Substring Statistics

Kyoji Umemura[1] and Kenneth Church[2]

[1] Toyohashi University of Technology, Tempaku, Toyohashi, Aichi 441-8580, Japan
[2] Microsoft, One Microsoft Way, Redmond, WA 98052, USA

Abstract. The goal of this work is to make it practical to compute corpus-based statistics for all substrings (ngrams). Anything you can do with words, we ought to be able to do with substrings. This paper will show how to compute many statistics of interest for all substrings (ngrams) in a large corpus. The method not only computes standard corpus frequency, $freq$, and document frequency, df, but generalizes naturally to compute, $df_k(str)$, the number of documents that mention the substring str at least k times. df_k can be used to estimate the probability distribution of str across documents, as well as summary statistics of this distribution, e.g., mean, variance (and other moments), entropy and adaptation.

1 Introduction

Substring (ngram) statistics are fundamental to nearly everything we do: language modeling (for speech recognition, OCR and spelling correction), compression, information retrieval, word breaking and more. Many textbooks discuss applications of ngrams including [14] [16] [17] [13] [8] [5] [7] [12]. This paper describes an easy-to-follow procedure for computing many popular statistics for all substrings (ngrams) in a large corpus. **C** code is posted at [21].

[19] showed how to compute standard corpus frequency, $freq$, and document frequency, df, for all substrings in a large corpus. Document frequency is a commonly used statistic, especially in the Information Retrieval community [2] [5]. Document frequency is traditionally defined over words or terms, though we will apply it to substrings.

Definition 1. $df(str) \equiv$ number of documents that mention str at least once.

Generalized document frequency, $df_k(str)$, is the number of documents that mention str at least k times. For expository convenience, we use cdf_k for the cumulative document frequency: $cdf_k(str) \equiv \sum_{i=k}^{\infty} df_i(str)$. We can work directly with cdf_k as in [20], or alternatively, cdf_k can be used to compute more standard quantities such as frequency and df_k:

$$freq = cdf_1$$
$$df_k = cdf_k - cdf_{k+1}$$

df_1 is the same as standard document frequency (df). df_2 plays an important role in *adaptation*, a term borrowed from the literature on language modeling

for speech recognition [13, chapter 14], where there is considerable interest in adapting the probabilities to the first few words of a document. If "Noriega" is mentioned early in the document, chances are that that word (and its friends) will be mentioned again[15]. Psychologists use the term priming [1] to reflect the fact that people react quicker and more accurately to "nurse" if it has been primed by a highly associated word like "doctor."

We define adaptation to be the chance that a term will be mentioned again, given that we have seen it before.

Definition 2. *adaptation* $\equiv Pr(k \geq 2 | k \geq 1) \approx df_2/df_1$

There are huge quantity discounts, especially for good keywords. The first mention costs $-log(df_1/D)$ bits, but subsequent mentions are cheaper: $-log(df_2/df_1)$ bits. For good keywords like "Noriega," the first mention is quite surprising (e.g., "Noriega" is mentioned in one document in a thousand Associcated Press (AP) stories, shortly after the US invasion of Panama), but the subsequent mention is less so (more than half of the documents that mention "Noriega" once, mention him a second time). There is considerably less adaptation for function words and meaningless random substrings, where the first mention and subsequent mentions are about equally likely (no quantity discounts).

This discounting view is reminiscent of the Given-New Distinction in Discourse Theory [3], which is commonly used in intonation studies such as [4]. The first mention ("new" information) is marked, whereas subsequent ("given") mentions are unmarked. In statistical terms, first mentions tend to be more surprising than subsequent mentions, at least for meaningful words. Random substrings behave more randomly, with less difference between first mentions and subsequent mentions.

In Japanese, we find that words adapt more than most substrings of Japanese characters, and therefore we believe adaptation could be useful in word-breaking applications for Asian languages.

2 Suffix Arrays

A suffix array [9] is a convenient data structure for computing the frequency and location of a substring (ngram) in a large corpus. The input text (corpus) is a sequence of N tokens. Tokens can be words, bytes, Asian characters, etc.

We will use different tokenization rules from time to time. The simplest tokenization rule is to split the text up into bytes, starting a suffix at each byte position. In this case, the number of suffixes, S, is the same as the number of bytes in the input corpus N. For Japanese and Chinese text, it is more appropriate to tokenize by characters (typically 2-bytes), rather than by bytes, so that $S \approx N/2$. For English text, it is often convenient to start suffixes at word boundaries so $S \approx N/5$. The code posted at [21] provides options for different tokenization rules. For expository convenience, we will assume, for the remainder of this discussion, the simplest (and worst case) where $S = N$.

```
s = (int *)malloc(N * sizeof(int));
for(i=0; i<N; i++) s[i] = i; /* initialize */
qsort(s, N, sizeof(*s), sufcmp); /* sort lexicographically */

int sufcmp(int *a, int *b) { return strcmp(text + *a, text + *b); }
```

Fig. 1. A simple procedure creating a suffix array from an input corpus (*text*). The suffix array, $s[i]$, is initialized to the integers from 0 to $N-1$. Each integer denotes a suffix or semi-infinite string, starting at position $s[i]$ and extending to the end of the corpus. The integers in s are then sorted so the semi-infinite strings will be in alphabetical order.

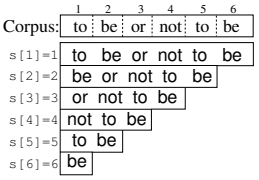

Fig. 2. Illustration of a suffix array, s, after initialization but before sorting. Each element in the suffix array, $s[i]$, is an integer denoting a suffix or semi-infinite string, starting at position $s[i]$ in the corpus and extending to the end of the corpus.

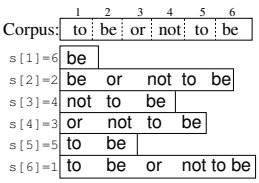

Fig. 3. The suffix array, s, after sorting. The integers in s are sorted so that the semi-infinite strings are now in alphabetical order.

The suffix array of the input is an array of integers, s, whose elements, $s[i]$, denote semi-infinite strings ($suffix[i]$), sorted in lexicographic order. Each semi-infinite string starts at position $s[i]$ in the text and extends to the end of the text. The semi-infinite string, $suffix[i]$, is represented in **C** with a single integer, consuming just $O(1)$ space, but it denotes a long substring, $substr(text, s[i])$, which would require $O(N)$ bytes (if we materialized it). Another way to express the materialized semi-infinite string is the **C** expression, $text + s[i]$, where $text$ is a string (char *) and i is an integer (int).

Definition 3. $suffix[i] \equiv substr(text, s[i]) = text + s[i]$

A simple procedure for constructing the suffix array, s, is shown in Figure 1. Figures 2 and 3 show the suffix array before and after sorting.

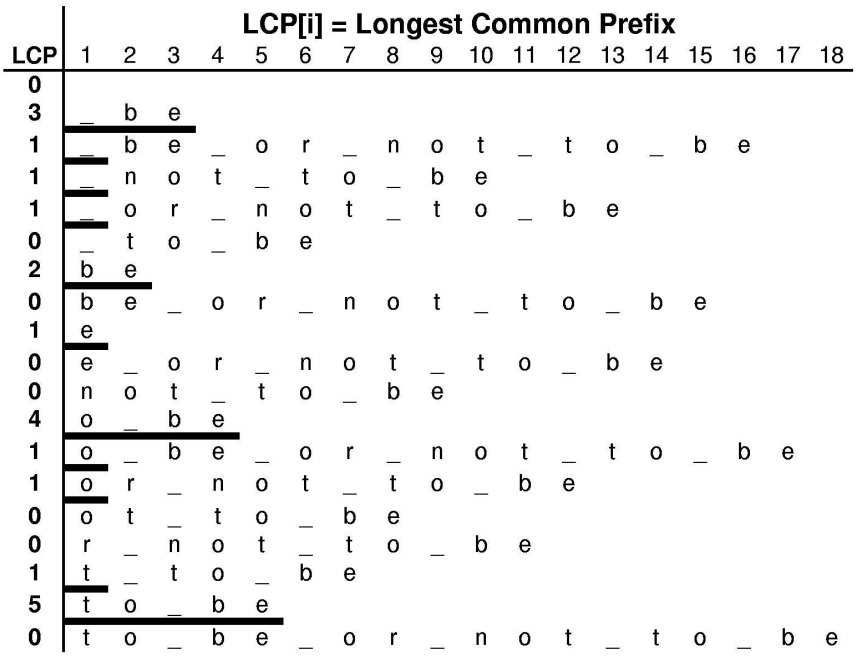

Fig. 4. LCP (Longest Common Prefix). The bars highlight the length of the common prefix shared between the pair of substrings just above and just below the bar.

See http://www.cs.dartmouth.edu/~doug for an excellent tutorial with **C** code of a more sophisticated algorithm with better theoretical bounds[9]. If appropriate care is taken to remove duplicate documents, it has been our experience that the complications of the more sophisticated algorithm are often not needed in practice, and can actually degrade performance (slightly).

2.1 LCP (Longest Common Prefix)

In addition to s, the suffix array literature also makes use of Longest Common Prefixes (LCP), as illustrated in Figure 4. $LCP[i]$ contains the length of the common prefix between $s[i]$ and $s[i+1]$. We will refer to "prefixes of suffixes" as "substrings." LCP is a vector of N ints, like s.

The suffix array literature shows how to compute the LCPs efficiently, even in the worst case. The code posted at [21] provides a much more straightforward implementation, which is simpler to understand, and is often faster in practice, but can take $O(N^2)$ time in the worst case.

2.2 sufconc

As mentioned above, suffix arrays make it convenient to find the frequency and location of any substring in a large corpus. The example below, for example, shows a couple of concordance lines for the substring "Manual Noriega" from a month of Associated Press news (December, 1989), as distributed by TIPSTER and the LDC [10]. The sufconc program takes an input substring pattern and performs two binary searches to find an interval on the suffix array: $< i, j >$. $s[i]$ is the first suffix in the suffix array that starts with the input pattern ("Manuel Noriega"); s[j] is the last suffix that starts with "Manuel Noriega." Standard corpus frequency can be computed straightfowardly from the width of the interval $< i, j >$. That is, $freq = j - i + 1$.

The sufconc program, as illustrated in Figure 5, prints out a concordance line for each suffix in the interval $< i, j >$. The subroutine that materializes suffixes and prints them out is called *pname*, by analogy to the LISP function that converts a symbol pointer to a string. We view the elements of a suffix array as analogous to symbols in a symbol table. As with LISP, most of the processing can be done in terms of the pointers, and except for printing routings, there shouldn't be much need to dereference the pointers and materialize the strings. The *pname* function converts newlines to "\n." Similar conversions are performed for other characters that would cause trouble. A carrot ("∧") is inserted just before the input pattern. The command line arguments specify how much context to print to the left and right of the carrot. The first two numbers of each line are $s[i]$ and the associated document id, $doc(s[i])$.

```
sufconc -l 10 -r 40 AP/AP8912 'Manuel Noriega' | sed 3q
17913368     5441: osed Gen.  ^ Manuel Noriega\nin Panama _ their wives
13789741     4193: apprehend  ^ Manuel Noriega\n   The situation in Pana
 3966027     1218: nian Gen.  ^ Manuel Noriega a\n$300,000 present, and
```

Fig. 5. A concordance computed from a suffix array. The sufconc program prints out a concordance line for each suffix in the interval $< i, j >$. Newlines are translated to "\n." Similar conversions are performed for other characters that would cause trouble. A carrot (∧) is inserted just before the input pattern. The command line arguments specify how much context to print to the left and right of the carrot. The first two numbers of each line are the suffix, $s[i]$, and the associated document id, $doc(s[i])$.

3 Classes

3.1 Distributional Equivalence

We have seen thus far how to compute the frequency and location for a particular substring. This section will show how to compute these statistics for all substrings, not just for a particular substring.

Although there are too many substrings ($N(N+1)/2$) to work with directly, they can be grouped into a manageable number of N equivalence classes [19]. The construction starts with an interval $< i, j >$ on the suffix array, where $i \leq j$. From this interval, we construct an equivalence class, the (possibly empty) set of substrings that start every suffix within the interval, and no other suffixes. We say that the interval is *valid* iff the class is non-empty.

Classes form an equivalence relation, *distributional equivalence*, on substrings. For example, in the "to be or not to be" example, the substrings "to" and "to be," which are in the same class, have identical distributions; they both start exactly the same set of suffixes. Distributional equivalence is reflexive, symmetric and transitive. Classes partition the set of all substrings; every substring is a member of one and only one class.

Distributionally equivalent substrings have the same statistics, at least for many popular statistics including standard corpus frequency, document frequency, joint document frequency, and combinations of these quantities including moments, entropy, adaptation, etc. In particular, all the substrings in $Class(< i, j >)$ have the same frequency $(j - i + 1)$. The set of substrings in a class can be computed from the LCP vector, as illustrated in Figure 6. $Class(< i, j >) = \{$ substrings $str \mid str$ starts every suffix in $< i, j >$ and no others $\} = \{substr(text + s[i], 1, k)\}$ for $LBL < k \leq SIL$.

The LBL (Longest Bounding LCP) $= max(LCP[i-1], LCP[j])$. The SIL (Shortest Interior LCP) $= MIN_{i \leq k < j} LCP[k]$. In fact, we can replace $s[i]$ with $s[w]$ for any witness $i \leq w \leq j$ since all of the suffixes in the interval are the same up to the SIL. We will refer to the longest member of $Class(< i, j >) = substr(text + s[i], 1, SIL)$ as the canonical member of the equivalence class.

Table 1 shows that there are lots of substrings and lots of intervals (order N^2), but relatively few interesting classes (at most N). We say that an interval is invalid if the class is empty, and we say that the class is trivial if the frequency is 1. Most of the N^2 intervals are invalid, and most of the valid intervals are trivial. There are relatively few remaining classes (non-trivial and non-empty).

Figure 8 illustrates graphically the massive reduction from N^2 down to N. The string "to be or not to be\$" has N=19 characters, and $N * (N-1)/2 = 171$ possible substrings. For each substring, there is a possible interval $< i, j >$ on the suffix array, but only 8 of the 171 possible intervals are interesting (non-empty and non-trivial).

The main motivation for grouping substrings into classes is the computational consideration: N is more manageable than N^2. Most statistics of interest can be computed over classes rather than substrings because distributionally equivalent substrings have the same statistics, at least for many popular statistics.

3.2 Enumerating Classes

Figure 9 shows that interesting (non-empty and non-trivial) intervals form a tree. The procedure in Figure 10 performs a depth-first traversal of this tree. The classes, $< i, j >$, are enumerated in sorted order, sorted first by j in increasing order and then by i in decreasing order.

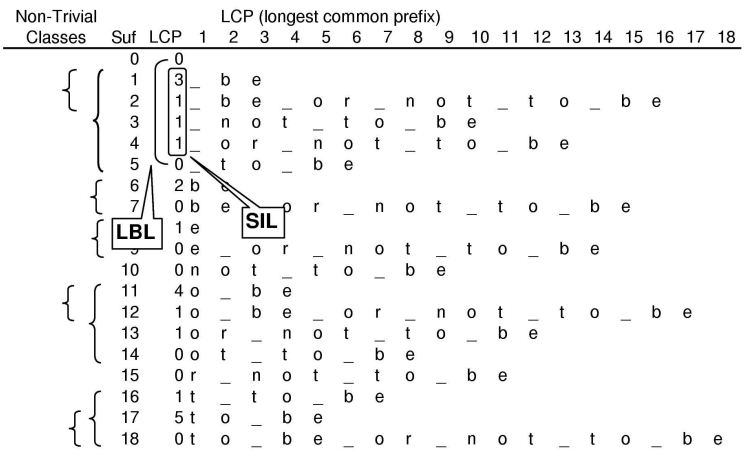

Fig. 6. $Class(< i,j >) = \{$ substrings $str \mid str$ starts every suffix in $< i,j >$ and no others $\} = \{\ substr(text + s[w], 1, k)\ \}$ for $LBL < k \leq SIL$. The LBL (Longest Bounding LCP) $= max(LCP[i-1], LCP[j])$. The SIL (Shortest Interior LCP) $= MIN_{i \leq k < j} LCP[k]$. The interval $< 1, 5 >$, for example, has an LBL of 0 and a SIL of 1. Thus, the class contains just one substring, $substr(text + s[1], 1, 1) =$ "_".

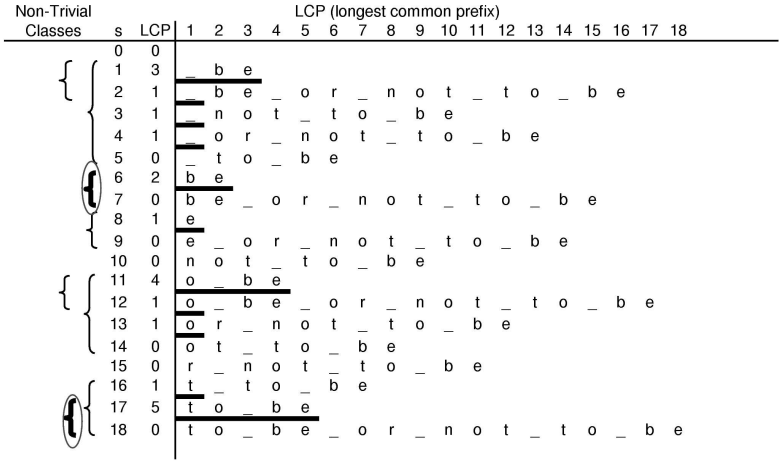

Fig. 7. The $N(N+1)/2$ substrings can be grouped into N or fewer equivalence classes. A class is defined in terms of intervals $< i,j >$ on the suffix array. The class contains the set of substrings that start every suffix within the interval and no suffixes outside the interval. The circles highlight two examples. $Class(< 6,7 >) =$ {b, be}. $Class(< 17,18 >) =$ {to, to_, to_b, to_be}.

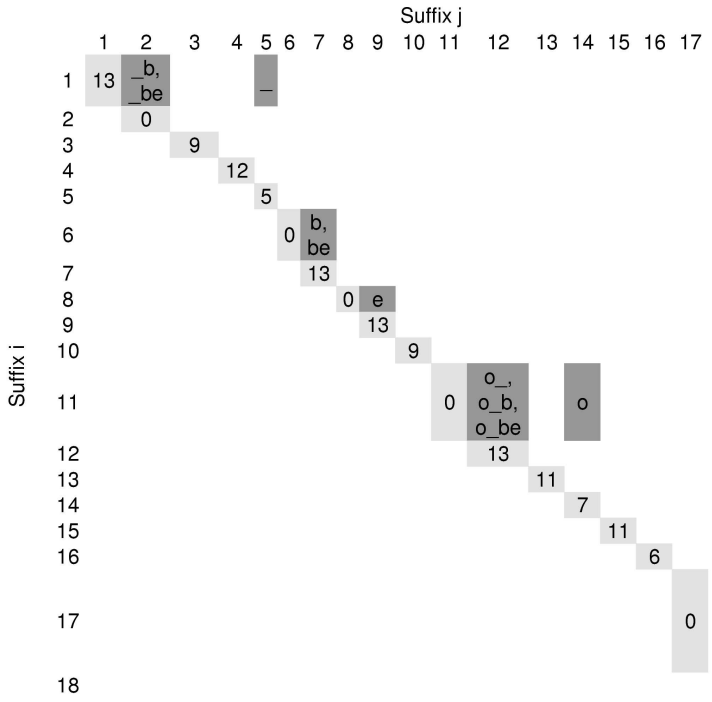

Fig. 8. The 171 substrings fall into just 8 interesting classes. Most of the classes are empty (invalid). Most of the rest are trivial (freq = 1). The trivial classes fall along the main diagonal. There are just 8 interesting classes (non-empty and non-trivial). 15 substrings appear in the 8 interesting classes; 135 substrings appear in trivial classes along the main diagonal.

Table 1. Although there are too many substrings to work with (N^2), they can be grouped into a manageable number of interesting (non-empty and non-trivial) classes.

Objects	Description	Quantity
Text	the Corpus	N tokens
Suffix Array	s[i] = substr(text, i)	N suffixes or less
Substring	substr(text, i, j)	$N(N+1)/2$ or less
Interval on Suffix Array	$< i, j >= \{s[k] \vert i \leq k \leq j\}$	$N(N+1)/2$ or less
$Class(< i, j >)$	{ substrings *str* | *str* starts every suffix in $< i, j >$, and no others }	$N(N+1)/2$ or less
Valid (Non-Empty) Class	$Class(< i, j >) \neq \emptyset$	$2N$ or less
Trivial Class	$< i, i > \Rightarrow freq(< i, i >) = 1$	N or less
Interesting Class (Non-Trivial and Non-Empty)	$Class(< i, j >) \neq \emptyset \wedge i \neq j$	N or less

The fact that the output is sorted is convenient for retrieval purposes. Table 2 shows the output from find_class. This program takes a string such as "Norieg" as input and retrieves the $Class(< i, j >)$ as well as the LBL, SIL, a number of pre-computed statistics and the set of distributionally equivalent substrings. The program performs three binary searches. There are two binary searches into the suffix array to find $< i, j >$, the first and last suffix starting with the input pattern. The third binary search is performed on the classes, as enumerated by the method described above. Some of the values in Table 2 could have been computed without classes (i, j, LBL, freq), but others benefit considerably from classes (SIL, df_k, distributionally equivalent substrings).

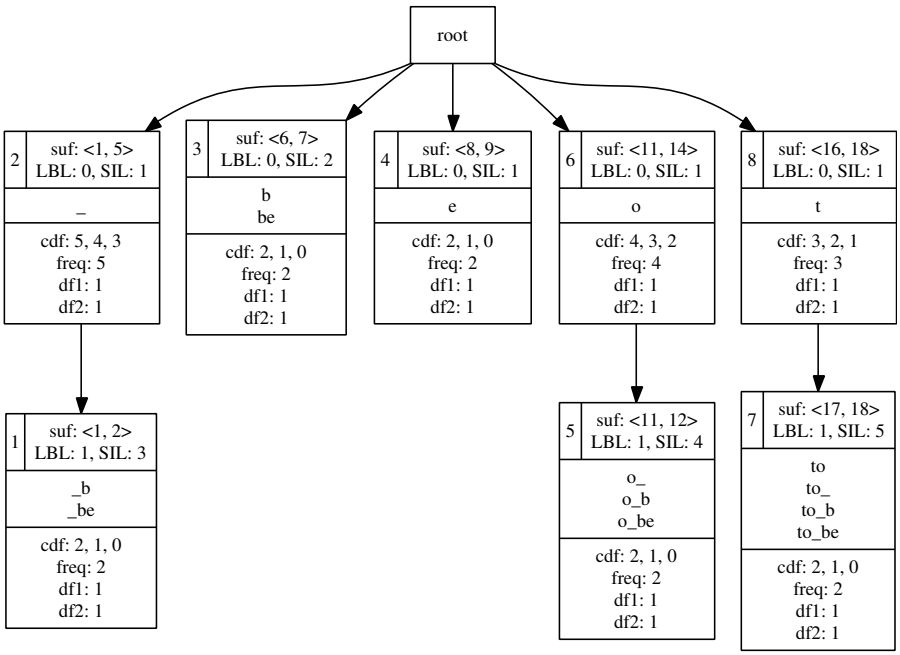

Fig. 9. Class Tree for "to be or not to be$." Interesting (non-empty and non-trivial) intervals form a tree. The procedure in Figure 10 performs a depth-first traversal of this tree. The numbers in the upper left hand corner of each node indicate the class id, the position of the class in the depth-first traversal. The class id is followed by the interval $< i, j >$, LBL and SIL. The middle of each node lists the substrings in $Class(< i, j >)$. At the bottom of each node are various statistics that will be discussed later.

Table 3 shows snapshots of the stack as each of the 8 classes are output using the depth-first traversal introduced in Figure 10. The stack pointer corresponds (roughly) to the depth of the class tree shown in Figure 9, though not exactly because of head recursion. Although the class tree reaches a depth of 3 in 3 places in Figure 9 (class ids 1, 5 and 7), the stack pointer reaches 3 in just one

```
struct stackframe { int i, j, SIL } *stack;
int sp = 0; /* stack pointer */
stack[sp].i = 0; stack[sp].SIL  = -1;

for(w=0; w<N; w++) {
    if(LCP[w] > stack[sp].SIL) {
        sp++; /* push */
        stack[sp].i = w;
        stack[sp].SIL = LCP[w]; }
    while(LCP[i] < stack[sp].SIL) {
       stack[sp].j = w;
       output(&stack[sp]);
       if(LCP[w] <= stack[sp-1].SIL)  sp--; /* pop */
       else stack[sp].SIL = LCP[w]; }} /* head recursion */
```

Fig. 10. A depth-first enumeration of classes.

.

Table 2. Find class: inputs a pattern (a substring such as "Norieg") and outputs $< i, j >$, LBL, SIL, a number of pre-computed statistics and the set of distributionally equivalent substrings, $Class(< i, j >)$. (The table below shows just the longest and shortest member of $Class(< i, j >)$; the others have be replaced with "...")

Source	i	j	LBL	SIL	freq	df_1	df_2	$Class(< i, j >)$
AP8901	6693634	6693741	4	7	108	22	14	{ Norie,...,Noriega }
AP8902	5927492	5927574	4	7	83	16	12	{ Norie,..., Noriega }
AP8903	6373730	6373804	3	7	75	14	11	{ Nori,...,Noriega }
AP8904	6121594	6121819	4	7	226	34	30	{ Norie,...,Noriega }
AP8905	6804959	6806255	4	7	1297	188	158	{ Norie,...,Noriega }
AP8906	6470367	6470572	3	7	206	45	31	{ Nori,...,Noriega }
AP8907	6389275	6389438	4	7	164	35	26	{ Norie,...,Noriega }
AP8908	6150308	6150676	4	7	369	63	52	{ Norie,...,Noriega }
AP8909	6353292	6353519	4	7	228	38	28	{ Norie,...,Noriega }
AP8910	6818197	6819484	4	6	1288	180	149	{ Norie,...,Norieg }
AP8911	6320624	6320745	4	7	122	29	20	{ Norie,...,Noriega }
AP8912	5968758	5971461	4	6	2704	403	327	{ Norie,...,Norieg }

place (class id 7). Head recursion avoids pushing the stack in the other two cases (class ids 1 and 5).

Head recursion is the dual of tail recursion [6]. Tail recursion replaces right branching recursion structures with iteration; head recursion does the same but for left branching structures. Class ids 1 and 5 are examples of left branching structures where the interval for the mother $(< i, j >)$ and the interval for the daughter $(< i, l >)$ share the same starting point (i). In the case of class id 1, both the mother $(< 1, 5 >)$ and the daughter $(< 1, 2 >)$ share the same starting point (1). Similarly, in the case of class id 5, both the mother $(< 11, 14 >)$ and

the daughter ($< 11, 12 >$) share the same starting point (11). Class id 7, on the other hand, is an example of a right branching structure since the mother ($< 16, 18 >$) and the daughter ($< 17, 18 >$) share the same endpoint (18).

The last line of Figure 10 implements the head recursion for left branching structures. After outputting class id 1, the last line of Figure 10 recycles the daughter's stackframe (class id 1) for its mother (class id 2). The same pattern applies after class id 5, where the daughter's stackframe is also recycled for its mother (class id 6).

Table 3. Stack Trace. Snapshots of the stack are shown as each of the 8 classes are output using the depth-first traversal introduced in Figure 10. The stack pointer corresponds (roughly) to the depth of the class tree shown in Figure 9. The stack frames for the root node are omitted because they are not very interesting.

Stack Pointer	i	SIL	Canonical member of class($< i, j >$)	Output class id	Output interval	Output SIL
2	1	3	_be	1	$< 1, 2 >$	3
2	1	1	_	2	$< 1, 5 >$	1
2	6	2	be	3	$< 6, 7 >$	2
2	8	1	e	4	$< 8, 9 >$	1
2	11	4	o_be	5	$< 11, 12 >$	4
2	11	1	o	6	$< 11, 14 >$	1
2	16	1	t			
3	17	5	to_be	7	$< 17, 18 >$	5
2	16	1	t	8	$< 16, 18 >$	1

With suffix arrays, we could compute the frequency and location for a particular substring (ngram). Classes make it feasible to do that and more (generalized document frequency) for all substrings.

4 Document Frequency

Generalized document frequency, $df_k(str)$, is the number of documents that contain str at least k times. The method will compute cumulative document frequency, $cdf_k = \sum_{i=k}^{\infty} df_i(str)$. As mentioned above, frequency and df_k can be recovered from cdf_k:

$$freq = cdf_1$$

$$df_k = cdf_k - cdf_{k+1}$$

A simple (but slow) method for computing cdf_k from an interval $< i, j >$ is

$$cdf_1(< i, j >) = \sum_{i \le w \le j} 1 = j - i + 1$$

$$cdf_2(< i, j >) = \sum_{i \le w \le j} \begin{cases} 1 & if\ neighbor[w] \ge i \\ 0 & otherwise \end{cases}$$

$$cdf_k(< i, j >) = \sum_{i \le w \le j} \begin{cases} 1 & if\ neighbor^{k-1}[w] \ge i \\ 0 & otherwise \end{cases}$$

where $Neighbors^k[s] \equiv Neighbors^{k-1}[Neighbors[s]]$, for $k \neq 1$. (The first two formulas above can be viewed as special cases of the third, where $neighbor^0$ is the identity function.)

Neighbors is a function from suffixes to suffixes. Neighbors$[s_2] = s_1$ if s_1 and s_2 are in the same document and s_1 and s_2 are adjacent. By adjacent, we mean there is no suffix s_3 in the same document that is between s_1 and s_2. Table 6 shows the neighbors for the sample corpus in Table 4.

Table 4. A sample corpus with three documents.

doc	body
0	Hi_Ho_Hi_Ho
1	Hi_Ho
2	Hi

Figure 11 shows a simple (but slow) method to compute cdf_k, using a straightforward implementation of the discussion above. This method is slow because the class tree structure may become very deep. Figure 12 is the same as Figure 11 except that the top level loop has been folded into the depth-first traversal of the class tree.

We recommend the faster (nearly linear time) improvement in Figure 13. The speed-up is achieved by propagating counts up the class tree. The mother receives all of the counts from her daughters plus whatever counts she receives on her own ($cdf_k[mother] \ge \sum_{d \in daughters} cdf_k[d]$). The results are shown in Figure 14.

See [21] for **C** code that inputs a text file and outputs a number of indexes including the suffix array, LCP, classes and cdf_1, cdf_2 and cdf_3, using the recommendation in Figure 13.

5 Some Practical Experience with AP Newswire

The code in [21] has been applied to 12 months of the 1989 AP news as distributed by TIPSTER and the LDC [10] to compute the suffix array, LCP, classes and cdf_1, cdf_2 and cdf_3. Space and time are dominated by the suffix array computation. While there are a prohibitive number of possible substrings ($N(N-1)/2$), Table 7 shows that there aren't that many interesting (non-trivial non-empty) classes (C). In practice, C is quite a bit smaller than N ($C \approx N/2$), which is quite a bit better than the bound in Table 1, $C \le N$, based on theoretical considerations.

Table 5. Suffix array for Table 4. p is an offset into the corpus; s is an offset into the suffix array.

doc	p	s	suffix
2	20	0	“”
1	17	1	“”
0	11	2	“”
2	18	3	“Hi”
0	6	4	“Hi_Ho”
1	12	5	“Hi_Ho”
0	0	6	“Hi_Ho_Hi_Ho”
1	15	7	“Ho”
0	9	8	“Ho”
0	3	9	“Ho_Hi_Ho”
0	5	10	“_Hi_Ho”
0	8	11	“_Ho”
1	14	12	“_Ho”
0	2	13	“_Ho_Hi_Ho”
2	19	14	“i”
0	7	15	“i_Ho”
1	13	16	“i_Ho”
0	1	17	“i_Ho_Hi_Ho”
1	16	18	“o”
0	10	19	“o”
0	4	20	“o_Hi_Ho”

```
struct stackframe { int start, SIL, cdfk } *stack;

/* returns neighbor^k(suf) or -1 if NA */
int kth_neighbor(int suf, int k)
{ if(suf >= 0 || k > 1)
    return(kth_neighbor(neighbors[suf], k-1));
  else return suf; }

struct class c;
while(fread(&c, sizeof(c), 1, stdin)) {
  int cdfk = 0;
  for(w=c.start; w<=c.end; w++)
    if(kth_neighbor(w, K-1) >= c.start) cdfk++;
  putw(cdfk, out); } /* report */
```

Fig. 11. Simple (but slow) code for cdf_k, using a straightforward implementation of $cdf_k(< i, j >) = \sum_{i \le w \le j} \begin{cases} 1 & if\ neighbor^{k-1}[w] \ge i \\ 0 & otherwise \end{cases}$

Table 6. Neighbors Neighbors[s_2] = s_1 if s_1 and s_2 are in the same document and s_1 and s_2 are adjacent. By adjacent, we mean there is no suffix s_3 in the same document that is between s_1 and s_2. For example, for the 21-byte collection in Figure 4, there are 21 suffixes as shown in Table 5. Each of the three documents in the collection are associated with a set of suffixes. The neighbors can be computed by "shifting" these sets. Thus, the first suffix in each document has no neighbor (neighbor[2] = neighbor[1] = neighbor[0] = "NA"). Subsequent suffixes are mapped to the previous suffix in the same document, as illustrated below: neighbors[4] = 2, neighbors[6] = 4, etc.

doc	s
0	2, 4, 6, 8, 9, 10, 11, 13, 15, 17, 19, 20
1	1, 5, 7, 12, 16, 18
2	0, 3, 14

doc 0		doc 1		doc 2	
s	neighbor	s	neighbor	s	neighbor
2	NA	1	NA	0	NA
4	2	5	1	3	0
6	4	7	5	14	3
8	6	12	7		
9	8	16	12		
10	9	18	16		
11	10				
13	11				
15	13				
17	15				
19	17				
20	19				

```
for(w=0; w<N; w++) {
   if(LCP[w]> stack[sp].SIL) {
      sp++;
      stack[sp].start = w;
      stack[sp].SIL = LCP[w];
      stack[sp].cdfk = 0; }

   for(sp1=0; sp1<=sp; sp1++) {
      if(kth_neighbor(w, K-1) >= stack[sp1].start)
        stack[sp1].cdfk++; }

    while(LCP[w] < stack[sp].SIL) {
      putw(stack[sp].cdfk, out); /* report */
      if(LCP[w] <= stack[sp-1].SIL) sp--;
      else stack[sp].SIL = LCP[w]; }}
```

Fig. 12. Same as Figure 11, but folded into the depth-first traversal of the class tree.

```
/* return first stack frame not before suffix */
/* binary search works because stack is sorted */
int find(int suffix)     /* log(max(LCP)) time */
{ int low = 0;
  int high = sp;
  while(low + 1 < high) {
    int mid = (low + high) / 2;
    if(stack[mid].start <= suffix) low = mid;
    else high = mid;  }
  if(stack[high].start <= suffix) return high;
  if(stack[low].start <= suffix)  return low;
  fatal("can't get here"); }

for(w=0; w<N; w++) {  /* N time */
  if(LCP[w]> stack[sp].SIL) {
    sp++;
    stack[sp].start = w;
    stack[sp].SIL = LCP[w];
    stack[sp].cdfk = 0; }

    int prev = kth_neighbor(w, K-1);
    if(prev >= 0) stack[find(prev)].cdfk++;

    while(LCP[w] < stack[sp].SIL) {
      putw(stack[sp].cdfk, out); /* report */
      if(LCP[w] <= stack[sp-1].SIL) {
         /* propagate counts up class tree */
         stack[sp-1].cdfk += stack[sp].cdfk;
         sp--; }
      else stack[sp].SIL = LCP[w]; }}
```

Fig. 13. We recommend this this $O(N\ max(k,\ log\ max(LCP)))$ procedure for cdf_k.

For practical applications, the most serious concern is physical memory. If we wanted to scale up from a month of newswire (4M words) to the web (20B pages), we would need to generalize the methods to work in external memory, distributed across a cluster of machines. It is not too difficult to convert most of the **C** code to work in external memory, though we are not aware of a publicly available external memory implementation of suffix arrays.

The recommended speed ups make it practical to compare df_1 and df_2 at scale, as illustrated in Figure 15. Both df_1 and df_2 have a Zipf-like distribution, but the df_1 curve is well above df_2 (especially at low frequencies). It is interesting that the two lines are not parallel. These two lines determine adaptation, $Pr(k \geq 2|k \geq 1) \approx df_2/df_1$.

The code is [21] also makes it easy to compute LCPs at scale, though that had been possible before the recommended speed ups. Figure 16 shows LCPs

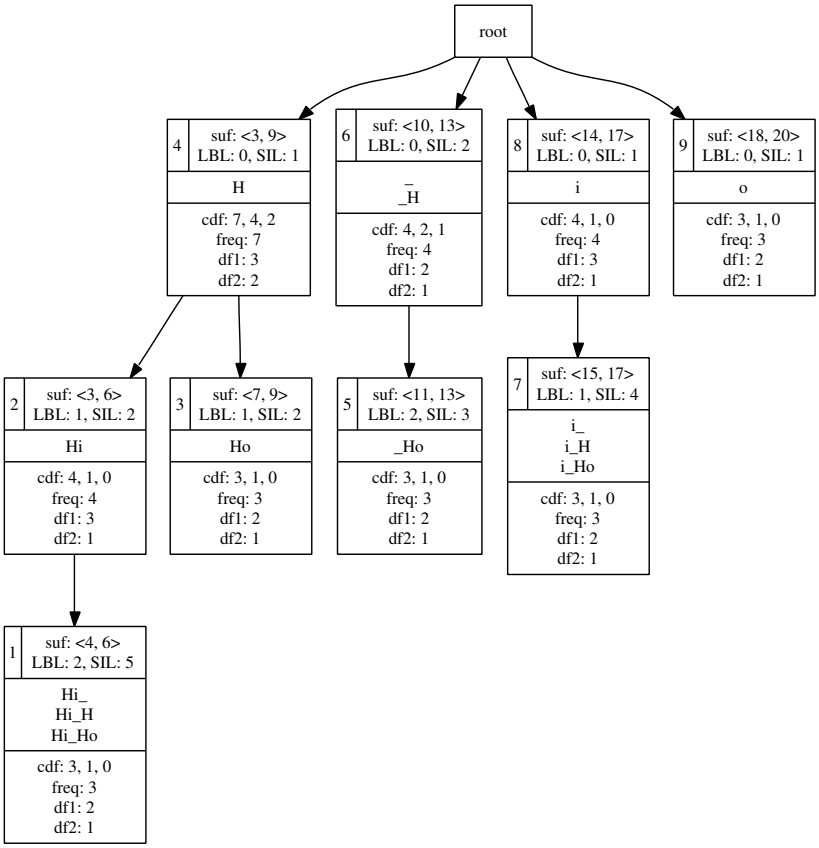

Fig. 14. Results (class tree and cdf_k) for input in Table 4. The numbers in the upper left hand corner of each node indicate the class id, the position of the class in the depth-first traversal of the class tree. The class id is followed by the interval $< i, j >$, LBL and SIL. The middle of each node lists the substrings in $Class(< i, j >)$. At the bottom of each node, cdf_1, cdf_2 and cdf_3 are presented, followed by $freq$, df_1 and df_2, where $freq = cdf_1$ $df_1 = cdf_1 - cdf_2$ $df_2 = cdf_2 - cdf_3$

for three months of AP news. The three distributions are nearly identical to one another (at least for reasonable LCPs up to tens of bytes). The very long LCPs are associated with artifacts in the AP data such as duplicated stories and boilerplate.

Artifacts raise serious issues for language modeling. Consider the difference between "in Monday" and "on Monday." The former sounds weird, but it is common in headers:

```
<HEAD>Eds: Also in Monday AMs report.</HEAD>.
```

The ability to compute df_1 and df_2 at scale can help identify artifacts like this.

Both "in Monday" and "on Monday" are reasonably frequent in this corpus (57 and 394 documents, respectively). The difference is more saliient in terms of adaptation. "On Monday" is repeated fairly often (10% of the 394 documents that it appears in), in constrast to "in Monday" (0 of 57).

Duplicate documents are quite common. Many of these duplicate documents have long repeated substrings, as illustrated in Figure 16. The code in [21] makes it easy to compute LCPs for all substrings, though that had been possible using previous methods such as [19].

Table 7. Instead of computing statistics of interest over the $N(N-1)/2$ substrings, we compute them over C classes. In practice, $C \approx N/2$, which is quite a bit better than the bound in Table 1, $C \leq N$, based on theoretical considerations.

Month	N = Text Size (millions of bytes)	C = Number of Classes (millions)
Jan	23	13
Feb	21	11
March	22	12
April	21	12
May	24	13
June	23	12
July	22	12
Aug	21	12
Sept	22	12
Oct	24	13
Nov	22	12
Dec	21	11

6 Conclusion

Substring (ngram) statistics are fundamental to nearly everything we do. The goal of this work is to make it practical to compute corpus-based statistics for all substrings (ngrams). Anything you can do with words (and short ngrams), we ought to be able to do with million-grams. Previous work [19] showed how to compute standard frequency ($freq$) and document frequency (df) for all substrings. This paper has simplified that procedure, and generalized the result. We showed how to compute cumulative document frequency, cdf_k, which encodes $freq$ and df, as well as generalized document frequency (df_k):

$$freq = cdf_1$$

$$df_k = cdf_k - cdf_{k+1}$$

These values determine the probability distribution of substrings in documents ($Pr(k) \approx df_k - df_{k+1}$), as well as summary statistics of $Pr(k)$ such as

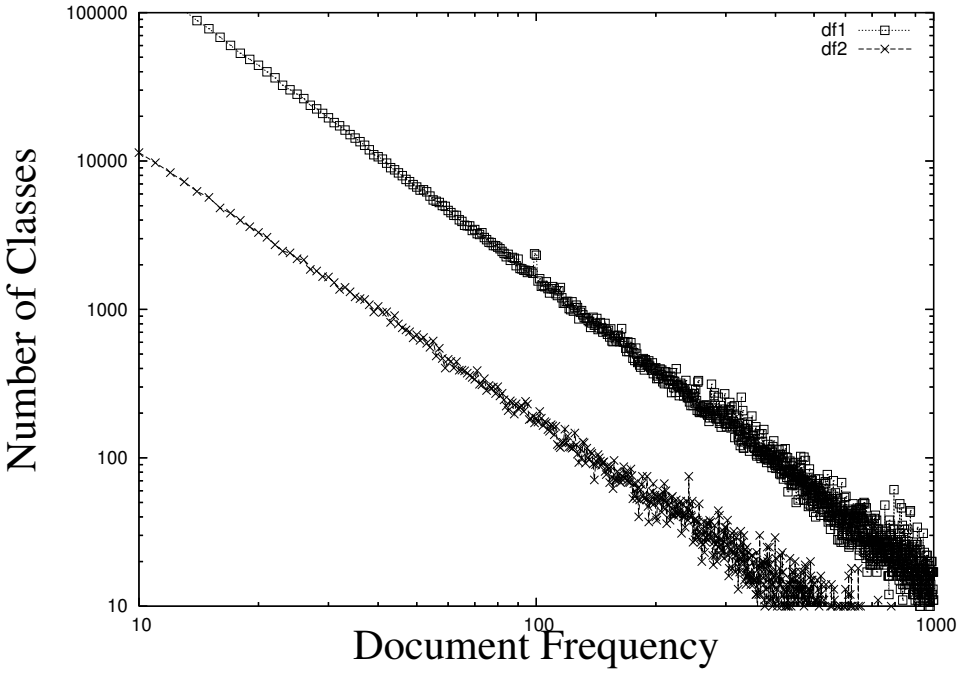

Fig. 15. df_1 and df_2 have a Zipf-like distribution (in January 1989 AP newswire). The df_1 curve is well above df_2 (especially at low frequencies).

moments, entropy, adaptation, etc. The leading terms of cdf_k are usually the largest and most important.

The computation of cdf_k started with suffix arrays. Suffix arrays make it easy to determine the frequency and location of a particular substring. To compute those statistics and more for all substrings, we grouped the N^2 substrings into N equivalance classes.

Classes are defined in terms of intervals on the suffix array: $< i, j >$. The set of substrings in a class is: $Class(< i, j >) = \{$ substrings $str|str$ starts every suffix in $< i, j >$ and no others $\} = substr(text, s[i], k)$ where $LBL < k \leq SIL$. Equivalent substrings, e.g., "to" and "to be" in the "to be or not to be" example, have identical distributions; wherever "to" appears in the corpus, "to be" is sure to follow.

Generalized document frequency can be computed directly from the classes

$$cdf_k(< i, j >) = \sum_{i \leq w \leq j} neighbor^{k-1}[w] \geq i$$

but it is more efficient to fold this calculation into a depth first traversal of the interval tree, propagating the counts from each daughter to its mother. The traversal outputs classes sorted first by i (in increasing order) and then by j

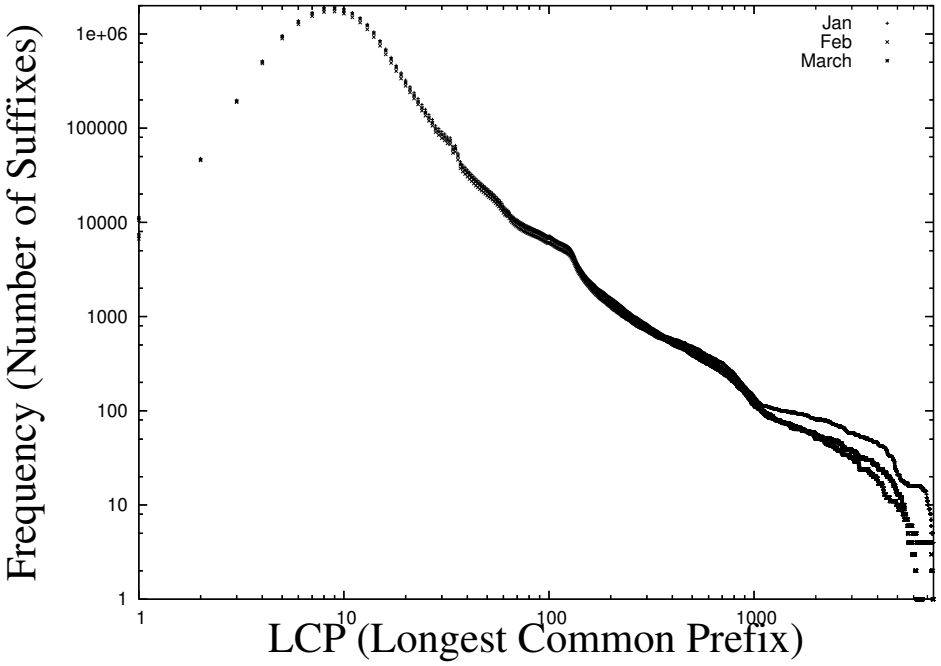

Fig. 16. The distribution of LCPs in three months of 1989 AP newswire. The distributions are remarkably similar from one month to the next. This peak is at 9 bytes, which corresponds roughly to bigrams. There are, of course, many common prefixes that extend to hundreds or even thousands of bytes. The very long LCPs tend to be associated with artifacts in the AP news such as duplicated documents.

(in decreasing order). The total order is convenient, making it possible to find classes quickly with a binary search.

References

1. David Meyer and Roger Schvaneveldt: Facilitation in recognizing pairs of words: Evidence of a dependence between retrieval operations, Journal of Experimental Psychology, Vol. 90, 227–234 (1971)
2. Karen Sparck Jones: A statistical interpretation of term specificity and its application in retrieval, Journal of Documentation, vol. 28, No.1, 11–21 (1972)
3. Ellen Prince: Toward a taxonomy of given-new information, Cole, P. Editor, Academic Press, New York, 236–256 (1981)
4. James Raymond Davis and Julia Hirschberg: Meeting of the Association for Computational Linguistics, 187–193 (1988)
5. Gerald Salton: Automatic text processing, Addison-Wesley Longman Publishing Co., Inc. (1988)

6. Guy Steele: Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO, ACM Proceedings of the 1977 annual conference, 187– 193, ACM Press (1988)
7. Timothy Bell and John Cleary and Ian Witten: Text Compression, Prentice Hall, Englewood Cliffs, NJ (1990)
8. Eugene Charniak: Statistical Language Learning, The MIT Pres, Cambridge, Massachusett (1993)
9. Udi Manber and Gene Myers: Suffix arrays: a new method for on-line string searches, SIAM J. Comput. vol.22, no.5, 935–948 (1993)
10. Donna Harman and Mark Liberman: TIPSTER Volume 1, LDC http://www.ldc.upenn.edu (1993)
11. Andrei Z. Broder and Steven C. Glassman and Mark S. Manasse and Geoffrey Zweig: Syntactic clustering of the Web, Comput. Netw. ISDN Syst., vol.29, no.8-3, 1157–1166 (1997)
12. Ian Witten and Alistair Moffat and Timothy Bell: Managing gigabytes: compressing and indexing documents and images, Van Nostrand Reinhold, New York, NY (1999)
13. Frederick Jelinek: Statistical Methods for Speech Recognition, The MIT Pres, Cambridge, Massachusett (1999)
14. Christopher D. Manning and Hinrich Schütze: Foundations of Statistical Natural Language Processing. The MIT Pres, Cambridge, Massachusett (1999)
15. Kenneth W. Church: Empirical Estimates of Adaptation: The chance of Two Noriegas is closer to $p/2$ than p^2, Coling (2000)
16. Daniel Jurafsky and James H. Martin: Speech and Language Processing. Prentice Hall, Upper Saddle River, NJ (2000)
17. Xuedong Huang and Alex Acero and Hsiao-Wuen Hon: Spoken Language Processing, Prentice Hall, Upper Saddle River, NJ (2001)
18. R. Harald Baayen: Word Frequency Distributions, Kluwer Academic Publishers, Dordrecht, The Netherlands (2001)
19. Mikio Yamamoto and Kenneth Church: Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus, Computational Linguistics, vol. 27, No. 1, 1–30 (2001)
20. Yinghui Xu and Kyoji Umemura: Improvements of Katz K Mixture Model, Information and Media Technologies, vol. 1, no. 1, 411–435, (2006)
21. Kyoji Umemura: www.cicling.org/2009/Umemura-Church/