Roy E. Kimbrell

Searching for Text? Send an N-Gram!

Short character strings called n-grams give every document a unique signature

Here's the problem: You maintain a full-text database of all the stories printed by a large daily newspaper. With several hundred issues stored, you want to find all the stories relating to a particular subject. How do you do it?

The keyword approach is too limited; the database contains too many subjects. Artificial intelligence (AI) won't help much; machines and programs aren't that smart yet. You could use special pattern-recognition hardware, but the cost is prohibitive.

The solution to the problem might be a novel approach developed by Raymond D'Amore and Clinton Mah at PAR Government Systems Corp. in McLean, Virginia. Their technique is simple, elegant, and it works. It uses pieces of words, which they call *n-grams*.

Fingerprinting Documents

An n-gram is a sequence of a specified number of characters occurring in a word. For example, the two-character n-grams (or 2-grams) in the word "duck" are "du," "uc," and "ck." An n-gram vector is a list of the n-grams found in a document and the number of times each was found, as shown in figure 1.

To set up a document-retrieval system using n-grams, you derive an n-gram vector for each document as you are storing it. The n-gram vector comes from the text. It is an index of the document, a unique "fingerprint" that you can use to identify it. To create the n-gram vector, you remove the common words from the text; then you remove the common endings from the remaining words. You count selected n-grams in the word fragments that are left and keep them in a list; that's the n-gram vector.

You then store the n-gram vector with a pointer to the location of the full-text document. You might want to store the vector along with other vectors that are similar to it.

Now you are ready to retrieve documents using words, phrases, or sentences that describe the subjects of interest. You can even use a sample document as a query to find others similar to it. To do this, you create an n-gram vector of the query and compare it to the vectors of the documents. The retrieval program computes the degree of similarity between the query's n-gram vector and those of the documents. When the similarity is great enough, the program selects the document, as shown in figure 2.

Beyond 2-Grams

To differentiate all but the shortest documents, counting only 2-grams is not sufficient. Some 2-grams are very common, such as "te." Others, like "qz," never occur. The common 2-grams don't have much value in indexing a document. For an index to be useful, it must differentiate between dissimilar documents. But if an n-gram occurs often in every document, it doesn't tell you anything. Those that don't occur at all also have no value because they don't tell you anything, either.

Rather than throwing away the common 2-grams, we can extend them to 3-grams. For example, rather than using "te," you would count all the possible 3-gram combinations that use "te"; that is, "tea," "teb," "tec," and so on.

D'Amore and Mah say that about 200 of the 676 possible alphabetic 2-grams (26 × 26) occur frequently enough to be candidates for extension. Unfortunately, many 3-grams are also very common; but you can extend the common 3-grams to 4-grams, and so on. Extending the n-grams improves the system's performance. However, you don't always *need* to go to 4-grams to index a document. The shorter the document, the smaller the size of the n-gram necessary to index it. Short documents of a few hundred words might need only 2- or 3-grams (as common as they are) to differentiate them from one another. For example, 2-grams alone work well enough with directories, such as telephone books.

D'Amore and Mah use about 12,000 different 2-, 3-, and 4-grams to index documents. An n-gram vector created using all these terms won't have 12,000 n-grams in it, however. The number of n-grams occurring in a document increases slowly as the number of words in it rises; a 3000-word document, for example, might have only 600 different n-grams. But if you have to keep track of 12,000 different n-grams, it would seem to make sense to use 4-

continued

Illustration by Alexa Grace © 1988

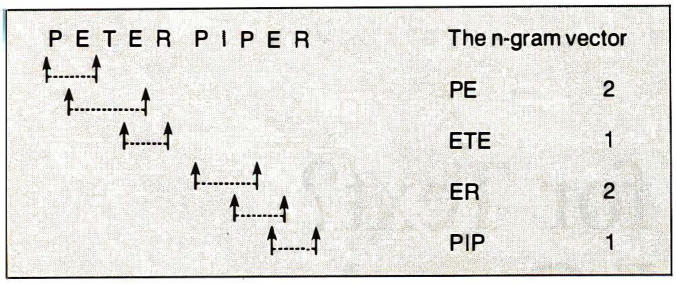


Figure 1: *In this n-gram extraction, four kinds of n-grams are included in the vector for the phrase "Peter Piper." This vector contains two 2-grams (each of which occurs twice) and two 3-grams. Note that n-grams can overlap.*

grams instead. Not so; there are too many of them. There are 676 2-grams if you use only alphabetic characters. But there are 17,576 alphabetic 3-grams, and almost half a million alphabetic 4-grams.

You can also include nonalphabetic characters in n-grams. The only ones generally useful are numbers, which occur frequently in documents and are often the subject of queries. "Commodore 64," "256K bytes of RAM," and "The Belchfire 8000 with 40 megabytes of hard disk storage" could all occur in a document and thus be the subjects of a query.

None of the numeric and alphanumeric n-grams possible are considered common. You should probably store purely numeric n-grams as 3-grams. There are only 1000 possible numeric 3-grams. Compared to the number of alphabetic n-grams, this is a relatively small number.

Cut Out the Noise

Another way to reduce the number of n-grams you need to differentiate documents is by noise reduction. Noise, for purposes of indexing, is information contained in a document that doesn't add much to your ability to find that document. Punctuation is considered noise. Common words such as *a, the, by,* and *for* are also noise.

Table 1 contains a list of 258 of the most commonly used words in American English. They comprise about 55 percent of the words used in the written language. Your database may have different common words. (If your database consists of articles about computers, for example, such words as *mother* and *father* might occur less frequently.) Because they are so common, these words add little information to text analysis. However, you must carefully consider their elimination. *Mother, father, children,* and *school* might be quite common in some contexts, but eliminating them might remove important information, particularly in an academic or sociological-factors database. Some words, too, are homonyms. *Begin,* the name of the former Israeli prime minister, is spelled the same as *begin.* Eliminating *begin* also, unfortunately, eliminates *Begin.*

Listing 1, COMMON.C, provides an efficient method for recognizing these common words. A large table at the end of COMMON.C (not shown in listing 1) was taken from a spelling checker written in Pascal (thus the need for the program to offset subscripts by 1). It contains an array of structures, each with a single character, a `wordend` flag, a `next` index, and an `alt` index.

A word fragment enters the table by converting the first character in the word to an index. This is simple: *a* is 1, *b* is 2, and so on. The `next` index for this entry is taken to be the current index. From this point, the characters of the word aren't used as indexes; they're just compared to the characters in each table entry. The function now works as in table 2. This method is fast, and you can expand it to include more words. You can also compact the table if you wish.

You can even consider common word endings to be noise. *Ended* and *endings* have the same root: *end.* If you reduce both words to their common root, you eliminate superfluous differences, and the similarity measurements will improve. This process of stemming a word down to its root is known as *conflation.*

Digging Up the Roots

Whenever a program must extract meaning from individual words of English text, the word forms are often conflated—that is, normalized or transformed into a simple, common form. Words such as *civilization, fishing,* and *halted* are transformed into their basic forms: *civilize, fish,* and *halt.* At least, that is the goal. To accomplish this, you need a set of rules similar to those used in knowledge-based systems. If you use only a few (20 or 30) rules, some transformations won't be accurate. *Civilization* might truncate to "civiliz." A word like *the* truncates to "th," as do *they* and *these,* when such words really shouldn't be conflated at all.

To deal accurately with this problem, you need 1000 or more rules that specify, for the most part, exceptions and special cases. When very high precision isn't necessary, a few rules

continued

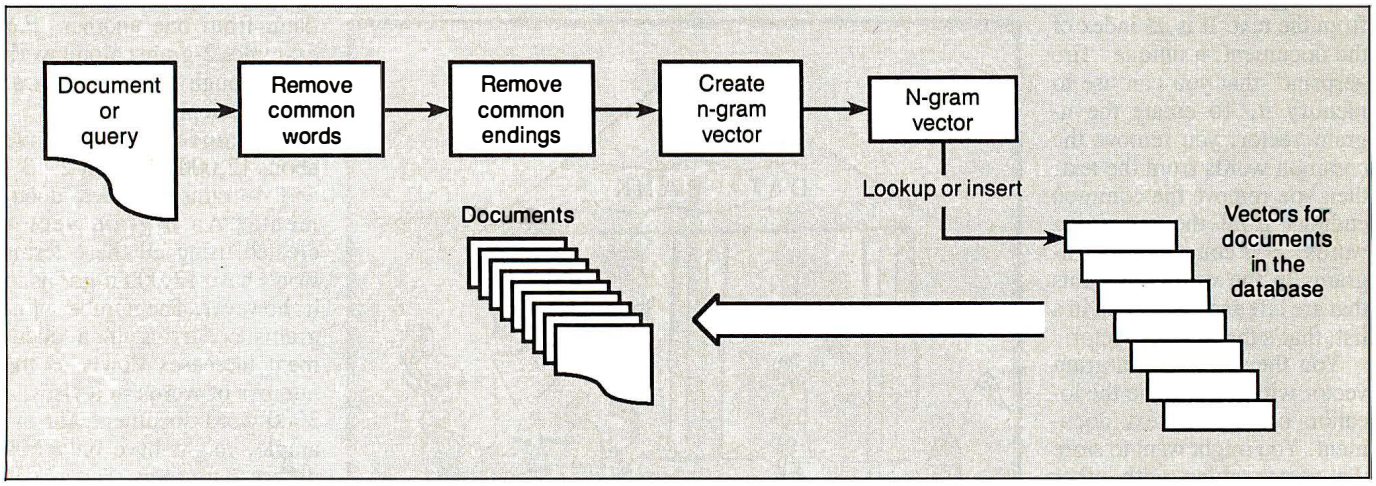


Figure 2: *An n-gram vector is a list of the n-grams in a text document, minus all common words and common endings. The n-gram vector is stored with a pointer to the location of the full-text document. By comparing the n-gram vector of a query to those of stored documents, you can find documents likely to contain what you're looking for.*

Listing 1: *COMMON.C, a C program used to recognize common words.*

```
/* this structure holds a directed graph
   used to recognize common words */
#define GRAFSIZE  405
struct {
  char c;
  char wordend;
  int next;
  int alt;
    } stopgraf[GRAFSIZE] { }
/* The data used to initialize stopgraf is
   given later
*/
int stopword(word,wl)
     char word[];
     int wl;   /* word length */
/*
     stopword enters the common word graph
     with the value in word[]; if word[] is
     in the graph, return 1, else 0
*/
{
int j=0, p;
  p = word[j] - 'a' + 1;  /* the first
           entry is 1, not zero */
  while (j < wl-1 && p) {
    j++;
    p = stopgraf[p].next;
    while (p && stopgraf[p].c < word[j])
                p = stopgraf[p].alt;

    if (stopgraf[p].c != word[j]) p = NULL;
    }
  return(p && stopgraf[p].wordend);
  }
```

will suffice, and the odd cases don't really matter.

The C function in CONFLATE.C (see listing 2) does a simple-minded job of conflation. The table `endings` is an array of structures that contains the rules. Each structure has the text of a word ending, its length, a possibly zero-length replacement string, *its* length, and the index of the next table entry so the program can check whether the current word ending matches one stored in the structure.

A word enters the table at `ending[0]`. If the word ends in "ably," the program truncates it. The process repeats, starting at `ending.next`. However, if the word doesn't end in "ably," then "ibly," "ily," "ss," and so on are checked in order. The checking (and replacing) continues until the program reaches the end of the table. Notice that there are three endings where the check string is zero-length. The first two are traps to prevent falling through to the lower part of the table. The last terminates execution of the function.

For example, the word *readabilities* passes the first few rules and then matches "ies." The program removes "ies" and replaces it with "y." The word becomes *readability*. Matching then continues at `ending[14]`. The program matches and removes "ability." The word becomes *read*. The `endings` table is checked at location 24, but matching fails to the end. Notice that these rules don't always create reasonable stems. *Movabilities* conflates to "mov," *invisibilities* to "invis." However, the program also conflates *move* and *invisible* to "mov" and "invis." In other words, all forms of the same word conflate to the same root, although the root may not be the one you'd expect.

Useful N-Grams: Worth the Weight

Rare n-grams are more useful in discriminating between different documents than are common ones because the rare ones are weighted more heavily during similarity computations. The system's performance determines the weighting scheme used. D'Amore and Mah have found the following to work well: $W_i = 1/\sqrt{P_i}$ where W_i is the weight to be used for n-gram number i,

continued

Table 1: *This list contains 258 of the most common words in American English. Such words fail to help distinguish documents from one another, and removing them makes documents' n-gram vectors more unique.*

a	before	each	got	just	more	or	should	things	well
about	began	earth	great	keep	most	other	show	think	went
above	being	end	had	kind	mother	our	side	this	were
after	below	enough	hand	know	Mr.	out	since	those	what
again	between	even	hard	land	much	over	small	thought	when
air	big	ever	has	large	must	own	so	three	where
all	both	every	have	last	my	page	some	through	which
almost	boy	eyes	he	left	name	paper	something	time	while
along	boys	far	head	let	near	part	sometimes	times	white
also	but	father	help	life	need	parts	soon	to	who
always	by	feet	her	light	never	people	sound	together	why
an	called	few	here	like	new	picture	still	too	will
and	came	find	high	line	next	place	story	took	with
animals	can	first	him	little	night	put	study	two	without
another	children	following	his	live	no	read	such	under	word
any	come	food	home	long	not	right	take	until	words
are	could	for	house	look	now	said	tell	up	work
around	country	form	how	looked	number	same	than	us	world
as	day	found	I	made	of	saw	that	use	would
asked	days	four	if	make	off	say	the	used	write
at	did	from	important	man	often	school	their	very	year
away	different	get	in	many	old	second	them	want	years
back	do	give	into	may	on	see	then	was	you
be	does	go	is	me	once	sentence	there	water	your
because	don't	going	it	men	one	set	these	way	
been	down	good	its	might	only	she	they	we	

and P_i is that n-gram's probability of occurrence.

You can calculate the probability that a specific n-gram will occur by counting all the occurrences of that n-gram in a large, representative body of documents. Then you divide this count by the total number of n-grams. You must calculate a weight for each n-gram used as an indexing term, but you do it only once; after that, the weight is a constant you look up in a table.

Computing Similarity

To determine the similarity between two n-gram vectors, you multiply the frequencies of corresponding n-grams by their weights and sum the results. When two n-gram vectors are dissimilar, the sum of the products of the corresponding frequencies is small: Where one vector has some of a particular kind of n-gram, the other hasn't any or has only a few. When the two numbers are multiplied, the result is zero or a small number. If two n-gram vectors are similar, they have more of the same n-grams, and the result is larger.

There is a scale of similarity, then, from small similarity

continued

Table 2: *The logical operation of the function in listing 1 after establishing a beginning current index.*

While there is a current index and you haven't run out of characters in the word,
  Get the next character from the word.
  Set the current index to the *next* value.
  While you have a current index and your current character in the word is greater than the character stored in the current table entry,
    Set the current index to the value at *alt*.
  Endwhile.
  If the character at the entry for the current index isn't the same as the current character in the word, you don't have a common word.
Endwhile.
When you finally come to the end of the word, if the current table entry's *wordend* flag is set and you haven't otherwise eliminated this word, it is a common word.

Listing 2: *CONFLATE.C. This routine will stem a word down to its root.*

```
/* A Conflating Function in C */

#define LT -1
#define EQ  0
#define GT  1
slteqgt(s1,s2)
  unsigned char *s1, *s2;
/* compares two strings */
{
  for(;;){
    if (*s1 < *s2) return(LT);
    if (*s1 > *s2) return(GT);
    if (*s1 == *s2 && !*s1) return(EQ);
    s1++; s2++;
    }
  }

/* the following are locations in
   the conflation table */
#define SSend    3
#define Eend    10
#define IONend  12
#define ARYend  14
#define ABLend  20
#define IVend   22
#define ATend   23
#define ISend   24
#define FIN     27
#define ENDINGS 28
struct{
  char *ending;  /* ending string */
  int   offset;  /* length */
  char *replace; /* replacement */
  int   replen;  /* length */
  int   next;    /* goto */
    } endings[ENDINGS] {
       "ably",    4,  "",    0,  ISend,
       "ibly",    4,  "",    0,  FIN,
       "ily",     3,  "",    0,  SSend,
/*SS*/ "ss",     2,  "ss",  2,  FIN,
       "ous",     3,  "",    0,  FIN,
       "ies",     3,  "y",   1,  ARYend,
       "s",       1,  "",    0,  Eend,
       "ied",     3,  "y",   1,  ARYend,
       "ed",      2,  "",    0,  ABLend,
       "ing",     3,  "",    0,  ABLend,
/*E*/  "e",       1,  "",    0,  ABLend,
       "al",      2,  "",    0,  IONend,
/*ION*/"ion",     3,  "",    0,  ATend,
       "",        0,  "",    0,  FIN,
/*ARY*/"ary",     3,  "",    0,  FIN,
       "ability",7,  "",    0,  ISend,
       "ibility",7,  "",    0,  FIN,
       "ity",     3,  "",    0,  IVend,
       "ify",     3,  "",    0,  FIN,
       "",        0,  "",    0,  FIN,
/*ABL*/"abl",     3,  "",    0,  ISend,
       "ibl",     3,  "",    0,  FIN,
/*IV*/ "iv",      2,  "",    0,  ATend,
/*AT*/ "at",      2,  "",    0,  ISend,
/*IS*/ "is",      2,  "",    0,  FIN,
       "ific",    4,  "",    0,  FIN,
       "olv",     3,  "olut",4,  FIN,
/*FIN*/"",        0,  "",    0,  FIN+1};

stem()
/*  if the ending of word[] is in
   endings.ending, it is removed and any
   replacement string is tacked on the
   end; search and replacement is
   controlled by endings.next */

{
int i;
extern char word[];
extern int wl;
  i = 0;
  while(i<ENDINGS){
    if (slteqgt(&word[wl-
                endings[i].offset],
          endings[i].ending) == EQ){
      cpystr(
           &word[wl-endings[i].offset],
           endings[i].replace,NULL);
      wl += endings[i].replen -
            endings[i].offset;
      i = endings[i].next;
      }
    else
      i++;
    }
  }
```

False similarity can occur when the similarity threshold is too low. You're less likely to miss a document but more apt to get dissimilar ones.

values to large ones. When responding to queries against a set of n-gram vectors for documents, you must determine the threshold above which you wish to select a document and below which you wish to reject it. Figure 3 represents the system's ability to discriminate between text items using similarity values.

You can get raw similarity values by multiplying corresponding n-gram counts and weights and adding the products. The size of these raw values depends as much on the n-gram vector size as on the counts in the vectors. That is, two documents might be equally similar to a third, but similarity computations will probably produce different values. The longer document will probably have a longer n-gram vector (because of the greater chance for having some of the rarer n-grams in it). In a longer vector, there is a greater opportunity for matching corresponding n-grams in another vector during similarity computations. This means the similarity value will be larger.

The method for reducing the similarity values to a common measure, called the normalization process, is a little complicated. It requires that you compute an estimate of the standard deviation and the expected value of the similarity values. The standard deviation is a measure of the variability of raw similarity values, and the expected value is a mean, or average, value.

Many of the values needed to compute the standard deviation are constants for a particular set of n-gram indexing terms. In addition, you need the total number of n-grams counted in each vector (the lengths of the vectors). For the formulas to use in normalizing the raw similarity values, see the text box "Making It Work."

The n-gram system is large and complicated and can malfunction. A malfunction occurs when a similarity computation produces a value unexpectedly large enough to cross whatever similarity threshold you have set for document selection. Few, if any, of the words from the query might actually appear in the document. When a query contains mostly common n-grams, the chance for false-similarity matching is relatively high. This happens more frequently when you use only 2-grams or 3-grams as indexing terms. The purpose of extending the n-grams to longer strings is to reduce their frequency and therefore the chance of false similarity.

False similarity can also occur when the similarity threshold is set too low. This reduces the chance of missing a document, but increases the chances of getting documents that don't apply to the query. In a mature system, false similarity can be well controlled and is relatively rare.

Using a Thesaurus

Synonyms can be a problem, particularly in short documents. For example, in a newspaper story about an aircraft accident, the word *airplane* might never appear. Instead, words such as *craft, jet,* and *Boeing 747* might be used. Further, *mishap* might not appear, while *accident* or *crash* does. In other words, a query of "airplane mishaps" might fail to produce this story from the database.

To circumvent this problem, you can implement a thesaurus containing groups of words with similar or related meanings as well as synonyms. You create an n-gram vector for each word group. You only need to keep the vectors on-line; you don't need to use the words themselves during similarity matching. You can now compare the query to the n-gram vectors representing the thesaurus. Those vectors that are similar to the query probably contain some of the words in it. Then you can use the query's n-gram vector and the thesaurus's n-gram vectors that are similar to the query and compare them to the documents' n-gram vectors. A similarity above the threshold indicates which documents to retrieve.

Creating the thesaurus is no small task. There are a lot of words to collect into groups and a lot of decisions to make. Your

continued

	Peter Piper picked a peck of pickled peppers	How many pickled peppers did Peter Piper pick?	Pied Piper of Hamlin	Peter Piper	'Twas brillig, and the slithy toves did gyre and gimble in the wabe
Peter Piper picked a peck of pickled peppers	31.5	30.9	16.4	26.0	−2.2
How many pickled peppers did Peter Piper pick?	30.9	28.2	17.2	33.8	−1.9
Pied Piper of Hamlin	16.4	17.2	32.9	23.5	−9.9
Peter Piper	26.0	33.8	23.5	48.9	−1.7
'Twas brillig, and the slithy toves did gyre and gimble in the wabe	−2.2	−1.7	−0.9	−1.9	20.8

Figure 3: *This table shows the similarities between five phrases, four of which resemble one another. The similarities were computed using the method shown in the text box "Making It Work." Higher values indicate greater similarity, while lower (or negative) numbers indicate dissimilarity. If, for example, you set the threshold to 25 and the query was "Peter Piper," the system would select the first two phrases. If the query was "Pied Piper of Hamlin," the system would select neither of those phrases.*

Making It Work

Step 1: Select the 10,000 to 15,000 2-, 3-, and 4-grams to be used as potential members of n-gram vectors. If the documents are short (such as telephone directories), you may need only 2- and 3-grams, or even just 2-grams.

One way to select the n-grams is to find a large body of text representative of the text you want to store and search. Count the various 2-grams in it. Take the 200 or so most common 2-grams and add characters (at the end). Each common 2-gram will expand to 26 3-grams. Count the number of 3-grams that occur in your representative documents. Expand the 150 or so most common 3-grams to 4-grams. (If you want somewhat better performance, you can also expand the 100 most common 4-grams to 5-grams.)

If your documents are anything like ordinary text, you will end up with about 12,000 n-grams. You won't find many of the expanded n-grams in your representative text; they are either nonsense or very rare—you won't be able to tell which in most cases. For these n-grams, just assign an arbitrary count of 1.

Step 2: Compute the probability of occurrence of each n-gram in the indexing set you've just created. This is the number of times the n-gram was found divided by the total number of n-grams counted.

Step 3: Compute weights for each n-gram. The weight is used to emphasize rarer n-grams and deemphasize more common n-grams when computing similarity. They improve the performance of the system.

D'Amore and Mah found that the following formula works well: $W_i = 1/\sqrt{P_i}$ where i indicates an individual n-gram. W_0 is the weight for the first n-gram in the set, W_1 is the weight for the second, and so on; P_i is the probability computed for the individual n-grams.

Step 4: Compute the following constants (they will be used during the calculation of similarity values):

$C_0 = sum_i[W_i * P_i^2]$
$C_1 = sum_i[W_i * P_i^3]$
$C_2 = sum_i[W_i * P_i^4]$
$C_3 = sum_i[sum_j[W_i * P_i^2 * W_j * P_j^2]]$
where i not $= j$

The *sum* here means "compute the value inside the brackets for each n-gram and add up the values." In the last, C_3, the weight times the probability squared for each n-gram is multiplied by the weight times the probability squared for every other n-gram. That is, the first is multiplied by the second, third, fourth, etc.; the second by the third, fourth, etc.; and so on. The values for all these multiplications are added together.

Step 5: Create an n-gram weight table. This table will contain the n-grams to be used in n-gram vectors and the associated weight for each n-gram. The table will be large, so storage and lookup might be a problem. While creating the n-gram vector for a document, n-grams are created and looked up in the table. If an n-gram is in the table, it is counted. When computing similarity, an n-gram's associated weight is used.

Step 6: Implement the algorithm to create an n-gram vector. (This is not the optimal way, but it is a simple way.) Scan each word in the text. Try the longer n-grams before the shorter ones. You can do this by sliding a window across the word. At first the window is four characters wide. Look this up in the n-gram weight table. If you find this n-gram, count it. If you don't, narrow the window to three characters and try again. Keep narrowing and trying until you find a countable n-gram. Expand the window to four characters again, shift it to the right, and continue looking for countable n-grams. When narrowing the window, be sure you don't narrow the window so much that it falls completely within the previous window. It should (if possible) overlap the previous window, but extend outside it as well.

The n-gram vector is just a list of the n-grams found and their counts. Rather than saving the character-string representation of the n-gram, you might want to save its index in the n-gram weight table. This makes it easy to compare two n-gram vectors and to look up their respective weights.

Step 7: Implement the algorithm to compute the similarity between two n-gram vectors and thus the similarity between a query and a document or between two documents. Start by computing R, the raw similarity value between them. If you have vectors a and b, then $R = sum_i[W_i * N_i^a * N_i^b]$ where W_i is the weight for each n-gram and N_i^a and N_i^b are the counts for the individual n-grams in each of the n-gram vectors (a and b are superscripts, not powers).

Step 8: Implement the algorithm to normalize the raw similarity value. Because the size of the raw value will depend on the relative sizes of the source documents for the vectors, you have to compensate for the document sizes. You do this by subtracting the expected similarity value from the raw similarity value and dividing by the estimated standard deviation of the raw similarity value.

The formula for the expected similarity value is $E = T^a * T^b * sum_i[W_i * P_i^2]$ where T^a is the total number of n-grams in the a vector, and T^b the total number in the b vector.

The formula for the standard deviation squared is $D^2 = T^a * T^b * (C_0 + (T^{ab} - 2) * C_1 - (T^{ab} - 1) * C_2 - (T^{ab} - 1) * C_3)$ where T^{ab} is $T^a + T^b$ and C_0, C_1, C_2, and C_3 are the constants computed in step 4.

The normalized similarity between the two vectors is then $S = (R - E) / D$. The normalized similarity values computed in this fashion seem to be stable and can be compared to one another and to constant thresholds.

Optional Step: Rather than using raw n-gram counts, in each step where n-grams are counted, you can substitute the square root of the count. This transformation seems to improve performance somewhat.

best bet might be to build the thesaurus a little at a time, as problems appear.

Natural Clusters

One alternative to the thesaurus is clustering. There is a natural tendency for documents with related subject matter to have similar n-gram vectors. You can look at one document as a complicated query and other, similar documents as the results of that query. Those documents that are similar to one another are clustered. You can create an n-gram vector for a cluster of documents by adding the corresponding n-gram counts in each vector to create a new vector that represents the cluster of documents.

When one document in a cluster is selected because the simi-

continued

larity between its vector and that of the query is greater than the threshold, the rest of the documents in that cluster are better candidates for selection. You might want to reduce the similarity threshold for these other documents so more of them will be selected. This helps prevent the synonym problem. However, if only one or very few of the documents in the cluster exhibit any similarity to the query, it is probably a spurious match. Using clustering can avoid false matching and missing valid documents.

When a database contains many documents, computing similarity between a query and each of the vectors can take a long time. Clustering can reduce the search time. Instead of scanning

continued

N-Gram Vectors in C

NGRAM.C in listing A is a central fragment of a real n-gram vector generator. The technique used here to extract n-grams from the text isn't used in practice; it's too slow. D'Amore and Mah use a highly optimized set of bit maps and tables to identify the n-grams to be used as indexing terms and to compute an index into a table of weights. This index also serves as a short, unique identifier for later use in an n-gram vector. Despite this deviation, this fragment of a program explains how to extract n-grams from text.

The program first defines a few constants and static variables. `MAXNGLEN` and `MINNGLEN` define the longest and shortest n-grams considered, respectively. The structure `NGDATA` defines an element of the `ngrams` array of 108,000 bytes that contains the n-gram strings used as index items and their weights.

The purpose of `ngfind` is to extract n-grams from words. The word (`word`) and word length (`wl`) are inputs to `ngfind`, which uses a variable-size window to frame possible n-grams. Its rules are:

- Try a maximum-size window first. The maximum size is `MAXNGLEN` or the word length, whichever is shorter. Look up this size n-gram in the `ngrams` array using the `lookup` function.
- If `lookup` returns 0, shorten the window from the right and continue to look up n-grams.
- If you can't find an n-gram of length `MINNGLEN` (this is considered an error), shift the beginning of the window to the right and expand it to maximum length. The shortest n-gram should be a 2-gram, and if you can't find any longer n-grams, you should at least be able to find a 2-gram.
- When you find an n-gram, shift the window one position to the right and expand the window to maximum length.
- Don't allow the current window to fall completely within an older window—that is, don't look at the same data twice. When the end of the current window falls within the previous one, move its start to the right.

When you call `lookup`, it must find the input string in `ngrams`. This is not a trivial task. Several methods are usable—B-trees, hashing, and so on. I used a hashing technique. The `ngrams` array is static (lookups only), so you can change hashing parameters until you obtain an optimal storage profile. The value returned by `lookup` is either 0, if the n-gram is not found, or the address of the string equal to the input n-gram string in the `ngrams` array, if it is found.

The function `ngcount` uses the return value from `lookup` as an address in `vector`. This works fairly well because `ngrams` is much larger than `vector`. There are many more locations in `vector` than you need, but this unused space is your trade-off for speed.

In `main`, `init` initializes `ngrams`. The function `getword` obtains the next word from the input stream. All characters have been converted to lowercase, and all but alphabetic and numeric characters have been converted to spaces. The function `stopword` returns a 1 if `word` is common; `stem` conflates `word`. Finally, `outvec` writes the completed vector to a temporary file.

Listing A: *NGRAM.C, a fragment of a vector generator in C.*

```
#define MAXNGLEN  4
#define MINNGLEN  2
#define VECTORSIZE 12000
unsigned vector[VECTORSIZE];
#define WL  43
char word[WL];
/* longest English word
   is 42 chars */
int wl = 0; /* word length */
#define NNGRAMS 12000
struct NGDATA {
       char ngram[MAXNGLEN];
       float weight;
       } ngrams[NNGRAMS];
int ngcount(gramid)
        unsigned gramid;
{
vector[gramid]++;
}
int ngfind(word,wl)
        char word[];
        int wl;
{
int oldend, len, start;
unsigned gramid;
if (wl < MINNGLEN) return;
oldend = wl;
len = MAXNGLEN;
if (len > wl) len = wl;
start = 0;
for(;;){
 if (gramid =
  lookup(&word[start],len)){
   ngcount(gramid);
   oldend = start + len;
   if (oldend == wl) return;
   start++;
   len = MAXNGLEN;
   if (start + len > wl)
        len = wl - start;
   }
   else{
      if (len == MINNGLEN){
      /* didn't find 2-gram */
        start++;
        len = MAXNGLEN;
        if (start + len > wl)
            len = wl - start;
        if (len < MINNGLEN)
            return;
        }
  else{
      len--;
      if (start-len <= oldend)
          start++;
          }
      }
  }
}
main()
{
init();
while(wl=getword(word)){
      if (!stopword(word,wl)){
          stem(word,wl);
          ngfind(word,wl);
          }
     }
outvec();
}
```

individual n-gram vectors during a search, you look at the vectors representing the clusters. When a cluster is similar to a query, you can either retrieve each document in the cluster or scan the vectors of the cluster documents for similarity.

To place a new document in a cluster, you must scan existing cluster vectors for similarity and, when you find one that meets or exceeds your threshold, add the n-gram counts in the new vector to the existing cluster vector. If you can't find a similar cluster, you can create a new one.

Fine-Tuning

Once you select a document's n-gram vector, you can retrieve the document. In a large database, you may select many documents. A few may have similarity with only part of the query or may be completely spurious. Rather than present the documents immediately, you can rescan each of the selected documents, eliminating the common words and stemming the rest. This time you compute n-gram vectors for individual words and compare them to the query's n-gram vector. This is a rapid process because a vector for a single word will be short; there are only a few n-gram types and therefore only a few multiplications to do. If there is sufficient similarity, the program considers the word significant and displays it with the document's identification. Then you can see how close the document comes to satisfying the query and choose which documents to select.

You can tune the retrieval operation to ignore mild misspellings in either the queries or the documents. Dropping a character or transposing two characters, for example, "speling mistkaes," is considered a mild misspelling. If, during the search, you lower the similarity threshold a bit, you will select documents with word variations. Some n-grams will match, though probably not the misspelled ones. If you have done some form of stemming, the word variations will not be due to grammatical differences, but to misspellings.

The Theoretical Model

D'Amore and Mah developed a model based on these concepts to convince doubters that these methods are valid and to predict the performance of new systems.

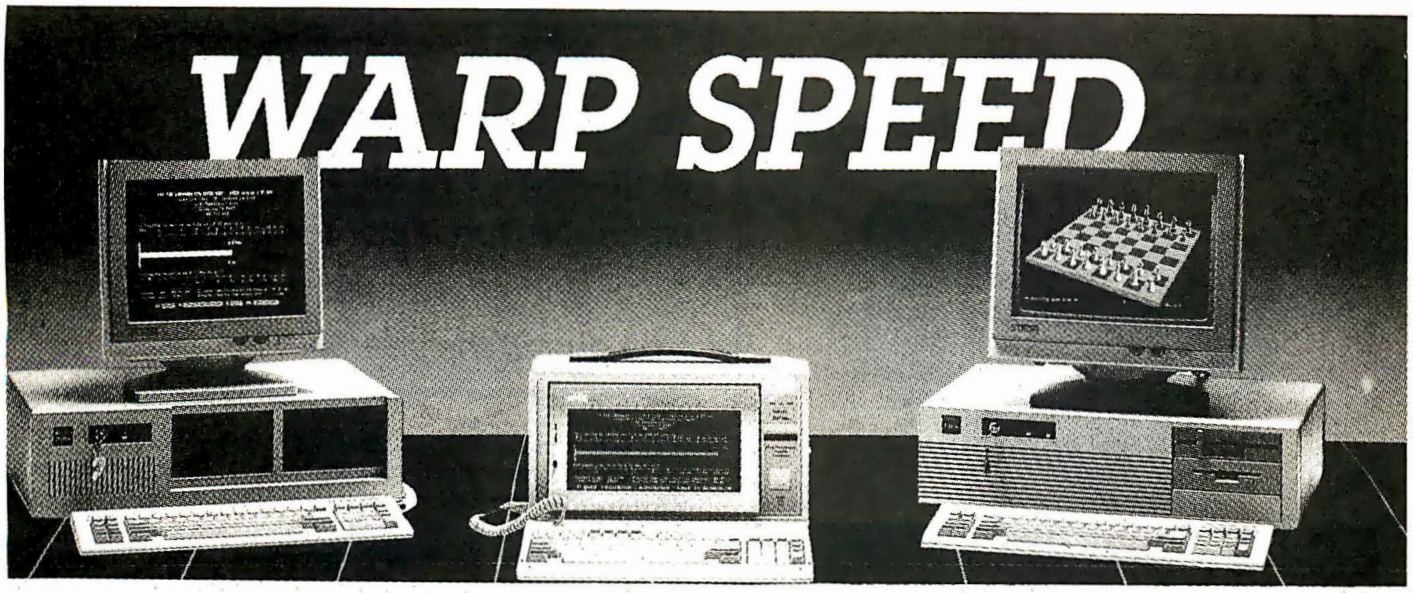
In testing their system, D'Amore and Mah used a variety of documents: about 1700 from the Associated Press, 1200 from the *New York Times*, 3100 from the Foreign Broadcast Information Service, 2800 physics abstracts, a few articles on exotic fuels, 700 articles from the Unix news network (with articles on AI, ham radio, the space shuttle, and others), and 300 miscellaneous messages from the Reuters wire service—altogether almost 10,000 documents. Document size varied from 700 words for the physics abstracts to nearly 12,000 words for the articles on exotic fuels. The average size was 2200 words each.

They used about 12,000 n-grams of various lengths to index the documents. They started with one document and gradually increased the size of their database. As they added documents, the model counted the number of unique n-grams it encountered. This number was proportional to the logarithm of the number of documents in the database.

Much of the work went to developing a theoretical framework for characterizing the statistical properties of n-gram indexes. This is important because D'Amore and Mah wanted to be able to describe the noise in n-gram indexing and calculate an n-gram vector's relevance to a document. This is critical if you are to retrieve documents using an n-gram vector created from a query and to collect similar documents.

Based on some assumptions about how text is generated, they described the statistical distribution of n-grams mathematically. Using this distribution, you can compute the similarity between two text items and the statistical significance of that similarity.

continued



30 Day Risk-Free Trial • On-Site Service Available

The Fastest PC's in the Galaxy, at down to Earth Prices

12 Slot Power 286/386

We put the 80286 on a card so you can upgrade to the 80386 whenever you're ready. Most of our corporate accts. prefer this American-designed, industrial quality machine. Why? 12 slots, 240 watt power supply, 4 drive openings & ruggedized construction. Tower, rack-mounted and motherboard versions available.

50MB, mono. system **$1895**

386 Speed, 286 Price

12MHz, zero wait states with 1 to 1 controller and Power Optimiser software makes this AT compatible the fastest at any price. Landmark Speed Test rates this 286 at 15.6MHz! The data transfer rate is 4 times faster. A complete system with 1MB RAM, monitor and 50MB, 33ms hard disk is only:

50MB **$1950**

12MHz 1 wait state, 640K, 50MB, 101 Keyboard, 1.2MB **$1695**

Passport 286 & 386

New large screen.

Micro 1 introduces the Passport 286 & 386 portables that keep pace with the best Compaq has to offer. 12, 16 or 20MHz and up to 8MB of zero wait state memory gives you more power than most desktop computers. Both the 11", 640 x 400 resolution backlit, supertwist, LCD screen and gas plasma screen are far more readible than typical laptop screens. External EGA and VGA output optional. 1 to 1 interleave controllers and 50 to 150MB drives optional. 200 watts, 6 slots standard.

386-16 1MB, 20MB **$3295**

286-12 640K, 20MB **$1995**

30 MHz Performance

The Power 386 outruns the IBM Model 80 and Compaq Deskpro 386/20. How do we do it? 20MHz, zero wait state with 64K of static RAM cache and ultra-high speed ESDI hard disks with 1 to 1 interleave buffered controllers. Add to this our special Power Optimiser software that accelerates reads and writes by 300% and you've got the best that money can buy, at 2/3 the price! Landmark SpeedTest rating 30MHz.

UNIX systems with DOSMERGE now available. Ask about our Tower!

16MHz (24MHz Landmark) 1MB Complete system with 50MB and monitor. **$2895**

Call for price on 20MHz and other configurations.

"Micro 1's clones are designed for industrial use . . . none are so well made"
Paul Muller, Ford Aerospace, Palo Alto, CA

Ask our customers about our quality, service and prices:

American Express
Anheuser-Busch
Associated Press
Boeing Aerospace
Clorox
Coca-Cola
Dean Witter
Ernst & Whinney
Federal Communications Com.
Ford Aerospace
Heath Zenith
Honeywell
Intel
Itel
ITT
Locheed Missiles & Space
Lucas Film
Martin-Marrietta
MasterCard
McDonnell Douglas
MCI
NASA
Pacific Bell
Rockwell International
Siemens
Tandem Computers
TRW
United Airlines
University of Calif.
Wells Fargo
Westinghouse

MICRO 1
557 Howard St.
San Fancisco, CA 94105
Tech Support: (415) 974-6997
Fax: (415) 974-6996

To order call toll free:

1-800-338-4061

In California call

(415) 974-5439

Circle 326 on Reader Service Card

You can tune the retrieval operation to ignore mild misspellings in either the queries or the documents by lowering the similarity threshold a bit during the search.

To validate their model, D'Amore and Mah took pairs of vectors from random text items to approximate a noise level. They also took pairs from different segments of the same text item to estimate the difference between similar vectors (those from the same text item) and dissimilar vectors (those randomly chosen). They conducted many such experiments and calculated statistical measures for each batch to compare against the model's predictions.

In their words: "The statistical model was validated in extensive experiments with a broad variety of text. The results were especially noteworthy because one seldom can make any good predictions about the general statistical characteristics of language. . . an n-gram description of text does contain significant information about its content." For an example of how to program an n-gram vector, see the text box "N-Gram Vectors in C."

Pluses and Minuses

There are some drawbacks to this new indexing method. First, it's complicated, in terms of both implementation and computation. Getting a new n-gram system up and running requires isolating and selecting thousands of n-gram indexing items and going through many processing steps.

Second, the n-gram method is memory- and processor-intensive. Creating a vector requires the expensive lookup of many more n-grams than there are words. Computing similarity requires many floating-point multiplications and square-root calculations. These take time, especially if you use software to do the floating-point mathematics. Without considerable optimization, looking up individual n-grams can be expensive.

Then, too, the system isn't exact. The *meaning* of the document isn't used to index it. Without this understanding, similarity computations can go astray and either find similarity where none exists or fail to find it when it does exist. Trying to prevent this adds even more complexity to the system.

There are, however, good reasons for using n-gram indexing. For one thing, it works. I know of no better method for doing what n-gram indexing can do. Keyword solutions, the next best thing, are highly limited. Searching thousands of keywords is computationally more intense than n-gram indexing, and the system is biased toward whatever keywords you use. Appropriate keywords may also be inadvertently omitted. When you add new keywords to the system, you must reindex the database. Scanning the entire text to answer queries is costly in terms of time and equipment and doesn't work as well as n-grams do.

The n-gram indexing system is adaptable to several different situations, and you don't need to reindex the system to answer completely new questions. ■

[Editor's note: *Source code listings for COMMON.C, CONFLATE.C, and NGRAM.C are available in a variety of formats. See page 3 for further details.*]

Roy E. Kimbrell is a senior programmer/analyst with Planning Research Corp. in Bellevue, Washington. He has master's degrees in computer science and meteorology.

 Circle 258 on Reader Service Card