichi Shimbun 91-95) in Japanese (about 216M characters and 436k articles) , using the algorithm in the previous section. In English, we tokenized the text into words, delimited by white space, whereas in Japanese we tokenized the text into characters (usually 2-bytes) because Japanese text has no word delimiter such as white space.

It took a few hours to compute all RIDF's using the suffix array. It takes much longer to compute the suffix array than to compute *tf* and *df*. We ignored substrings with $tf < 10$ to avoid noise, resulting in about 1.6M English phrases (#classes = 1.4M) and about 15M substrings of Japanese words/phrases (#classes = 10M).

MI of the longest substring of each class was also computed by the following formula.

$$MI(xyz) = \log \frac{p(xyz)}{p(xy)p(z \mid y)}$$

Where $xyz$ is a phrase or string, $x$ and $z$ are a word or a character and $y$ is a sub-phrase or sub-string.

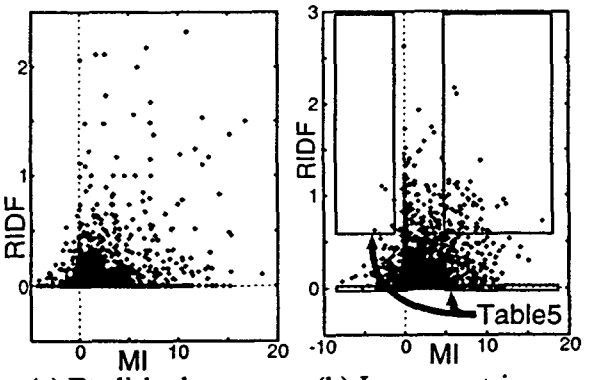
### 3.2 Little correlation between RIDF and MI

We are interested in comparing and contrasting RIDF and MI. Figure 10 (a) plots RIDF vs MI for phrases in WSJ (length > 1), showing little, if any, correlation between RIDF and MI. Figure 10 (b) also plots RIDF vs MI but this time the corpus is in Japanese and the words were manually selected by the newspaper to be keywords. Both Figures 10 (a) and 10 (b) suggest that RIDF and MI are largely independent. There are many substrings with a large RIDF value and a small MI, and vice versa.

MI is very different from RIDF. Both pick out interesting phrases, but phrases with large MI are interesting in different ways from phrases with large RIDF. Consider the phrases in Table 1, which all contain the word "having." These phrases have large MI values and small RIDF values. A lexicographer such as Patrick Hanks, who works on dictionaries for learners, might be interested in these phrases because these kinds of collocations tend to be difficult for non-native speakers of the language. On the other hand, these kinds of collocations are not very good keywords.

Table 2 is a random sample of phrases containing the substring /Mr/, sorted by RIDF. The ones at the top of the list tend to be better keywords than the ones further down.

Table 3.A and 3.B show a few phrases starting with /the/, sorted by MI (Table 3.A) and sorted by RIDF (Table 3.B). Most of the phrases are interesting in one way or another, but those at the top of Table 3.A tend to be somewhat



(a) English phrases (b) Japanese strings

**Figure 10**: Scatter plot of RIDF and MI

**Table 1**: phrases with 'having'

| tf | df | RIDF | MI | Phrase |
|---|---|---|---|---|
| 18 | 18 | -0.0001 | 10.4564 | admits to having |
| 14 | 14 | -0.0001 | 9.7154 | admit to having |
| 25 | 23 | 0.1201 | 8.8551 | diagnosed as having |
| 20 | 20 | -0.0001 | 7.4444 | suspected of having |
| 301 | 293 | 0.0369 | 7.2870 | without having |
| 15 | 13 | 0.2064 | 6.9419 | denies having |
| 59 | 59 | -0.0004 | 6.7612 | avoid having |
| 18 | 18 | -0.0001 | 5.9760 | without ever having |
| 12 | 12 | -0.0001 | 5.9157 | Besides having |
| 26 | 26 | -0.0002 | 5.7678 | denied having |

**Table 2**: phrases with 'Mr'

| tf | df | RIDF | MI | Phrase |
|---|---|---|---|---|
| 11 | 3 | 1.8744 | 0.6486 | . Mr. Hinz |
| 18 | 5 | 1.8479 | 6.5583 | Mr. Bradbury |
| 51 | 16 | 1.6721 | 6.6880 | Mr. Roemer |
| 67 | 25 | 1.4218 | 6.7856 | Mr. Melamed |
| 54 | 27 | 0.9997 | 5.7704 | Mr. Burnett |
| 16 | 9 | 0.8300 | 5.8364 | Mrs. Brown |
| 11 | 8 | 0.4594 | 1.0931 | Mr. Eiszner said |
| 53 | 40 | 0.4057 | 0.2855 | Mr. Johnson . |
| 21 | 16 | 0.3922 | 0.1997 | Mr. Nichols said . |
| 13 | 10 | 0.3784 | 0.4197 | . Mr. Shulman |
| 176 | 138 | 0.3498 | 0.4580 | Mr. Bush has |
| 13 | 11 | 0.2409 | 1.5295 | to Mr. Trump's |
| 13 | 11 | 0.2409 | -0.9301 | Mr. Bowman , |
| 35 | 32 | 0.1291 | 1.1673 | wrote Mr. |
| 12 | 11 | 0.1255 | 1.7330 | M r. Lee to |
| 22 | 21 | 0.0670 | 1.4293 | facing Mr. |
| 11 | 11 | -0.0001 | 0.7004 | Mr. Poehl also |
| 13 | 13 | -0.0001 | 1.4061 | inadequate . " Mr. |
| 16 | 16 | -0.0001 | 1.5771 | The 41-year-old Mr. |
| 19 | 19 | -0.0001 | 0.4738 | 14 . Mr. |
| 26 | 26 | -0.0002 | 0.0126 | in November . Mr. |
| 27 | 27 | -0.0002 | -0.0112 | " For his part , Mr. |
| 38 | 38 | -0.0002 | 1.3589 | . AMR , |
| 39 | 39 | -0.0002 | -0.3260 | for instance , Mr. |

**Table 3.A**: Worse Keywords

| tf | df | RIDF | MI | Phrase |
|---|---|---|---|---|
| 11 | 11 | -0.0001 | 11.0968 | the up side |
| 73 | 66 | 0.1450 | 9.3222 | the will of |
| 16 | 16 | -0.0001 | 8.5967 | the sell side |
| 17 | 16 | 0.0874 | 8.5250 | the Stock Exchange of |
| 16 | 15 | 0.0930 | 8.4617 | the buy side |
| 20 | 20 | -0.0001 | 8.4322 | the down side |
| 55 | 54 | 0.0261 | 8.3287 | the will to |
| 14 | 14 | -0.0001 | 8.1208 | the saying goes |
| 15 | 15 | -0.0001 | 7.5643 | the going gets |

**Table 3.B**: Better Keywords

| tf | df | RIDF | MI | Phrase |
|---|---|---|---|---|
| 37 | 3 | 3.6243 | 2.2561 | the joint commission |
| 66 | 8 | 3.0440 | 3.5640 | the SSC |
| 55 | 7 | 2.9737 | 2.0317 | the Delaware & |
| 37 | 5 | 2.8873 | 3.6492 | the NHS |
| 22 | 3 | 2.8743 | 3.3670 | the kibbutz |
| 22 | 3 | 2.8743 | 4.1142 | the NSA's |
| 29 | 4 | 2.8578 | 4.1502 | the DeBartolos |
| 36 | 5 | 2.8478 | 2.3061 | the Basic Law |
| 21 | 3 | 2.8072 | 2.2983 | the national output |

**Table 3.C**: Concordance of the phrase "the Basic Law"

The first col. is the token id and the last col. is the doc id (position of the start word in the corpus)

| | | | |
|---|---|---|---|
| 2229521: | line in the drafting of the Basic | Law that will determine how Hon | 2228648 |
| 2229902: | s policy as expressed in the Basic | Law -- as Gov. Wilson's debut s | 2228648 |
| 9746758: | he U.S. Constitution and the Basic | Law of the Federal Republic of | 9746014 |
| 11824764: | any changes must follow the Basic | Law , Hong Kong's miniconstitut | 11824269 |
| 33007637: | sts a tentative draft of the Basic | Law , and although this may be | 33007425 |
| 33007720: | the relationship between the Basic | Law and the Chinese Constitutio | 33007425 |
| 33007729: | onstitution . Originally the Basic | Law was to deal with this topic | 33007425 |
| 33007945: | wer of interpretation of the Basic | Law shall be vested in the NPC | 33007425 |
| 33007975: | tation of a provision of the Basic | Law , the courts of the HKSAR { | 33007425 |
| 33008031: | interpret provisions of the Basic | Law . If a case involves the in | 33007425 |
| 33008045: | tation of a provision of the Basic | Law concerning defense , foreig | 33007425 |
| 33008115: | etation of an article of the Basic | Law regarding " defense , forei | 33007425 |
| 33008205: | nland representatives of the Basic | Law Drafting Committee fear tha | 33007425 |
| 33008398: | e : Mainland drafters of the Basic | Law simply do not appreciate th | 33007425 |
| 33008488: | pret all the articles of the Basic | Law . While recognizing that th | 33007425 |
| 33008506: | y and power to interpret the Basic | Law , it should irrevocably del | 33007425 |
| 33008521: | pret those provisions of the Basic | Law within the scope of Hong Ko | 33007425 |
| 33008545: | r the tentative draft of the Basic | Law , I cannot help but conclud | 33007425 |
| 33008690: | d of being guaranteed by the Basic | Law , are being redefined out o | 33007425 |
| 33008712: | uncilor , is a member of the Basic | Law Drafting Committee . <EOA> | 33007425 |
| 39020313: | sts a tentative draft of the Basic | Law , and although this may be | 39020101 |
| 39020396: | the relationship between the Basic | Law and the Chinese Constitutio | 39020101 |
| 39020405: | onstitution . Originally the Basic | Law was to deal with this topic | 39020101 |
| 39020621: | wer of interpretation of the Basic | Law shall be vested in the NPC | 39020101 |
| 39020651: | tation of a provision of the Basic | Law , the courts of the HKSAR { | 39020101 |
| 39020707: | interpret provisions of the Basic | Law . If a case involves the in | 39020101 |
| 39020721: | tation of a provision of the Basic | Law concerning defense , foreig | 39020101 |
| 39020791: | etation of an article of the Basic | Law regarding " defense , forei | 39020101 |
| 39020881: | nland representatives of the Basic | Law Drafting Committee fear tha | 39020101 |
| 39021074: | e : Mainland drafters of the Basic | Law simply do not appreciate th | 39020101 |
| 39021164: | pret all the articles of the Basic | Law . While recognizing that th | 39020101 |
| 39021182: | y and power to interpret the Basic | Law , it should irrevocably del | 39020101 |
| 39021197: | pret those provisions of the Basic | Law within the scope of Hong Ko | 39020101 |
| 39021221: | r the tentative draft of the Basic | Law , I cannot help but conclud | 39020101 |
| 39021366: | d of being guaranteed by the Basic | Law , are being redefined out o | 39020101 |
| 39021388: | uncilor , is a member of the Basic | Law Drafting Committee . <EOA> | 39020101 |

**Table 4**: Phrases with prepositions

| tf | df | RIDF | MI | Phrase with 'for' |
|---|---|---|---|---|
| 14 | 14 | -0.0001 | 14.5587 | feedlots for fattening |
| 15 | 15 | -0.0001 | 14.4294 | error for subgroups |
| 12 | 12 | -0.0001 | 14.1123 | Voice for Food |
| 10 | 5 | 0.9999 | 13.7514 | Quest for Value |
| 12 | 4 | 1.5849 | 13.7514 | Friends for Education |
| 13 | 13 | -0.0001 | 13.6803 | Commissioner for Refugees |
| 23 | 21 | 0.1311 | 13.6676 | meteorologist for Weather |
| 10 | 2 | 2.3219 | 13.4009 | Just for Men |
| 10 | 9 | 0.1519 | 13.3591 | Witness for Peace |
| 19 | 16 | 0.2478 | 12.9440 | priced for reoffering |

| tf | df | RIDF | MI | Phrase with 'on' |
|---|---|---|---|---|
| 11 | 5 | 1.1374 | 14.3393 | Terrorist on Trial |
| 11 | 10 | 0.1374 | 13.1068 | War on Poverty |
| 13 | 12 | 0.1154 | 12.6849 | Institute on Drug |
| 16 | 16 | -0.0001 | 12.5599 | dead on arrival |
| 12 | 12 | -0.0001 | 11.5885 | from on high |
| 12 | 12 | -0.0001 | 11.5694 | knocking on doors |
| 22 | 18 | 0.2894 | 11.3317 | warnings on cigarette |
| 11 | 11 | -0.0001 | 11.2137 | Subcommittee on Oversight |
| 17 | 12 | 0.5024 | 11.1847 | Group on Health |
| 22 | 20 | 0.1374 | 11.1421 | free on bail |

| tf | df | RIDF | MI | Phrase with 'by' |
|---|---|---|---|---|
| 11 | 11 | -0.0001 | 12.8665 | piece by piece |
| 13 | 13 | -0.0001 | 12.5731 | guilt by association |
| 13 | 13 | -0.0001 | 12.4577 | step by step |
| 15 | 15 | -0.0001 | 12.4349 | bit by bit |
| 16 | 16 | -0.0001 | 11.8276 | engineer by training |
| 61 | 59 | 0.0477 | 11.5281 | side by side |
| 17 | 17 | -0.0001 | 11.4577 | each by Korea's |
| 12 | 12 | -0.0001 | 11.3059 | hemmed in by |
| 11 | 11 | -0.0001 | 10.8176 | dictated by formula |
| 20 | 20 | -0.0001 | 10.6641 | 70%-owned by Exxon |

| tf | df | RIDF | MI | Phrase with 'of' |
|---|---|---|---|---|
| 11 | 10 | 0.1374 | 16.7880 | Joan of Arc |
| 12 | 5 | 1.2630 | 16.2177 | Ports of Call |
| 16 | 16 | -0.0001 | 16.0725 | Articles of Confederation |
| 14 | 13 | 0.1068 | 16.0604 | writ of mandamus |
| 10 | 9 | 0.1519 | 15.8551 | Oil of Olay |
| 11 | 11 | -0.0001 | 15.8365 | shortness of breath |
| 10 | 9 | 0.1519 | 15.6210 | Archbishop of Canterbur |
| 10 | 8 | 0.3219 | 15.3454 | Secret of My |
| 12 | 12 | -0.0001 | 15.2030 | Lukman of Nigeria |
| 16 | 4 | 1.9999 | 15.1600 | Days of Rage |

**Table 5**: Examples of keywords with interesting RIDF and MI

| RIDF | MI | Substrings | Features |
|---|---|---|---|
| High 10% | Low 10% | 本江(native last name)<br>SUN (company name)<br>エリーダ(foreign name)<br>たわし(brush)<br>ソファー (sofa) | Kanji character<br>English character<br>Katakana character<br>Hiragana character<br>Loan word, Katakana |
| Low 10% | High 10% | ばくだい (huge)<br>受動的 (passive)<br>肝いり(determination)<br>広沢務(native full name)<br>榊直樹(native full name) | General vocabulary<br>General vocab., Kanji<br>General vocabulary<br>Kanji character<br>Kanji character |

idiomatic (in the WSJ domain) whereas those at the top of Table 3.B tend to pick out specific stories or events in the news. For example, the phrase, "the Basic Law," selects for stories about the British handover of Hong Kong to China, as illustrated in Table 3.C.

Table 4 shows a number of phrases with high MI containing common prepositions. The high MI indicates an interesting association, but again most of them are not good keywords, though there are a few exceptions such as "Just for Men," a well-known brand name.

RIDF and MI for Japanese substrings tend to be similar. Substrings with both high RIDF and MI tend to be good keywords such as 合併 (merger), 株券 (stock certificate), 辞典 (dictionary), 無線 (wireless) and so on. Substrings with both low RIDF and MI tend to be poor keywords such as 今期公式戦 (current regular-season game) and meaningless fragments such as 私こ (??). Table 5 shows examples where MI and RIDF point in opposite directions (rectangles in Figure 10 (b)). Words with low RIDF and high MI tend to be general vocabulary (often written in *Kanji* characters). In contrast, words with high RIDF and low MI tend to be domain specific words such as loan words (often written in *Katakana* characters). MI is high for words in general vocabulary (words found in dictionary) and RIDF is high for good keywords for IR.

## 3.3 Word extraction

Sproat and Shih (1990) found MI to be useful for word extraction in Chinese. We performed the following experiment to see if both MI and RIDF are useful for word extraction in Japanese.

We extracted four random samples of 100 substrings each. The four samples cover all four combinations of high and low RIDF and high and low MI, where high is defined to be in the top decile and low is defined to be in the bottom decile. Then we manually scored each sample substring using our own judgment as a good (the substring is a word) or bad the substring is not a word) or gray (the judge is not sure). The results are presented in Table 6, which shows that

Table 6: RIDF and MI are complementary

| | All MI | MI (high 10%) | MI (low 10%) |
|---|---|---|---|
| All RIDF | --- | 20-44% | 2-11% |
| RIDF (high 10%) | 29-51% | 38-55% | 11-35% |
| RIDF (low 10%) | 3-18% | 4-13% | 0-8% |

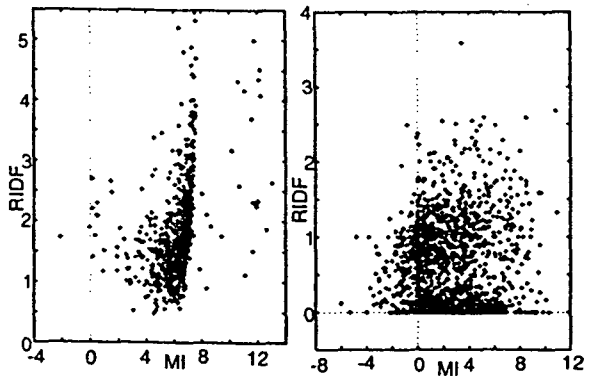
Each cell is computed over a sample of 100 examples. The smaller values are counts of 'good' words and the larger values, 'not bad' words ('good' and 'gray' words). Good or 'not bad' word ratio of pairs of characters with high MI is 51-76%.

substrings with high scores in both dimensions are more likely to be words than substrings that score high in just one dimension. Conversely, substrings with low scores in both dimensions are very unlikely to be words.

### 3.4 Case study: Names

We also compared RIDF and MI for people's names. We made a list of people's names from corpora using simple heuristics. A phrase or substring is accepted as a person's name if English phrase starts with the title 'Mr.' 'Ms.' or 'Dr.' and is followed by a series of capitalized words. For Japanese, we selected phrases in the keyword list ending with '氏' (-shi), which is roughly the equivalent of the English titles 'Mr.' and 'Ms.'

Figure 11 plots RIDF and MI for names in English (a) and Japanese (b) with $tf \geq 10$, respectively. Figure 11 (a) shows that MI has a more limited range than RIDF, suggesting that RIDF may be more effective with names than MI. The English name 'Mr. From' is a particularly




(a) English names (b) Japanese names

**Figure 11**: MI and RIDF of people's names

interesting case, since both 'Mr.' and 'From' is a stop word. In this case, the RIDF was large and the MI was not.

The Japanese names in Figure 11 (b) split naturally at RIDF = 0.5. Japanese names with RIDF below 0.5 are different from names after 0.5. The group whose RIDF is under 0.5 included first name and full name (first and last name) at rate of 90% and another group whose RIDF is up to 0.5 included only last name at rate of 90%. The reason of this separation is that full name (and first name as a substring of full name) appears once in the beginning of the document, but last name is repeated as a reference in the article. Recall that RIDF tends to give higher value to substrings which appear many times in a few documents. In summary, RIDF can discriminate difference of some words which cannot be done by MI.

## 5 Conclusion

We showed that RIDF is efficiently and naturally scalable to long phrases or substrings. RIDF for all substrings in a corpus can be computed using the algorithm which computes *tf*s and *df*s for all substrings based on Suffix Array. It remains an open question how to do this for MI. We found that RIDF is useful for finding good keywords, word extraction and so on. The combination of MI and RIDF is better than either by itself. RI*DF* is like MI, but different.

## References


Church, K. and P. Hanks (1990) Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16:1, pp. 22 - 29.

Church, K. and W. Gale (1995) Poisson mixtures. *Natural Language Engineering*, 1:2, pp. 163 - 190.

Manber, U. and G. Myers (1993) Suffix array: A new method for on-line string searches. *SIAM Journal on Computing*, 22:5, pp. 935 - 948. *http://glimpse.cs.arizona.edu/udi.html*

Nagao, M. and S. Mori (1994) A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of Japanese, *Coling-94*, pp.611-615.

Sproat, R and C. Shih (1990) A statistical method for finding word boundaries in Chinese text. *Computer Processing of Chinese and Oriental Languages*, Vol.4, pp. 336 - 351.