

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

Mikio Yamamoto*
University of Tsukuba

Kenneth W. Church†
AT&T Labs - Research

Bigrams and trigrams are commonly used in statistical natural language processing; this paper will describe techniques for working with much longer ngrams. Suffix arrays were first introduced to compute the frequency and location of a substring (ngram) in a sequence (corpus) of length N . To compute frequencies over all $N(N+1)/2$ substrings in a corpus, the substrings are grouped into a manageable number of equivalence classes. In this way, a prohibitive computation over substrings is reduced to a manageable computation over classes. This paper presents both the algorithms and the code that were used to compute term frequency (tf) and document frequency (df) for all ngrams in two large corpora, an English corpus of 50 million words of Wall Street Journal and a Japanese corpus of 216 million characters of Mainichi Shimbun.

The second half of the paper uses these frequencies to find “interesting” substrings. Lexicographers have been interested in ngrams with high Mutual Information (MI) where the joint term frequency is higher than what would be expected by chance (compositionality). Residual Inverse Document Frequency ($RIDF$) compares document frequency to a different notion of chance, highlighting technical terminology, names and good keywords for information retrieval. The combination of both MI and $RIDF$ is better than either by itself in a Japanese word identification task.

1 Introduction

Suffix arrays will be used to compute a number of statistics of interest, including term frequency and document frequency, for all ngrams in large corpora. Term frequency (tf) and document frequency (df), and functions of these quantities such as mutual information (MI) and inverse document frequency (IDF) have received considerable attention in the corpus-based and information retrieval (IR) literatures (Charniak, 1993; Jelinek, 1997; Sparck Jones, 1972). Term frequency is the standard notion of frequency in corpus-based

* Institute of Information Sciences and Electronics, 1-1-1 Tennodai, Tsukuba 305-8573, JAPAN

† 180 Park Avenue, Florham Park, NJ 07932, U.S.A

natural language processing (NLP). Document frequency is the number of documents that contain the term. Term frequency is an integer between 0 and N , the size of the corpus. Document frequency is an integer between 0 and D , the number of documents in the corpus. These statistics are usually computed over short ngrams such as bigrams and trigrams (Charniak, 1993; Jelinek, 1997). This paper will show how to work with much longer ngrams, including million-grams and even billion-grams.

In corpus-based NLP, term frequencies are often converted into probabilities, using the maximum likelihood estimator (MLE), the Good-Turing method (Katz, 1987) or Deleted Interpolation (Jelinek, 1997, Chapter 15). These probabilities are used in noisy channel applications such as speech recognition to distinguish more likely sequences from less likely sequences, reducing the search space (perplexity) for the acoustic recognizer. In Information Retrieval, document frequencies are converted into inverse document frequency (IDF), which plays an important role in term weighting (Sparck Jones, 1972).

$$IDF(t) = -\log_2 \frac{df(t)}{D}$$

$IDF(t)$ can be interpreted as the number of bits of information the system is given if it is told that the document in question contains the term t . Rare terms contribute more bits than common terms.

Mutual Information (MI) and Residual IDF (RIDF) both compare tf and df to what would be expected by chance, using two different notions of chance. MI compares the frequency of the whole to the frequencies of the parts. A large MI indicates a large deviation from compositionality. RIDF (Church and Gale, 1995) compares the distribution of a term over documents to what would be expected by a random (meaningless) term. Good keywords for information retrieval tend to pick out a relatively small number of documents, unlike random terms which are found in a relatively large number of documents

according to a Poisson distribution.

$$\begin{aligned}
 MI(xYz) &= \log \frac{p(xYz)}{p(xY)p(z|Y)} \\
 &= \log \frac{tf(xYz)}{\frac{N}{tf(xY)} \frac{tf(Yz)}}{\frac{N}{tf(Y)}} \\
 &= \log \frac{tf(xYz)tf(Y)}{tf(xY)tf(Yz)},
 \end{aligned}$$

where x and z are tokens, and Y and xYz are ngrams (sequences of tokens).

$$\begin{aligned}
 \textit{Residual IDF} &= \textit{observed IDF} - \textit{predicted IDF} \\
 &= -\log \frac{df}{D} + \log \left\{ 1 - \frac{\theta^0 \exp(-\theta)}{0!} \right\} \\
 &= -\log \frac{df}{D} + \log \left\{ 1 - \exp\left(-\frac{N}{D} \frac{tf}{N}\right) \right\} \\
 &= -\log \frac{df}{D} + \log \left\{ 1 - \exp\left(-\frac{tf}{D}\right) \right\}
 \end{aligned}$$

The rest of the paper is divided into two sections. Section 2 describes the algorithms and the code that were used to compute term frequencies and document frequencies for all substrings in two large corpora, an English corpus of 50 million words of the Wall Street Journal, and a Japanese corpus of 216 million characters of the Mainichi Shimbun.

Section 3 uses these frequencies to find “interesting” substrings, where what counts as “interesting” depends on the application. MI finds phrases of interest to lexicography, general vocabulary whose distribution is far from compositional, whereas RIDF picks out technical terminology, names and keywords that are useful for Information Retrieval, whose distribution over documents is far from uniform or Poisson. These observations may be particularly useful for Japanese morphology, segmenting Japanese characters into words. Sequences of characters that are high in both MI and RIDF are more likely to be words than sequences that are high in just one, which are more likely than sequences that are high in neither.

2 Computing *tf* and *df* for all substrings

2.1 Suffix arrays

This section will introduce an algorithm based on suffix arrays for computing *tf* and *df* and many functions of these quantities for all substrings in a corpus in $O(N \log N)$ time, even though there are $N(N + 1)/2$ such substrings in a corpus of size N . The algorithm groups the $N(N + 1)/2$ substrings into at most $2N - 1$ equivalence classes. By grouping substrings in this way, many of the statistics of interest can be computed over the relatively small number of classes, which is manageable, rather than over the quadratic number of substrings, which would have been prohibitive.

The suffix array data structure (Manber and Myers, 1990) was introduced as a database indexing technique. Suffix arrays can be viewed as a compact representation of suffix trees (McCreight, 1976; Ukkonen, 1995), a data structure that has been extensively studied over the last thirty years. See (Gusfield, 1997) for comprehensive introduction to suffix trees. Hui (1992) shows how to compute *df* for all substrings using generalized suffix trees. The major advantage of suffix arrays over suffix trees is space. The space requirements for suffix trees (but not for suffix arrays) grow with alphabet size: $O(N|\Sigma|)$ space, where $|\Sigma|$ is the alphabet size. The dependency on alphabet size is a serious issue for Japanese. Manber and Myers (1990) reported that suffix arrays are an order of magnitude more efficient in space than suffix trees even in the case of relatively small alphabet size ($|\Sigma| = 96$). The advantages of suffix arrays over suffix trees becomes much more significant for larger alphabets such as Japanese (and English words).

The suffix array data structure makes it convenient to compute the frequency and location of a substring (ngram) in a long sequence (corpus). The early work was motivated by biological applications such as matching of DNA sequences. Suffix arrays are closely related to PAT arrays, which were motivated in part by a project at the University

of Waterloo to distribute the Oxford English Dictionary with indexes on CD-ROM. PAT arrays have also been motivated by applications in Information Retrieval (Gaston H. Gonnet and Snider, 1992). A similar data structure was proposed by Nagao and Mori (1994) for processing Japanese text.

The alphabet sizes vary considerably in each of these cases. DNA has a relatively small alphabet of just 4 characters whereas Japanese has a relatively large alphabet of more than 5000 characters. The methods scale naturally over alphabet size. In section 3, the data structure is applied to a large corpus of English text, where the alphabet is assumed to be the set of all English words, an unbounded set. It is sometimes assumed that larger alphabets are more challenging than smaller ones, but ironically, it can be just the reverse because there is often an inverse relationship between the size of the alphabet and the length of meaningful/interesting substrings.

This section starts by reviewing the construction of suffix arrays and how they have been used to compute the frequency and locations of a substring in a sequence. We will then show how these methods can be applied to find not only the frequency of a particular substring but also the frequency of all substrings. Finally, the methods are generalized to compute document frequencies as well as term frequencies.

A suffix array, s , is an array of all N suffixes, sorted alphabetically. A suffix, $s[i]$, also known as a semi-infinite string, is a string that starts at position i in the corpus and continues to the end of the corpus. In practical implementations, it is typically denoted by a 4-byte integer, i . In this way, a small (constant) amount of space is used to represent a very long substring, which one might have thought would require N space.

The algorithm, `suffix_array`, presented below takes a corpus and its length N as input, and outputs the suffix array, s .

```
suffix_array ← function(corpus,  $N$ ){
```

Let s be a vector of the integers from 0 to $N - 1$ which denote suffixes starting from $s[i]$ on the *corpus*.

Sort s in alphabetical order of suffixes denoted by each integer.

Return s . }

The C program below implements this algorithm.

```
char *corpus;

int suffix_compare(int *a, int *b){ return strcmp(corpus+*a, corpus+*b);}

int *suffix_array(int n){
    int i, *s = (int *)malloc(n*sizeof(int));
    for(i=0; i < n; i++) s[i] = i; /* initialize */
    qsort(s, n, sizeof(int), suffix_compare); /* sort */
    return s;}

```

Figures 1 and 2 illustrate a simple example where the corpus (“to_be_or_not_to_be”) consists of $N = 18$ (19 bytes): 13 alphabetic characters plus 5 spaces (and 1 null termination). The C program (above) starts by allocating memory for the suffix array (18 integers of 4-bytes each). The suffix array is initialized to the integers from 0 to 17. Finally, the suffix array is sorted by alphabetical order. The suffix array after initialization is shown in Figure 1. The suffix array after sorting is shown in Figure 2.

As mentioned above, suffix arrays were designed to make it easy to compute the frequency (tf) and locations of a substring (ngram/term) in a sequence (corpus). Given a substring or term, t , a binary search is used to find the first and last suffix that start with t . Let $s[i]$ be the first such suffix and $s[j]$ be the last such suffix. Then $tf(t) = j - i + 1$ and the term is located at positions: $\{s[i], s[i + 1], \dots, s[j]\}$, and only these positions.

Input corpus: "to be or not to be"

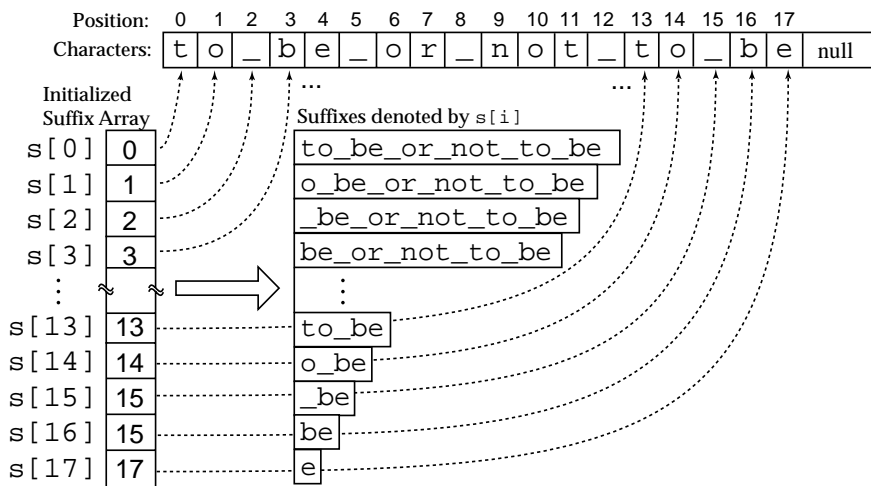


Figure 1
 Illustration of a suffix array, s , that has just been initialized and not yet sorted. Each element in the suffix array, $s[i]$, is an integer denoting a suffix or a semi-infinite string, starting at position i in the corpus and extending to the end of the corpus.

Figure 2 shows how this procedure can be used to compute the frequency and locations of the term “to_be” in the corpus “to_be_or_not_to_be”. As illustrated in the Figure, $s[i = 16]$ is the first suffix to start with the term “to_be” and $s[j = 17]$ is the last suffix to start with this term. Consequently, $tf(\text{“to_be”}) = 17 - 16 + 1 = 2$. Moreover, the term appears at $positions(\text{“to_be”}) = \{s[16], s[17]\} = \{13, 0\}$, and only these positions. Similarly, the substring “to” has the same tf and $positions$, as do the substrings, “to_” and “to_b.” Although there may be $N(N + 1)/2$ ways to pick i and j , it will turn out that we need only consider $2N - 1$ of them when computing tf for all substrings.

This procedure was used by Nagao and Mori (1994) who applied it quite successfully to a large corpus of Japanese text. They report that it takes $O(N \log N)$ time, assuming that the sort step performs $O(N \log N)$ comparisons, and that each comparison can be done in $O(1)$ time. While these are often reasonable assumptions, we have found that if the corpus contains long repeated substrings (e.g., duplicated articles), as our English corpus does (Paul and Baker, 1992), then the sort can consume quadratic time since each

Suffix Array		Suffixes denoted by $s[i]$
$s[0]$	15	_be
$s[1]$	2	_be_or_not_to_be
$s[2]$	8	_not_to_be
$s[3]$	5	_or_not_to_be
$s[4]$	12	_to_be
$s[5]$	16	be
$s[6]$	3	be_or_not_to_be
$s[7]$	17	e
$s[8]$	4	e_or_not_to_be
$s[9]$	9	not_to_be
$s[10]$	14	o_be
$s[11]$	1	o_be_or_not_to_be
$s[12]$	6	or_not_to_be
$s[13]$	10	ot_to_be
$s[14]$	7	r_not_to_be
$s[15]$	11	t_to_be
$s[16]$	13	to_be
$s[17]$	0	to_be_or_not_to_be

Figure 2

Illustration of the suffix array in Figure 1 after sorting. The integers in s are sorted so that the semi-infinite strings are now in alphabetical order.

comparison can take N time. Like Nagao and Mori (1994), we were also able to apply this procedure quite successfully to our Japanese corpus, but for the English corpus, after 50 hours of cpu time, we gave up and turned to Manber and Myers' (1990) algorithm,¹ which took only two hours. Manber and Myers' algorithm uses some clever, but difficult to describe techniques, to achieve $O(N \log N)$ time, even for a corpus with long repeated substrings. For a corpus that would otherwise consume quadratic time, the Manber and Myers' algorithm is well worth the effort, but otherwise, the procedure described above is simpler, and can even be a bit faster.

The "to_be_or_not_to_be" example used the standard English alphabet (one byte per character). As mentioned above, suffix arrays can be generalized in a straightforward

¹ We used Doug McIlroy's implementation: <<http://cm.bell-labs.com/cm/cs/who/doug/ssort.c>>.

way to work with larger alphabets such as Japanese (typically two bytes per character). In Section 3, we use the open-ended set of English words as the alphabet. Each token (English word) is represented as a 4-byte pointer into a symbol table (dictionary). The corpus, “to_be_or_not_to_be,” for example, is tokenized into 6 tokens: “to”, “be”, “or”, “not”, “to”, and “be”, where each token is represented as a 4-byte pointer into a dictionary.

2.2 Longest Common Prefixes (LCPs)

The suffix array literature makes use of an auxiliary array for storing LCPs (longest common prefixes). The *lcp* array contains $N + 1$ integers. Each element, $lcp[i]$, indicates the length of the common prefix between $s[i - 1]$ and $s[i]$. We pad the *lcp* vector with zeros ($lcp[0] = lcp[N] = 0$) to simplify the code and definitions for some concepts in the next section. The padding avoids the need to test for certain end conditions.

Figure 3 shows the *lcp* vector for the suffix array of “to_be_or_not_to_be.” For example, since $s[10]$ and $s[11]$ both start with the substring, “o_be,” $lcp[11]$ is set to 4, the length of the longest common prefix. Manber and Myers (1990) use the *lcp* vector in their $O(P + \log N)$ algorithm for computing the frequency and location of a substring of length P in a sequence of length N . They showed that the *lcp* vector can be computed in $O(N \log N)$ time. These algorithms are much faster than the obvious straightforward implementation when the corpus contains long repeated substrings, though for many corpora, the complications required to avoid quadratic behavior are unnecessary.

2.3 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? As mentioned above, the $N(N + 1)/2$ substrings will be clustered into a relatively small number of classes, and then the statistics will be computed over the classes rather than over the substrings, which would have been prohibitive. The reduction

Suffix Array: Suffix denoted by $s[i]$		Lcp vector		
$s[0]$	15	be	$lcp[0]$	0 ← always 0
$s[1]$	2	be_or_not_to_be	$lcp[1]$	3
$s[2]$	8	not_to_be	$lcp[2]$	1
$s[3]$	5	or_not_to_be	$lcp[3]$	1
$s[4]$	12	to_be	$lcp[4]$	1
$s[5]$	16	be	$lcp[5]$	0
$s[6]$	3	be_or_not_to_be	$lcp[6]$	2
$s[7]$	17	e	$lcp[7]$	0
$s[8]$	4	e_or_not_to_be	$lcp[8]$	1
$s[9]$	9	not_to_be	$lcp[9]$	0
$s[10]$	14	o_be	$lcp[10]$	0
$s[11]$	1	o_be_or_not_to_be	$lcp[11]$	4
$s[12]$	6	or_not_to_be	$lcp[12]$	1
$s[13]$	10	ot_to_be	$lcp[13]$	1
$s[14]$	7	r_not_to_be	$lcp[14]$	0
$s[15]$	11	t_to_be	$lcp[15]$	0
$s[16]$	13	to_be	$lcp[16]$	1
$s[17]$	0	to_be_or_not_to_be	$lcp[17]$	5
			$lcp[18]$	0 ← always 0

The dotted lines denote lcp's.

Figure 3

The Longest Common Prefix is a vector of $N + 1$ integers. $lcp[i]$ denotes the length of the common prefix between the suffix $s[i - 1]$ and the suffix $s[i]$. Thus, for example, $s[10]$ and $s[11]$ share a common prefix of 4 characters, and therefore $lcp[11] = 4$. The common prefix is highlighted by a dotted line in the suffix array. The suffix array is the same as in the previous figure.

of the computation over substrings to a computation over classes is made possible by four properties. The substrings in a class all share the same tf (property 1) and the same df (property 2). Moreover, every substring is a member of some class (property 3b). Finally, the number of classes is manageable, much smaller than the number of substrings (property 4).

Classes are defined in terms of intervals. Let $\langle i, j \rangle$ be an interval on the suffix array, $s[i], s[i + 1], \dots, s[j]$. Classes are constructed so that the substrings in a class are equivalent, at least with respect to tf (property 1) and df (property 2). $Class(\langle i, j \rangle)$ is the set of substrings that start every suffix within the interval and no substring outside the interval. Property 1 immediately follows from this construction. That is, every substring

in $class(\langle i, j \rangle)$ has $tf = j - i + 1$, and consequently, if there are two substrings in a class, they would both share the same term frequency.

The set of substrings in a class can be constructed from the lcp vector:

$$class(\langle i, j \rangle) = \{s[i]_m \mid \max(lcp[i], lcp[j + 1]) < m \leq \min(lcp[i + 1], lcp[i + 2], \dots, lcp[j])\},$$

where $s[i]_m$ denotes the first m characters of the suffix $s[i]$. We will refer to $lcp[i]$ and $lcp[j + 1]$ as bounding lcps and $lcp[i + 1], lcp[i + 2], \dots, lcp[j]$ as interior lcps. The equation above can be rewritten as

$$class(\langle i, j \rangle) = \{s[i]_m \mid LBL(\langle i, j \rangle) < m \leq SIL(\langle i, j \rangle)\},$$

where LBL (longest bounding lcp) is

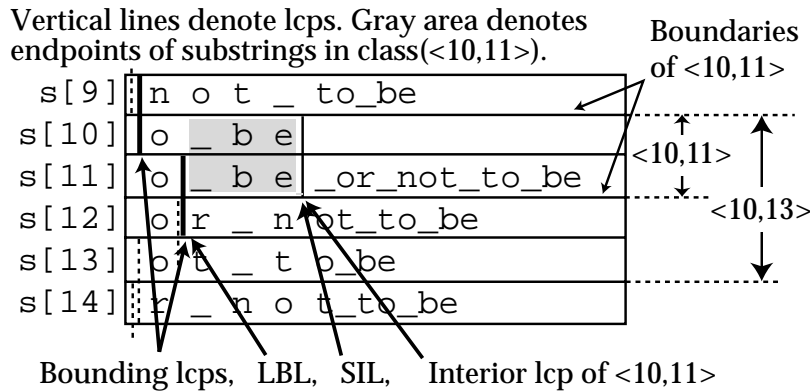
$$LBL(\langle i, j \rangle) = \max(lcp[i], lcp[j + 1]),$$

and SIL (shortest interior lcp) is

$$SIL(\langle i, j \rangle) = \min(lcp[i + 1], lcp[i + 2], \dots, lcp[j]).$$

By construction, the class will be empty unless there is some room between the LBL and SIL. We say that an interval is *lcp-delimited* when $LBL < SIL$. Except for trivial intervals (see below), Classes are non-empty iff the interval is lcp-delimited and the number of substrings in a class depends on the gap between the LBL and the SIL. That is, $|class(\langle i, j \rangle)| = SIL(\langle i, j \rangle) - LBL(\langle i, j \rangle)$.

Figure 4 shows eight examples of lcp-delimited intervals. The top part of the figure highlights the interval, $\langle 10, 11 \rangle$. The solid vertical lines denote the bounding lcp's, and the thin vertical lines denote the interior lcp's. The interval $\langle 10, 11 \rangle$ is lcp-delimited because the bounding *lcps*, $lcp[10] = 0$ and $lcp[12] = 1$, are smaller than the interior *lcp*, $lcp[11] = 4$. That is, the LBL (=1) is less than the SIL (=4). Thus there is room for m between the LBL and the SIL, and the class is non-empty: $class(\langle 10, 11 \rangle) = \{s[10]_m \mid 1 <$



LCP-delimited interval	Class	LBL	SIL	tf
$\langle 10,11 \rangle$	{"o_", "o_b", "o_be"}	1	4	2
$\langle 10,13 \rangle$	{"o"}	0	1	4
$\langle 9,9 \rangle$	{"n", "no", "not", ...}	0	infinity	1
$\langle 10,10 \rangle$	{}	4	infinity	1
$\langle 11,11 \rangle$	{"o_be_", "o_be_o", ...}	4	infinity	1
$\langle 12,12 \rangle$	{"or", "or_", "or_n", ...}	1	infinity	1
$\langle 13,13 \rangle$	{"ot", "ot_", "ot_t", ...}	1	infinity	1
$\langle 14,14 \rangle$	{"r", "r_", "r_n", ...}	0	infinity	1

Figure 4
 Six suffixes in Figure 3, $s[9]$ - $s[14]$, are shown again with the lcp's. Two non-trivial lcp-delimited intervals and six trivial lcp-delimited intervals are shown along with their longest bounding LCP (LBL) and shortest interior LCP (SIL). Intervals are associated with sets of strings, which we call classes. These substrings are common prefixes of all of the suffixes within the interval, and by no suffix outside the interval. All of the strings in a class have the same term frequency.

$m \leq 4\} = \{“o_”, “o_b”, “o_be”\}$. The endpoints m between LBL and SIL are highlighted in gray. Every suffix within the interval $\langle 10, 11 \rangle$ starts with all of these substrings and no suffix outside $\langle 10, 11 \rangle$ starts with any of these substrings. In particular, the substring “o” is excluded from the class, because it is shared by suffixes outside the interval, namely $s[12]$ and $s[13]$. The longer substring, “o.be_,” is excluded from the class because it is not shared by $s[10]$, a suffix within the interval.

We call an interval *trivial* if the interval starts and ends at the same place: $\langle i, i \rangle$. Trivial intervals typically have quite large classes. After all, there are only N trivial intervals but they cover all (and only) the substrings with $tf = 1$, typically most of the $N(N + 1)/2$ substrings in a corpus. Therefore, trivial classes have to be quite large

because there are only N such classes but they account for order N^2 substrings. We define the class of a trivial interval, $class(\langle i, i \rangle)$, to be $\{s[i]_m | LBL < m\}$. The SIL of a trivial interval is defined to be infinite. Note that there are empty classes of trivial intervals such as $\langle 10, 10 \rangle$ in Figure 4. The LBL of $\langle 10, 10 \rangle$ plus the position of the suffix, $s[10]$, is the end of the corpus.

It is possible for lcp-delimited intervals to be nested, as in the case of $\langle 10, 11 \rangle$ and $\langle 10, 13 \rangle$. We say that one interval $\langle i, j \rangle$ is nested within another $\langle u, v \rangle$ if $i \leq u \leq v \leq j$ (and $\langle i, j \rangle \neq \langle u, v \rangle$). Nested intervals have disjoint classes and different SILs.² The substrings in the class of the nested interval are longer than the substrings in the class of the outer interval.

Although it is possible for lcp-delimited intervals to be nested, it is not possible for lcp-delimited intervals to overlap. We say that one non-trivial interval $\langle a, b \rangle$ overlaps another non-trivial interval $\langle c, d \rangle$ if $a < c \leq b < d$. If two intervals overlap, then at least one of the intervals is not lcp-delimited and has an empty class. The fact that lcp-delimited intervals are nested and do not overlap will turn out to be convenient for enumerating lcp-delimited intervals.

The remaining six intervals mentioned in Figure 4 are trivial intervals. That is, they all start and end at the same place. As mentioned above, trivial intervals tend to have large classes. They contain all substrings $s[i]_m$ such that $m > LBL$. All of these substrings have a term frequency of 1.

Not every interval is lcp-delimited. The interval, $\langle 11, 12 \rangle$, for example, is not lcp-delimited because there is no room between the LBL (=4) and the SIL (=1). When the interval is not lcp-delimited, the class is empty. There are no substrings starting all the suffixes within the interval $\langle 11, 12 \rangle$, and not starting any suffix outside the interval.

² Because $\langle u, v \rangle$ is lcp-delimited, there must be a bounding lcp of $\langle u, v \rangle$ that is smaller than any lcp within $\langle u, v \rangle$. This bounding lcp must be within $\langle i, j \rangle$, and as a result, $class(\langle i, j \rangle)$ and $class(\langle u, v \rangle)$ must be disjoint.

2.4 Four Properties

As mentioned above, classes are constructed so that it is practical to reduce the computation of various statistics over substrings to a computation over classes. This subsection will discuss four properties of classes that help make this reduction feasible.

The first two properties are convenient because they allow us to associate tf and df with classes rather than with substrings. The substrings in a class all have the same tf value (property 1) and the same df value (property 2). That is, if s_1 and s_2 are two substrings in $class(\langle i, j \rangle)$ then

Property 1: $tf(s_1) = tf(s_2) = j - i + 1$

Property 2: $df(s_1) = df(s_2)$.

Both of these properties follow straightforwardly from the construction of intervals. The value of tf is a simple function of the endpoints; the calculation of df is more complicated and will be discussed in section 2.6. While tf and df treat each member of a class as equivalent, not all statistics do. Mutual information (MI) is an important counter example; in most cases, $MI(s_1) \neq MI(s_2)$.

The third property is convenient because it allows us to iterate over classes rather than substrings, without worrying about missing any of the substrings.

Property 3: The classes partition the set of all substrings.

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 distinct classes: $class(\langle i, j \rangle)$ and $class(\langle u, v \rangle)$. There are three possibilities: one interval precedes the other, they are properly nested or they overlap. In all three cases, s cannot be a member of both classes. If one interval precedes

the other, then there must be a bounding lcp between the two intervals which is shorter than s . And therefore, s cannot be in both classes. The nesting case was mentioned previously where it was noted that nested intervals have disjoint classes. The overlapping case was also discussed previously where it was noted that two overlapping intervals cannot both be lcp-delimited, and therefore at least one of the classes would have to be empty.

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 $\langle i, j \rangle$ is lcp-delimited ($LBL(\langle i, j \rangle) < |s|$ and $SIL(\langle i, j \rangle) \geq |s|$), and therefore, s is an element of $class(\langle i, j \rangle)$.

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 $2N - 1$) 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 trivial intervals $\langle i, i \rangle$. These are all and only the intervals with $tf = 1$. By construction, these intervals are lcp-delimited.

To argue the second clause, we make use of a uniqueness property: an lcp-delimited interval $\langle i, j \rangle$ can be uniquely determined by its SIL and a representative element k , where $i < k \leq j$. Suppose there were two distinct intervals, $\langle i, j \rangle$ and $\langle u, v \rangle$, with the same SIL, $SIL(\langle i, j \rangle) = SIL(\langle u, v \rangle)$, and the same representative k , where $i < k \leq j$ and $u < k \leq v$. Since they share a common representative, k , one interval must be nested inside the other. But nested intervals have disjoint classes and different SILs.

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 $\langle i, j \rangle$ 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 cannot 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 does not count trivial intervals because their SILs (= infinity) are not in the lcp vector; the lcp vector contains integers less than N .)

From property 4, it follows that there are at most N distinct values of RIDF. The N trivial intervals $\langle i, i \rangle$ have just one RIDF value since $tf = df = 1$ for these intervals. The other $N - 1$ intervals could have as many as another $N - 1$ RIDF values. Similar arguments hold for many other statistics that make use of tf and df , and treat all members of a class as equivalent.

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.

2.5 Computing all classes using suffix arrays

This subsection describes a single-pass procedure, `print_LDIs`, for computing tf for all LDIs (lcp-delimited intervals). Since lcp-delimited intervals are properly nested, the procedure is based on a push-down stack. The procedure outputs four quantities for each lcp-delimited interval, $\langle i, j \rangle$. The four quantities are the two endpoints (i and j), the term frequency (tf) and a representative (k), such that $i < k \leq j$ and $lcp[k] = SIL(\langle i, j \rangle)$. This procedure will be described twice. The first implementation is expressed in a recursive form; the second implementation avoids recursion by implementing the stack explicitly. C code is provided in the appendices.

The recursive implementation is presented first, because it is simpler. The function

`print_LDI`s is initially called with `print_LDI`s(0,0), which will cause the function to be called once for each value of k between 0 and $N - 1$. k is a representative in the range: $i < k \leq j$, where i and j are the endpoints of an interval. For each of the N values of k , a trivial LDI is reported at $\langle k, k \rangle$. In addition, there could be up to $N - 1$ non-trivial intervals, where k is the representative and $lcp[k]$ is the SIL. Recall that lcp-delimited intervals are uniquely determined by a representative k such that $i < k \leq j$ and a SIL. Not all of these candidates will produce LDI. The recursion searches for j 's such that $LBL(\langle i, j \rangle) \leq SIL(\langle i, j \rangle)$, but reports intervals at $\langle i, j \rangle$ only when the inequality is a strict inequality, that is, $LBL(\langle i, j \rangle) < SIL(\langle i, j \rangle)$. The program stack keeps track of the left and right edges of these intervals. While $lcp[k]$ is monotonically increasing, the left edge is remembered on the stack, as `print_LDI`s is called recursively. The recursion unwinds as $lcp[j] < lcp[k]$. Figure 5 illustrates the flow.

```

print_LDI ← function( $i, k$ ) {
     $j \leftarrow k$ .
    Output a trivial lcp-delimited interval  $\langle k, k \rangle$  with  $tf = 1$ .
    While  $lcp[k] \leq lcp[j + 1]$  and  $j + 1 < N$ , do  $j \leftarrow \text{print\_LDI}(k, j + 1)$ .
    Output an interval  $\langle i, j \rangle$  with  $tf = j - i + 1$  and  $\text{rep} = k$ , if it is lcp-delimited.
    Return  $j$ . }

```

The second implementation (below) introduces its own explicit stack, a complication that turns out to be important in practice, especially for large corpora. C code is provided in the appendices.

```

print_LDI_stack ← function( $N$ ){
     $\text{stack}_i \leftarrow$  an integer array for the stack of the left edges,  $i$ .
     $\text{stack}_k \leftarrow$  an integer array for the stack of the representatives,  $k$ .
     $\text{stack}_i[0] \leftarrow 0$ .

```

Suffix Array

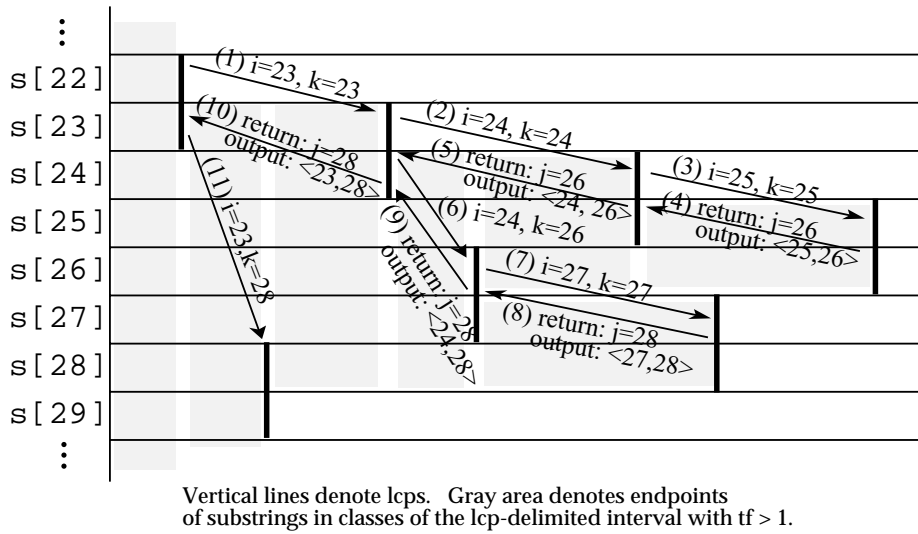


Figure 5
Argument and returned values of `print_LDIs`.

$stack_k[0] \leftarrow 0.$

$sp \leftarrow 1$ (a stack pointer).

For $j \leftarrow 0, 1, 2, \dots, N - 1$ do

Output an lcp-delimited interval $\langle j, j \rangle$ with $tf = 1$.

While $lcp[j + 1] < lcp[stack_k[sp - 1]]$ do

Output an interval $\langle i, j \rangle$ with $tf = j - i + 1$, if it is lcp-delimited.

$sp \leftarrow sp - 1.$

$stack_i[sp] \leftarrow stack_k[sp - 1].$

$stack_k[sp] \leftarrow j + 1.$

$sp \leftarrow sp + 1. \}$

2.6 Computing df for all classes

Thus far we have seen how to compute term frequency for all substrings (ngrams) in a sequence (corpus). This section will extend the solution to compute document frequency, df , as well as term frequency, tf . The solution runs in $O(N \log N)$ time and $O(N)$ space.

C code is provided in appendix C.

This section will use the running example shown in Figure 6, where the corpus is: “to_be\$or\$not_to_be\$”. The corpus consists of three documents, “to_be\$”, “or\$” and “not_to_be\$”. The special character ‘\$’ is used to denote the end of a document. The procedure outputs a sequence of intervals with their term frequencies and document frequencies. These results are also presented for the non-trivial intervals.

The suffix array is computed using the same procedures discussed above. In addition to the suffix array and the lcp vector, Figure 6 introduces a new third table that is used to map from suffixes to document ids. This table of document id’s will be used by the function *get_docnum* to map from suffixes to document ids. Suffixes are terminated in Figure 6 after the first end of document symbol, unlike before, where suffixes were terminated with the end of corpus symbol.

A straightforward method for computing *df* for an interval is to enumerate the suffixes within the interval and then compute their document ids, remove duplicates and return the number of distinct documents. Thus, for example, *df*(“o”) in Figure 6, can be computed by finding corresponding interval, $\langle 11, 14 \rangle$, where every suffix within the interval starts with “o” and no suffix outside the interval starts with “o”. Then we enumerate the suffixes within the interval $\{s[11], s[12], s[13], s[14]\}$ and compute their document ids, $\{0, 2, 1, 2\}$, and remove duplicates. In the end we discover that *df*(“o”) = 3. That is, “o” appears in all three documents.

Unfortunately, this straightforward approach is almost certainly too slow. Some document ids will be computed multiple times, especially when suffixes appear in nested intervals. We take advantage of the nesting property of lcp-delimited intervals to compute all *df*’s efficiently. The *df* of an lcp-delimited interval can be computed recursively in terms of its constituents (nested subintervals), thus avoiding unnecessary recomputation.

The procedure `print_LDIIs_with_df` presented below is similar to `print_LDIIs_stack`

Suffix Array	Suffixes denoted		Document id's	
	by $s[i]$	$lcp[i]$	of $s[i]$	
s[0]	2	_be\$	0	0
s[1]	15	_be\$	4	2
s[2]	12	_to_be\$	1	2
s[3]	5	\$	0	0
s[4]	8	\$	1	1
s[5]	18	\$	1	2
s[6]	3	be\$	0	0
s[7]	16	be\$	3	2
s[8]	4	e\$	0	0
s[9]	17	e\$	2	2
s[10]	9	not_to_be\$	0	2
s[11]	1	o_be\$	0	0
s[12]	14	o_be\$	5	2
s[13]	6	or\$	1	1
s[14]	10	ot_to_be\$	1	2
s[15]	7	r\$	0	1
s[16]	11	t_to_be\$	0	2
s[17]	0	to_be\$	1	0
s[18]	13	to_be\$	6	2
			0	0

Input documents: $d_0 = \text{"to_be\$"}$
 $d_1 = \text{"or\$"}$
 $d_2 = \text{"not_to_be\$"}$

Corpus = $d_0 + d_1 + d_2 = \text{"to_be\$or\$not_to_be\$"}$

Resulting non-trivial lcp-delimited intervals:

- $\langle 0, 1 \rangle$, rep= 1, tf=2, df=2
- $\langle 0, 2 \rangle$, rep= 2, tf=3, df=2
- $\langle 3, 5 \rangle$, rep= 4, tf=3, df=3
- $\langle 6, 7 \rangle$, rep= 7, tf=2, df=2
- $\langle 8, 9 \rangle$, rep= 9, tf=2, df=2
- $\langle 11, 12 \rangle$, rep=12, tf=2, df=2
- $\langle 11, 14 \rangle$, rep=13, tf=4, df=3
- $\langle 17, 18 \rangle$, rep=18, tf=2, df=2
- $\langle 16, 18 \rangle$, rep=17, tf=3, df=2

Figure 6

A suffix array for a corpus consisting of three documents. The special character '\$' denotes the end of a document. The procedure outputs a sequence of intervals with their term frequencies and document frequencies. These results are also presented for the non-trivial intervals.

but modified to compute df as well as tf . The stack keeps track of i and k , as before, but now the stack also keeps track of df .

i , the left edge of an interval,

k , the representative ($SIL = lcp[k]$),

df , partial results for df , counting documents seen thus far, minus

duplicates.

```

print_LDIs_with_df ← function( $N$ ){
     $stack\_i$  ← an integer array for the stack of the left edges,  $i$ .
     $stack\_k$  ← an integer array for the stack of the representatives,  $k$ .
     $stack\_df$  ← an integer array for the stack of the df counter.
     $doclink[0..D]$  : an integer array for the document link initialized with -1.
        D = the number of documents.
     $stack\_i[0]$  ← 0.
     $stack\_k[0]$  ← 0.
     $stack\_df[0]$  ← 1.
     $sp$  ← 1 (a stack pointer).
    (1) For  $j$  ← 0, 1, 2, ...,  $N - 1$  do
    (2)     (Output a trivial lcp-delimited interval  $\langle j, j \rangle$  with  $tf = 1$  and  $df = 1$ .)
    (3)      $doc$  ← get_docnum( $s[j]$ )
    (4)     if  $doclink[doc] \neq -1$ , do
    (5)         let  $x$  be the largest  $x$  such that  $doclink[doc] \geq stack\_i[x]$ .
    (6)          $stack\_df[x] \leftarrow stack\_df[x] - 1$ .
    (7)      $doclink[doc] \leftarrow j$ .
    (8)      $df \leftarrow 1$ .
    (9)     While  $lcp[j + 1] < lcp[stack\_k[sp - 1]]$  do
    (10)          $df \leftarrow stack\_df[sp - 1] + df$ .
    (11)         Output a non-trivial interval  $\langle i, j \rangle$  with  $tf = j - i + 1$  and  $df$ ,
                if it is lcp-delimited.
    (12)          $sp \leftarrow sp - 1$ .
    (13)      $stack\_i[sp] \leftarrow stack\_k[sp - 1]$ .

```

(14) $stack_k[sp] \leftarrow j + 1.$

(15) $stack_df[sp] \leftarrow df.$

(16) $sp \leftarrow sp + 1. \}$

Lines 5-6 take care of duplicate documents. The duplication processing makes use of *doclink*, an array of length D , the number of documents in the collection, that keeps track of which suffixes have been seen in which document. *doclink* is initialized with -1 indicating that no suffixes have been seen yet. As suffixes are processed, *doclink* is updated (on line 7) so that *doclink*[d] contains the most recently processed suffix in document d . As illustrated in Figure 7, when $j = 16$ (snapshot A), the most recently processed suffix in document 0 is $s[11]$ (“o_be\$”), the most recently processed suffix in document 1 is $s[15]$ (“r\$”), and the most recently processed suffix in document 2 is $s[16]$ (“t_to_be\$”). Thus, *doclink*[0]=11, *doclink*[1]=15 and *doclink*[2]=16. After processing $s[17]$ (“to_be\$”), which is in document 0, *doclink*[0] is updated from 11 to 17, as shown in snapshot B of Figure 7.

Stack_df keeps track of document frequencies as suffixes are processed. The invariant is: *stack_df*[x] contains the document frequency for suffixes seen thus far starting at $i = stack_i[x]$. (x is a stack offset.) When a new suffix is processed, line 5 checks for double counting by searching for intervals on the stack (still being processed) that have suffixes in the same document as the current suffix. If there is any double counting, *stack_df* is decremented appropriately on line 6.

There is an example of this decrementing in snapshot C of Figure 7, highlighted by the circle around the binding of *df* to 0 on the stack element: [$i=0, k=17, df=0$]. Note that *df* was previously bound to 1 in snapshot B. The binding of *df* was decremented when processing $s[18]$ because $s[18]$ is in the same document as $s[16]$. This duplication was identified by line 5. The decrementing was performed by line 6.

Intervals are processed in depth first order so that more deeply nested intervals are processed before less deeply nested intervals. In this way, double counting is only an issue for intervals higher on the stack. The most deeply nested intervals are trivial intervals. They are processed first. They have a df of 1 (line 8). For the remaining non-trivial intervals, $stack_df$ contains the partial results for intervals in process. As the stack is popped, the df values are aggregated up to compute the df value for the outer intervals. The aggregation occurs on line 10 and the popping of the stack occurs on line 12. The aggregation step is illustrated in snapshots C and D of Figure 7 by the two arrows with the ‘+’ combination symbol pointing at a value of df in an output statement.

2.7 Class arrays

The classes identified by the previous calculation are stored in a data structure we call a *class array*, to make it relatively easy to look up the term frequency and document frequency for an arbitrary substring. The class array is a stored list of 5-tuples: $\langle SIL, LBL, tf, df, longest\ suffix \rangle$. The fifth element of the 5-tuple is a canonical member of the class (the longest suffix). The 5-tuples are sorted by the the alphabetical order of the canonical members. In our C code implementation, classes are represented by 5 integers, one for each element in the 5-tuple. Since there are N trivial classes and at most $N - 1$ non-trivial classes, the class array will require at most $10N - 5$ integers. However, for many practical applications, the trivial classes can be omitted.

Figure 8 shows an example of the non-trivial class array for the corpus: “to_be\$or\$not_to_be\$”. The class array makes it relatively easy to determine that the substring “o” appears in all three documents. That is, $df(\text{“o”}) = 3$. We use a binary search to find that tuple $c[5]$ is the relevant 5-tuple for “o”. Having found the relevant tuple, it requires a simple record access to return the document frequency field.

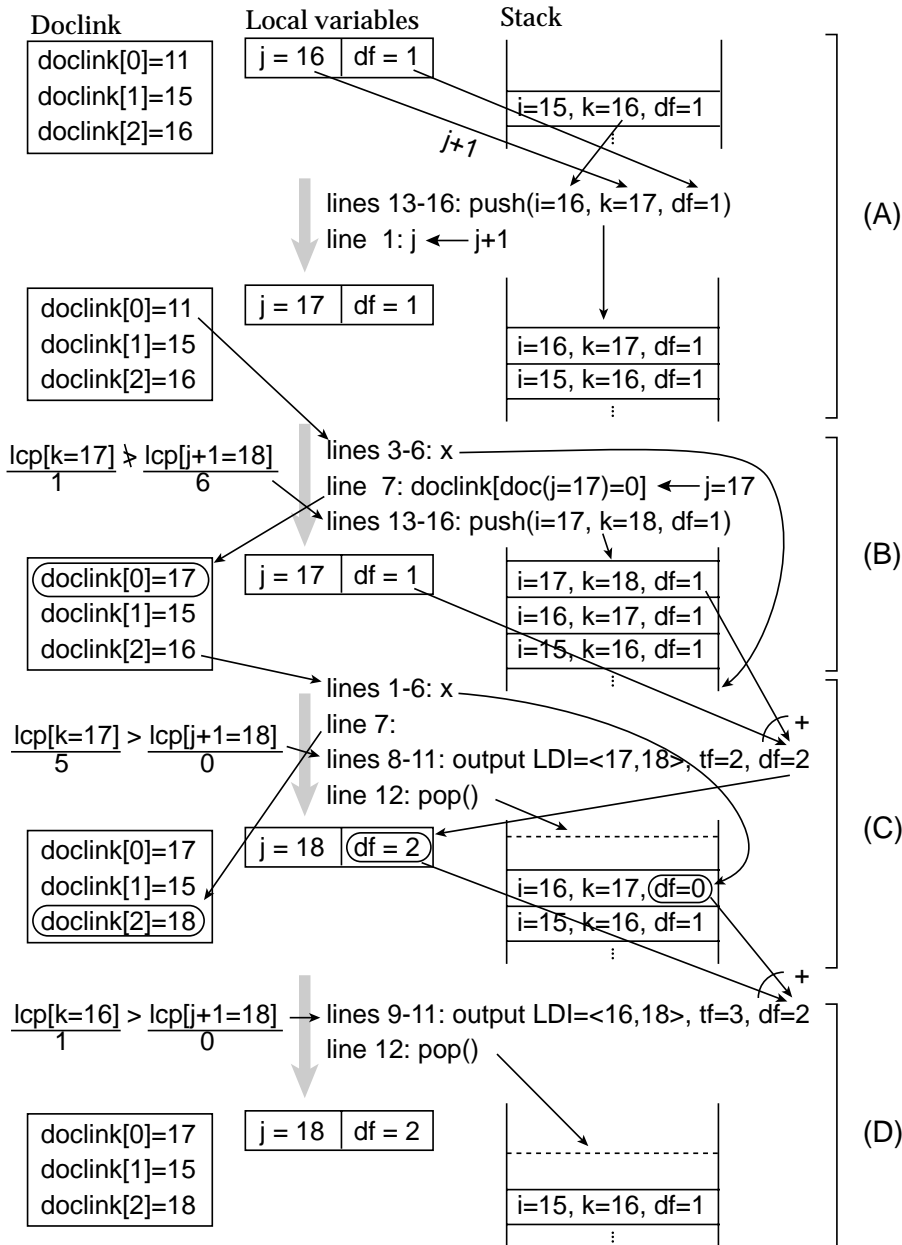


Figure 7 Snapshots of the `doclink` array and the stack during the processing of `print_LDIswith_df` on the corpus: “to_be\$or\$not_to_be\$”. The four snapshots A-D illustrate the state as `j` progresses from 16 to 18. Two non-trivial intervals are emitted while `j` is in this range: `(17, 18)` and `(16, 18)`. The more deeply nested interval is emitted before the less deeply nested interval.

Class array (Pointer to corpus)		The longest suffix denoted by c[i]				
		SIL	LBL	tf	df	
c[0]	0	_	1	0	3	2
c[1]	0	_be\$	4	1	2	2
c[2]	5	\$	1	0	3	3
c[3]	3	be\$	3	0	2	2
c[4]	4	e\$	2	0	2	2
c[5]	1	o	1	0	4	3
c[6]	1	o_be\$	5	1	2	2
c[7]	11	t	1	0	3	2
c[8]	0	to_be\$	6	1	2	2

Figure 8
An example of the class array for the corpus: “to_be\$or\$not_to_be\$”.

Table 1
Statistics of the English and Japanese corpora.

Statistic	Wall Street Journal	Mainichi Shimbun
N (corpus size in tokens)	49,810,697 words	215,789,699 characters
V (vocabulary in types)	410,957	5509
# articles	112,915	435,859
# non-trivial classes	16,519,064	82,442,441
# substrings in non-trivial classes	2,548,140,677	1,388,049,631
substrings per class (in non-trivial classes)	154.3	16.8

3 Experimental results

3.1 RIDF and MI for English and Japanese

We used the methods described above to compute df , tf and RIDF for all substrings in two corpora of newspapers summarized in Table 1. MI was computed for the longest substring in each class. The entire computation took a few hours. The processing time was dominated by the calculation of the suffix array.

The English collection consists of 50 million words (113 thousand articles) of Wall Street Journal (distributed by the ACL/DCI) and the Japanese collection consists of 216 million characters (436 thousand articles) of CD-Mainichi Shimbun from 1991-1995 (which are distributed in CD-ROM format). The English corpus was tokenized into words delimited by white space, whereas the Japanese corpus was tokenized into characters (typically 2-bytes each).

Table 1 indicates that there are a large number of non-trivial classes in both corpora. The English corpus has more substrings per non-trivial class than the Japanese corpus. It has been noted elsewhere that the English corpus contains quite a few duplicated articles (Paul and Baker, 1992). The duplicated articles could explain why there are so many substrings per non-trivial class in the English corpus when compared with the Japanese corpus.

For subsequent processing, we excluded substrings with $tf < 10$ to avoid noise,

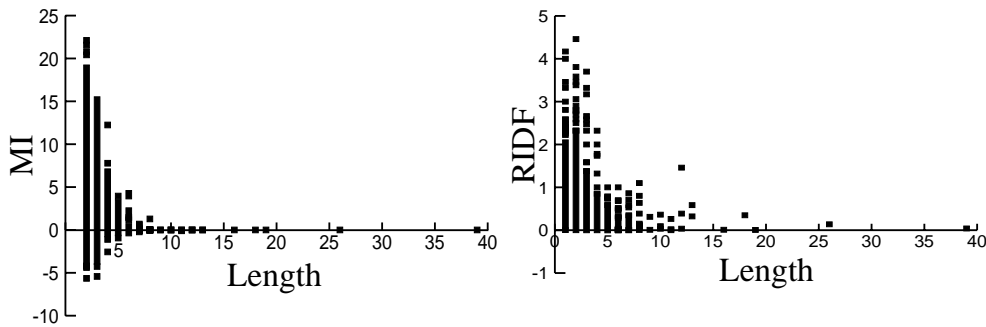


Figure 9

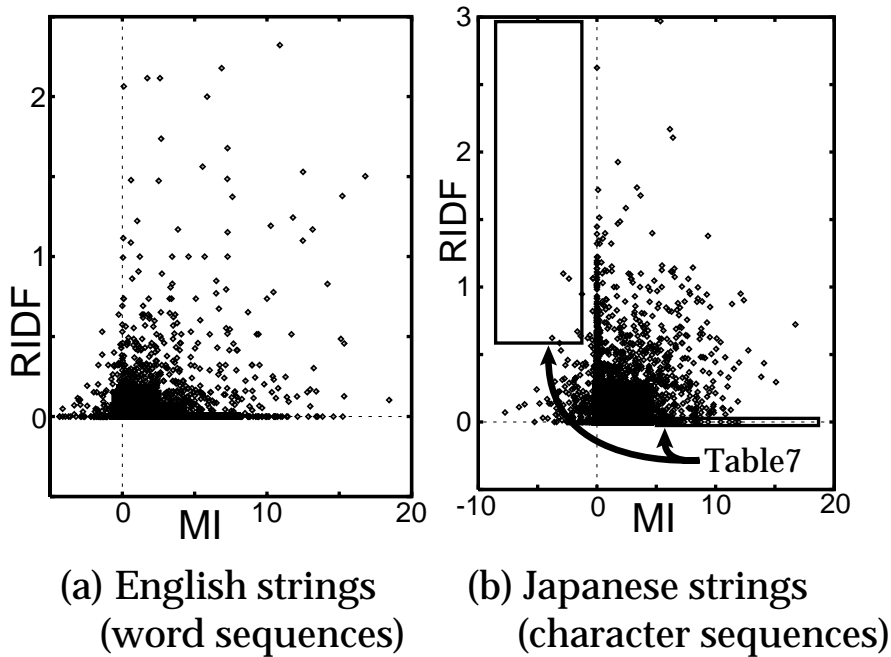
The left panel plots MI as a function of the length of the ngram; the right panel plots RIDF as a function of the length of the ngram. Both panels were computed from the Japanese corpus. Note that while there is more dynamic range for shorter ngrams than for longer ngrams, there is plenty of dynamic range for ngrams well beyond bigrams and trigrams.

resulting in about 1.4 million classes (1.6 million substrings) for English and 10 million classes (15 million substrings) for Japanese. We computed RIDF and MI values for the longest substring in each of these 1.4 million English classes and 10 million Japanese classes. These values can be applied to the other substrings in these classes for RIDF, but not for MI. (As mentioned above, two substrings in the same class need not have the same MI value.)

Figure 9 plots RIDF and MI values of random samples as a function of string length. In both cases, shorter substrings have more dynamic range. That is, RIDF and MI vary more for bigrams than million-grams. But there is considerable dynamic range for ngrams well beyond bigrams and trigrams.

3.2 Little correlation between RIDF and MI

We are interested in comparing and contrasting RIDF and MI. Figure 10 shows that RIDF and MI are largely independent. There is little if any correlation between the RIDF of a string and MI of the same string. Panel (a) compares RIDF and MI for a sample of English word sequences from the WSJ corpus (excluding unigrams); panel (b) makes the same comparison but for Japanese phrases identified as keywords on the CD-ROM. In both cases, there are many substrings with a large RIDF value and a small MI,

**Figure 10**

Both panels plot RIDF versus MI. Panel (a) plots RIDF and MI for a sample of English ngrams; panel (b) plots RIDF and MI for Japanese phrases identified as keywords on the CD-ROM. The right panel highlights the 10% highest RIDF and 10% lowest MI with a box, as well as the 10% lowest RIDF and the 10% highest MI. The arrows pointing to the box add extra emphases.

and vice versa.

We believe the two statistics are both useful but in different ways. Both pick out interesting ngrams, but ngrams with large MI are interesting in different ways from ngrams with large RIDF. Consider the English word sequences in Table 2, which all contain the word /having/. These sequences have large MI values and small RIDF values. In our collaboration with lexicographers, especially those working on dictionaries for learners, we have found considerable interest in statistics such as MI that pick out these kinds of phrases. Collocations can be quite challenging for non-native speakers of the language. On the other hand, these kinds of phrases are not very good keywords for information retrieval.

Table 3 shows MI and RIDF values for a random sample of word sequences containing

Table 2

English phrases containing the word “having”. Note that these phrases have large MI and low RIDF. They tend to be more interesting for lexicography than information retrieval. The table is sorted by MI.

tf	df	RIDF	MI	Phrase
18	18	-0.00	10.5	admits to having
14	14	-0.00	9.7	admit to having
25	23	0.12	8.9	diagnosed as having
20	20	-0.00	7.4	suspected of having
301	293	0.04	7.3	without having
15	13	0.21	7.0	denies having
59	59	-0.00	6.8	avoid having
18	18	-0.00	6.0	without ever having
12	12	-0.00	5.9	Besides having
26	26	-0.00	5.8	denied having

Table 3

English phrases containing the word “Mr.” (sorted by RIDF). The word sequences near the top of the list are better keywords than the sequences near the bottom of the list. None of them are of much interest to lexicography.

tf	df	RIDF	MI	Phrase
11	3	1.9	0.6	. Mr. Hinz
18	5	1.8	6.6	Mr. Bradbury
51	16	1.7	6.7	Mr. Roemer
67	25	1.4	6.8	Mr. Melamed
54	27	1.0	5.8	Mr. Burnett
11	8	0.5	1.1	Mr. Eiszner said
53	40	0.4	0.3	Mr. Johnson .
21	16	0.4	0.2	Mr. Nichols said .
13	10	0.4	0.4	. Mr. Shulman
176	138	0.3	0.5	Mr. Bush has
13	11	0.2	1.5	to Mr. Trump’s
13	11	0.2	-0.9	Mr. Bowman ,
35	32	0.1	1.2	wrote Mr.
12	11	0.1	1.7	Mr. Lee to
22	21	0.1	1.4	facing Mr.
11	11	-0.0	0.7	Mr. Poehl also
13	13	-0.0	1.4	inadequate . ” Mr.
16	16	-0.0	1.6	The 41-year-old Mr.
19	19	-0.0	0.5	14 . Mr.
26	26	-0.0	0.0	in November . Mr.
27	27	-0.0	-0.0	” For his part , Mr.
38	38	-0.0	1.4	. AMR ,
39	39	-0.0	-0.3	for instance , Mr.

the word “Mr.” The table is sorted by RIDF. The sequences near the top of the list are better keywords than the sequences further down. None of these sequences would be of much interest to a lexicographer (unless he/she were studying names). Many of the sequences have rather small MI values.

Table 4 shows a few word sequences starting with the word “the” with large MI values. All of these sequences have high MI (by construction), but some are high in RIDF as well (labeled B), and some are not (labeled A). Most of the sequences are interesting in one way or another, but the A sequences are different from the B sequences. The A sequences would be of more interest to someone studying the grammar in the WSJ subdomain, whereas the B sequences would be of more interest to someone studying the terminology in this subdomain. The B sequences in Table 4 tend to pick out specific events in the news, if not specific stories. The phrase, “the Basic Law,” for example,

Table 4

Selected English phrases containing “the.” All of these phrases have high MI. Some have high RIDF, and some do not.

(A): Low RIDF (poor keywords)

tf	df	RIDF	MI	Phrase
11	11	-0.0	11.1	the up side
73	66	0.1	9.3	the will of
16	16	-0.0	8.6	the sell side
17	16	0.1	8.5	the Stock Exchange of
16	15	0.1	8.5	the buy side
20	20	-0.0	8.4	the down side
55	54	0.0	8.3	the will to
14	14	-0.0	8.1	the saying goes
15	15	-0.0	7.6	the going gets

(B): High RIDF (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 5

Concordance of the phrase “the Basic Law”. Note that most of the instances of “the Basic Law” appear in just two stories, as indicated by the doc-id (the token-id of the first word in the document).

token-id	left context	right context	doc-id
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 .	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 .	39020101

picks out a pair of stories that discuss the event of the handover of Hong Kong to China, as illustrated in the concordance shown in Table 5.

Table 6 shows a number of word sequences with high MI containing common prepositions. The high MI indicates an interesting association, but again most have low RIDF and are not particularly good keywords, though there are a few exceptions (“Just for Men,” a well-known brand name, has a high RIDF and is a good keyword).

The Japanese substrings are similar to the English substrings. Substrings with high

Table 6

English phrases containing common prepositions. All have high MI (by construction); most do not have high RIDF (though there are a few exceptions such as “Just for Men,” a well-known brand name).

tf	df	RIDF	MI	Preposition = “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	Preposition = “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	Preposition = “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	Preposition = “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 Canterbury
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

RIDF pick out specific documents (and/or events) and therefore tend to be relatively good keywords. Substrings with high MI have non-compositional distributions (if not semantics), and are therefore likely to be interesting to a lexicographer/linguist. Substrings that are high in both are more likely to be meaningful units (words/phrases) than substrings that are high in just one or the other. Meaningless fragments tend to be low in both MI and RIDF.

We grouped the Japanese classes into nine cells depending on whether the RIDF was in the top 10%, the bottom 10%, or in between and whether the MI was in the top 10%, the bottom 10% or in between. Substrings in the top 10% in both RIDF and MI tend to be meaningful words such as “merger,” “stock certificate,” “dictionary,” “wireless” in English sense and so on. Substrings in the bottom 10% in both RIDF and MI tend to be meaningless fragments, or straightforward compositional combinations of words such as “current regular-season game.” Table 7 shows examples where MI and RIDF point in opposite directions (see highlighted rectangles in panel b of Figure 10).

We have observed previously that MI is high for general vocabulary (words found in dictionary) and RIDF is high for names, technical terminology and good keywords for information retrieval. Table 7 suggests an intriguing pattern. Japanese uses different character sets for general vocabulary and loan words. Words that are high in MI tend to use the general vocabulary character sets (Hiragana and Kanji) whereas words that are high in RIDF tend to use the loan word character sets (Katakana and English). (There is an important exception, though, for names, which will be discussed in the next subsection.)

The character sets largely reflect the history of the language. Japanese uses four character sets (Shibatani, 1990). Typically, functional words of Japanese origin are written in Hiragana. Words that were borrowed from Chinese many hundreds of years ago are written in Kanji. Loan words borrowed more recently from Western languages are

Table 7

Examples of keywords with extreme values of RIDF and MI that point in opposite directions. The top half (high RIDF and low MI) tends to have more loan words, largely written in Katakana and English. The bottom half (low RIDF and high MI) tends to have more general vocabulary, largely written in Chinese Kanji.

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

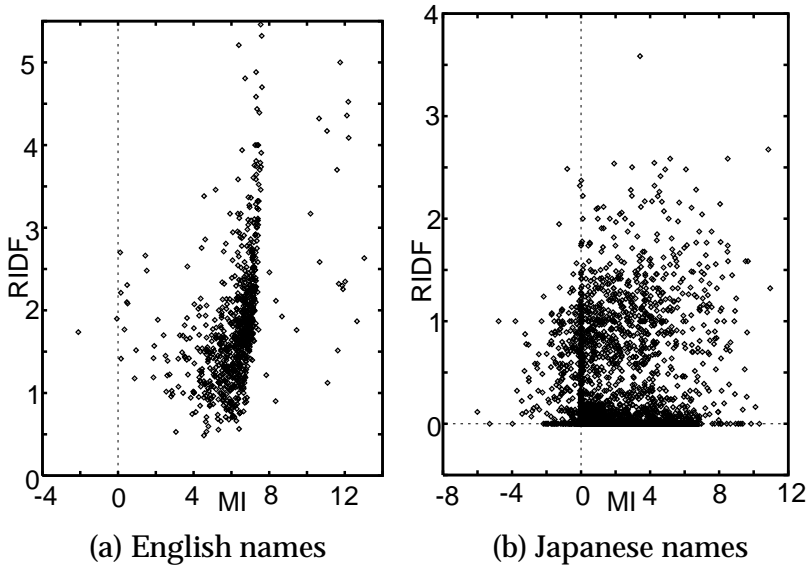


Figure 11
MI and RIDF of people's names.

written in Katakana. Truly foreign words are written in the English character set (also known as Romaji). We were pleasantly surprised to discover that MI and RIDF were distinguishing substrings on the basis of these character set distinctions.

3.3 Names

As mentioned above, names are an important exception to the rule that Chinese Kanji are used for general vocabulary (words found in the dictionary) which were borrowed hundreds of years ago and Katakana characters are used for more recent loan words (such as technical terminology). As illustrated in Table 7, Kanji are also used for Japanese names and Katakana are used for foreign names.

Names are quite different in English and Japanese. Figure 11 shows a striking contrast in the distributions of MI and RIDF values. MI has a more compact distribution in English than Japanese. RIDF is bimodal in Japanese, but not in English.

The names shown in Figure 11 were collected using a simple set of heuristics. For English, we selected substrings starting with the titles: “Mr.,” “Ms” or “Dr.” For Japanese, we selected keywords (as identified by the CD-ROM) ending with the special character (-shi), which is roughly the equivalent of the English titles “Mr.” and “Ms.” In both cases, phrases were required to have $tf \geq 10$.³

The English names have a sharp cutoff around $MI = 7$ due in large part of the title “Mr.” $MI('Mr.', x) = \log_2 \frac{N}{tf('Mr.')} - \log_2 \frac{tf(x)}{tf('Mr.', x)} = 7.4 - \log_2 \frac{tf(x)}{tf('Mr.', x)}$. Since $\log_2 \frac{tf(x)}{tf('Mr.', x)}$ is a small positive number, typically 0-3, $MI('Mr', x) < 7.4$.

Names generally have RIDF values ranging from practically nothing (for common names like “Jones”) to extremely large values for excellent keywords. The Japanese names, however, cluster into two groups, those with RIDF above 0.5, and those with RIDF below 0.5. The separation above and below RIDF=0.5, we believe, is a reflection of the well-known distinction between new information and given information in discourse structure. It is common in both English and Japanese, for the first mention of a name in a news article to describe the name in more detail than subsequent uses. In English,

³ This procedure produced the interesting substring, “Mr. From,” where both words would normally appear on a stop list. This name has a large RIDF. (The MI, though, is small because the parts are so high in frequency.)

for example, terms like “spokesman” and apositives are quite common for the first use of a name, and less so, for subsequent uses. In Japanese, the pattern appears to be even more rigid than in English. The first use will very often list the full name (first name plus last name), unlike subsequence uses which almost always omit the first name. As a consequence, the last name exhibits a large range of RIDF values, as in English, but the full name will usually (90%) fall below the $RIDF=0.5$ threshold. The MI values have a broader range as well, depending on the compositionality of the name.

To summarize, RIDF and MI can be used to identify a number of interesting similarities and differences in the use of names. Names are interestingly different from general vocabulary. Many names are very good keywords and have large RIDF. Names are bring up some interesting differences between English and Japanese such as the tendency for Japanese names to fall into two groups separated by the $RIDF=0.5$ threshold.

3.4 Word extraction

RIDF and MI may be useful for word extraction, a key issue when processing texts in many languages such as Chinese, Japanese and Thai. Unlike English, many of these languages do not use delimiters between words. There is a large literature on morphology and word extraction methods. Nagao and Mori (1994) and Nagata (1996) proposed ngram methods for Japanese. Sproat and Shih (1990) found MI to be useful for word extraction in Chinese.

We performed the following simple experiment to see if both MI and RIDF could be 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 subjective judgment. Substrings were labeled “good” (the substring is a word), “bad”

Table 8

The combination of RIDF and MI is better in a word extraction task than either by itself, which is better than neither. Each cell reports performance over a sample of 100 substrings. Substrings were subjectively judged to be “good” (the substring is a word), “bad” (the substring is not a word) or “gray” (the judge is not sure). Two performance values are reported indicating how much of the 100 substrings are words. The larger performance values count the “gray” substrings as words; the smaller performance values count the “gray” substrings as non-words.

	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%

(the substring is not a word) or “gray” (the judge is not sure). The results are presented in Table 8, which shows that substrings with high scores in both dimensions are more likely to be words than substrings that score high in just one dimension. Substrings with low scores in both dimensions are very unlikely to be words. These results demonstrate plausibility for the use of multiple statistics. The approach could be combined with other methods in the literature such as (Kita et al., 1994) to produce a more practical system. In any case, the automatic word extraction is not an easy task for Japanese (Nagata, 1996).

4 Conclusion

Bigrams and trigrams are commonly used in statistical natural language processing; this paper described techniques for working with much longer ngrams, including million-grams and even billion-grams. We presented algorithms (and C code) for computing term frequency (tf) and document frequency (df) for all ngrams (substrings) in a corpus (sequence). The method took only a few hours to compute tf and df for all the 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.

The method works by grouping substrings into classes so that the computation of tf and df over order N^2 substrings can be reduced to a computation over order N classes.

The reduction makes use of four properties:

properties 1-2 : all substrings in a class have the same statistics

(at least for the statistics of interest, namely *tf* and *df*),

property 3 : the set of all substrings are partitioned by the classes,

and

property 4 : there are many fewer classes (order N) than sub-

strings (order N^2)

The second half of the paper used the results of computing *tf* and *df* for all ngrams in the two large corpora mentioned above. We compared and contrasted RIDF and MI, statistics that are motivated by work in lexicography and information retrieval. Both statistics use distributional evidence as a surrogate for meaning. We would prefer to use meaning, if we could, but distributional evidence is available. Both statistics compare the frequency of an ngram to chance, but they use different notions of chance. RIDF looks for ngrams whose distributions over documents cannot be attributed to random (Poisson). These ngrams tend to be good keywords for information retrieval such as technical terms and names. MI looks for ngrams whose internal structure cannot be attributed to compositionality. MI tends to pick out general vocabulary, words and phrases that appear in dictionaries. We believe that both statistics are useful, but in different and complementary ways. In a Japanese word extraction task, the combination of MI and RIDF performed better than either by itself.

Acknowledgments

We would like to thank the anonymous reviewers for *Computational Linguistics* who made insightful comments on an earlier draft.

References

- Charniak, Eugene. 1993. *Statistical Language Learning*. MIT Press.
 Church, Kenneth W. and William A. Gale. 1995. Poisson mixtures. *Natural Language Engineering*, 1(2):163–190.
 Church, Kenneth W. and Patrick Hanks. 1990. Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29.
 Gaston H. Gonnet, Ricardo A. Baeza-Yates

- and Tim Snider. 1992. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structure & Algorithms*, pages 66–82. Prentice Hall PTR.
- Gusfield, Dan. 1997. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press.
- Hui, Lucas Chi Kwong. 1992. Color set size problem with applications to string matching. In *Lecture Notes in Computer Science, Volume 644, Springer (CPM92: Combinatorial Pattern Matching, 3rd Annual Symposium)*, pages 230–243.
- Jelinek, Frederick. 1997. *Statistical Methods for Speech Recognition*. MIT Press.
- Katz, Slava M. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(3):400–401.
- Kita, Kenji, Yasuhiko Kato, Takashi Omoto, and Yoneo Yano. 1994. A comparative study of automatic extraction of collocations from corpora: mutual information vs. cost criteria. *Journal of Natural Language Processing*, 1(1):21–33.
- Manber, Udi and Gene Myers. 1990. Suffix arrays: A new method for on-line string searches. In *the first Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327.
URL=<http://glimpse.cs.arizona.edu/udi.html>.
- McCreight, Edward M. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272.
- Nagao, Makoto and Shinsuke 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. In *Proceedings, 15th International Conference on Computational Linguistics*, pages 611–615.
- Nagata, Masaaki. 1996. Automatic extraction of new words from Japanese texts using generalized forward-backward search. In *Proceedings, Conference on Empirical Methods in Natural Language Processing*, pages 48–59.
- Paul, Douglas B. and Janet M. Baker. 1992. The design for the Wall Street Journal-based CSR corpus. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 357–361.
- Shibatani, Masayoshi. 1990. *The Languages of Japan*. Cambridge Language Surveys. Cambridge University Press.
- Sparck Jones, Karen. 1972. A statistical interpretation of ter specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21.
- Sproat, Richard and Chilin Shih. 1990. A statistical method for finding word boundaries in Chinese text. *Computer Processing of Chinese and Oriental Languages*, 4(4):336–351.
- Ukkonen, Esko. 1995. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Appendix: Implementations of the algorithms in C language

All codes in this appendix and the demonstration versions can be found at

<http://www.milab.is.tsukuba.ac.jp/~myama/tfdf>.

A : C code to print all lcp-delimited intervals using C language's stack

The function `output` (below) is called $2N - 1$ times. It will output an interval if the interval is lcp-delimited ($LBL < SIL$). Trivial intervals are always lcp-delimited. Non-trivial intervals are lcp-delimited if the bounding lcps are smaller than the $SIL=lcp[k]$, where k is the representative.

```
void output(int i, int j, int k){
    int LBL = (lcp[i] > lcp[j+1]) ? lcp[i] : lcp[j+1];
    int SIL = lcp[k];
    if(i==j) printf("trivial <%d,%d>, tf=1\n", i, j);
    else if(LBL < SIL) printf("nontrival <%d, %d>, rep=%d, tf=%d\n",
        i, j, k, j-i+1);
}

int print_LDIIs(int i, int k){
    int j = k;
    output(k,k,0); /* trivial intervals */
    while(lcp[k] <= lcp[j+1] && j+1 < N) j = print_LDIIs(k, j+1);
    output(i,j,k); /* non-trivial intervals */
    return j;}

```

B : C code to print all lcp-delimited intervals using the own stack

print_LDIstack is similar to print_LDI, but uses its own stack. It takes the corpus size, N, as an argument.

```
#define STACK_SIZE 100000

#define Top_i (stack[sp-1].i)
#define Top_k (stack[sp-1].k)

struct STACK {int i; int k;} stack[STACK_SIZE];

int sp = 0; /* stack pointer */

void push(int i, int k) {
    if(sp >= STACK_SIZE) {
        fprintf(stderr, "stack overflow\n");
        exit(2);}
    stack[sp].i = i;
    stack[sp++].k = k;}

void pop() {sp--;}

void print_LDIstack(int N) {
    int j;
    push(0,0);
    for(j = 0; j < N; j++) {
        output(j, j, 0);
        while(lcp[j+1] < lcp[Top_k]) {
```

```

        output(Top_i, j, Top_k);

        pop();}

    push(Top_k, j+1);}}

```

C : C code to print all lcp-delimited intervals with tf and df

The steps 5 and 6 of the algorithm in Section 2.6 are implemented as the function `dec_df` using the binary search.

```

#define STACK_SIZE 100000

#define Top_i (stack[sp-1].i)
#define Top_k (stack[sp-1].k)
#define Top_df (stack[sp-1].df)

struct STACK {int i; int k; int df;} stack[STACK_SIZE];

int sp = 0; /* stack pointer */

void push(int i, int k, int df) {
    if(sp >= STACK_SIZE){
        fprintf(stderr, "stack overflow\n");
        exit(2);}

    stack[sp].i = i;
    stack[sp].k = k;
    stack[sp++].df = df;}

void pop() {sp--;}

```

```
void output(int i, int j, int k, int df) {
    int LBL;
    if(lcp[i] > lcp[j+1]) LBL = lcp[i];
    else LBL = lcp[j+1];
    if(i==j) printf("trivial <%d,%d>, tf=1\n", i, j);
    else if(LBL < lcp[k])
        printf("nontrivial <%d, %d>, rep=%d, tf=%d, df=%d\n",
            i, j, k, j-i+1, df);
}

/*
 * Print_LDIIs_with_df does not only print tf, but also df.
 * It takes the corpus size, N, and the number of documents, D.
 * doc() returns the document number of the suffix array's index.
 * dec_df() decrease a df-counter in the stack when duplicate
 * counting occurs.
 */

void dec_df(int docid) {
    int beg=0, end=sp, mid=sp/2;
    while(beg != mid) {
        if(doclink[docid] >= stack[mid].i) beg = mid;
        else end = mid;
        mid = (beg + end) / 2;
    }
    stack[mid].df--;
}
```

```
}

print_LDIs_with_df(int N, int D) {
    int i, j, df;
    doclink = (int *)malloc(sizeof(int) * D);
    for(i = 0; i < D; i++) doclink[i] = -1;
    push(0,0,1);
    for(j = 0; j < N; j++) {
        output(j,j,0,1);
        if(doclink[doc(j)] != -1) dec_df(doc(j));
        doclink[doc(j)] = j;
        df = 1;
        while (lcp[j+1] < lcp[Top_k]) {
            df = Top_df + df;
            output(Top_i,j,Top_k,df);
            pop();
        }
        push(Top_k, j+1, df);
    }
}
```