N-Gram-Based Text Filtering For TREC-2

William B. Cavnar
Environmental Research Institute of Michigan
P.O. Box 134001
Ann Arbor, MI 48113-4001
cavnar@erim.org

Abstract

Most text retrieval and filtering systems depend heavily on the accuracy of the text they process. In other words, the various mechanisms that they use depend on every word in the text and in the queries being correctly and completely spelled. To get around this limitation, our experimental text filtering system uses N-gram-based matching for document retrieval and routing tasks. The system's first application was for the TREC-2 retrieval and routing task. Its performance on this task was promising, pointing the way for several types of enhancements, both for speed and effectiveness.

1.0 Background

Even though modern character recognition techniques are steadily improving, scanning and recognizing characters in paper documents still results in many errors. These errors can impact the ability of a standard, classical text retrieval system to process the resulting ASCII text by interfering with word stemming, stopword identification, or even basic indexing. Furthermore, as paper documents age, their print quality can degrade, possibly causing further recognition problems. This is becoming a serious problem, especially given the vast amount of material that exists in a paper form that has a limited lifetime.

Another difficulty is that the texts themselves may contain spelling variations (e.g., British vs. American) or outright spelling errors. The same is true of the queries entered by a human user. Discrepancies between the spelling of a word in a document and a word in a query may prevent a match.

A third problem area is that current word stemming and stopword list construction algorithms sometimes involve a significant amount of knowledge about the documents' language. Although there are automatic approaches to acquiring this kind of knowledge, it is complicated [1]. An ideal text retrieval method would require no special language knowledge, and in fact would be able to handle multiple languages (represented, say, in Unicode) simultaneously.

All of these problems have analogs in a somewhat different problem domain, namely, reading and interpreting postal addresses on pieces of mail. For the last eight years, the Environmental Research Institute of Michigan (ERIM) has conducted research for the United States Postal Service (USPS) in automatic address interpretation. The ultimate goal of this research is to build a high-performance address interpretation unit that can reliably and efficiently read the address information on a mailpiece, and determine its correct 9-digit or 11-digit ZIP code, even if that code is not present on the mailpiece or is in error.

There are two parts to interpreting an address from a mailpiece, and both of them are difficult:

- Recognizing the characters in the address block. This is difficult because of poor print quality, poor contrast, complicated backgrounds, presence of logos and non-address information, use of proportional fonts, and a host of other problems.

- Interpreting the lines of the address. This is difficult because of spelling errors, addressing errors, unusual abbreviations, strange local addressing conventions, and errors and omissions in the USPS databases.

Our current address interpretation systems use a number of different techniques to get around these problems. Among these is the use of N-gram-based matching to find matches in postal databases. (Please see [2] and [3] for details of these systems.)

In this work, N-grams are N-character slices of some longer string. We do not use positional N-grams; i.e., what is important is whether a given string contains a particular N-gram, not where it falls in the string. We use bi-grams (N=2) and tri-grams (N=3) together in the same system. Bi-grams and tri-grams complement each other in the sense that bi-grams provide somewhat better matching for individual words, while tri-grams provide the connections between words to improve phrase matching.

N-gram-based matching addresses the difficulties mentioned above for postal addresses in ways that have application for document retrieval:

- It usually finds good matches for postal addresses even in the face of a significant number of individual character recognition errors. This is particularly true if the system can use redundant data for the search key, such as the city, state, and ZIP together. This is identical to the problem of matching document text that was scanned from poor quality images. Words that occur near one another in a text also "reinforce" each other for searching the way that city, state, and ZIP do.

- It easily overcomes problems with spelling and addressing variations and errors on mailpieces or in postal databases. This is analogous to the problem of matching document text that contains spelling variations. It also achieves an effect similar to stemming in that different words having the same root are considered to be similar.

- It can also handle abbreviations fairly easily. This is also similar to the problem of word stemming.

- Some postal address interpretation systems depend heavily on the system's ability to recognize certain key words, such as STREET or BOX. These words play a role similar to stopwords in that they carry little content, and mostly provide structural information. N-gram-based matching can handle errors even in these words.

Drawing on the experience of using N-grams in postal work, ERIM has been adapting these techniques for use in full-text retrieval and filtering. We currently have a working full-text retrieval system, called *zview*, which provides a simple query facility for accessing a wide variety of ASCII documents, including biographies, abstracts, and documentation. The *zview* system's N-gram-based matching provides a very simple, easy-to-use text retrieval system. It is tolerant of spelling variations and errors in both the text and the documents. It also provides a straightforward ranking of documents that enables the user to see which documents best match the query. This system has given us some confidence in the wider utility of N-gram-based matching for information retrieval. Although the TREC task would not seem to directly require inexact matching (the documents were high quality ASCII), our experience with *zview* indicates that this kind of tolerance for variations in user input is very useful for general retrieval.

Incidentally, one criticism that one can make of N-gram-based systems is that they require large amounts of storage. Although this is true of the naive implementation, there are a number of variations using various superimposed coding techniques that provide substantial space savings without impacting matching performance. See [2] for details.

2.0 System Description

The *zview* system mentioned above has two parts: an off-line index builder, and an on-line retrieval interface. In its current form, *zview* does not yet use our most compact N-gram representation techniques, and is thus not well-suited for very large (i.e., gigabyte size) databases. Also, it can handle only single query strings, not the multiple query strings necessary for TREC. Instead, for this task we opted to use a simple filter architecture, and concentrated on making it as fast as possible. This architecture has several advantages:

- It requires no disk space for an index. In fact, we were able to save even more disk space by leaving the original documents in their compressed form, and uncompressing them on the fly.

- It is conceptually simple, requiring only a very modest amount of code (987 lines of C source). The whole system took only a few days to design and build.

This architecture is similar in concept to that used by Mark Zimmerman's PARA system, which participated in last year's TREC. That system used a set of handcrafted Unix *awk* scripts to filter the documents. The PARA system, although it got good results, took 22 days on a more or less dedicated NeXT machine, even though its match criteria generally specified fewer match terms than our system did. Also Zimmerman's system used the notion of proximity to identify sets of matching lines that occurred near one another, whereas our system counts matches throughout the document without regard to the proximity of match terms. See [4] for details.

Obviously, a significant disadvantage of such a filter-type architecture is that the system has to read all of the data to simulate a retrieval. This is very computationally intensive, but we were able to mitigate this somewhat in our system by handling much of the actual N-gram-based matching with a highly parallel bit-vector approach, which we describe below. Also, we were able to split the processing load up among a suite of computers and run them all in a coarse-grained parallel fashion.

Figure 1 gives an overview of our system in dataflow fashion. We discuss each of the processes in this diagram in the following sections.

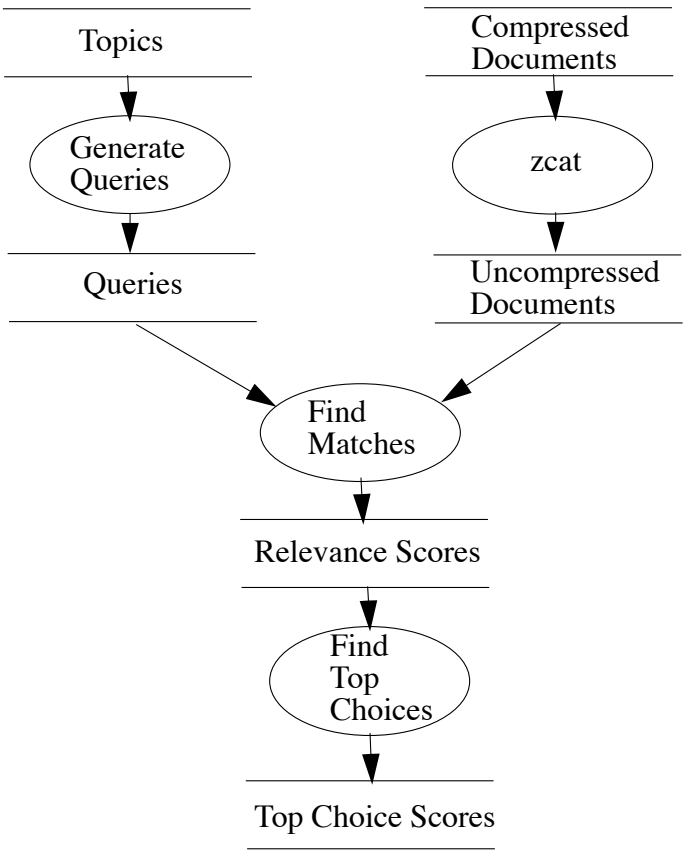

Figure 1: Dataflow Diagram for N-Gram-Based Text Retrieval

2.1 Generating Query Strings From TREC Topics

The process labelled "Generate Queries" represents a program *gen_query*, which takes the TREC topics file and generates a set of query strings for each topic. We considered several different schemes for extracting query strings from the topics, but finally settled on using just the phrases in the concept and nationality sections. The following is a typical topic from TREC-1:

<num> Number: 007
<title> Topic: U.S. Budget Deficit
<desc> Description: Document will mention a proposal to decrease the U.S. budget deficit.
...<con> Concept(s):
1. U.S. budget deficit, federal budget shortfall
2. foreign affairs budget, defense budget, entitlements
3. increased revenues, tax increase, tax reform, auction quota
4. reduction in expenditures, spending cuts, cutting domestic
programs, eliminating government subsidies
5. NOT financing the U.S. budget deficit
...
<nat> Nationality: U.S.

Given this topic, the *gen_query* program generates the following set of query strings:

007 000 U.S.
007 000 U.S. budget deficit
007 001 federal budget shortfall
007 002 foreign affairs budget
007 003 defense budget
007 004 entitlements
007 005 increased revenues
007 006 tax increase
007 007 tax reform
007 008 auction quota
007 009 reduction in expenditures
007 010 spending cuts
007 011 cutting domestic programs
007 012 eliminating government subsidies
007 013 NOT financing the U.S. budget deficit

Each line of the query output corresponds to one nationality or concept string from the topic. The nationality strings are listed first, followed by the concept strings in the order they appeared. There is also a rank number associated with each query string. Generally, as concept strings were used in the topics, strings that appeared earlier in the list were more important than strings that appeared later. The actual filtering program also uses these rank numbers to weight the query strings in importance. Each string receives a weight equal to 2k - r, where k is the number of query strings for the topic and r is the string's rank value. Any nationality or concept string can also have a NOT prefix, which triggers a slightly different kind of processing for these strings, described below.

2.2 The Problem of Finding Matches While Filtering Text

In Figure 1 above, the process labelled "zcat" represents a Unix utility program that takes a compressed file and returns its uncompressed form. The process labelled "Find Matches" represents a program *find_matches*, which takes a set of queries for all of the topics at once, and applies them to a single uncompressed text file. Given the filter architecture for this system, it was important for us to minimize the total amount of I/O. By letting *find_matches* process the queries for all topics simultaneously, the system has to pass the data only once.

To make this process as efficient as possible, *find_matches* does most of its matching using bit-vectors representing the presence or absence of particular N-grams. Suppose that we had a string that we wanted to search for in a document using N-grams. An obvious way to do it would be to perform the following steps:

- Break the query string up into N-grams.
- For every line in the document, count how many of those N-grams occurred in the line. This count would be the match score for that line.
- Keep a sorted list of the top-scoring lines, bumping the lowest scoring entry on this list when a new line has a better score.

When this algorithm finishes, we will have a list of the top scoring matches for our query string. We could make this process a little more sophisticated by putting a threshold of some sort on it. If a line's match score was below this threshold, say 80% of the number of N-grams in the query string, we could ignore this line as simply not containing enough of the query string to consider.

Unfortunately, this naive implementation of N-gram-based matching, which uses nested loops to compare every N-gram in the query string with every N-gram in a line, is computationally very costly.

2.3 Matching and Scoring Lines

We can do significantly better than the naive N-gram-based algorithm described above by using hashing. Figure 2 illustrates this process for bi-grams. The key idea here is to break the matching process into two parts:

- Initialize a hash-code-based look-up table for all of the N-grams in the query string. Each slot simply records whether *any* N-gram hashed to that position (1) or not (0). Thus the hash table is a simple bit vector.
- For each line in the document, make a single pass for all of the line's N-grams, checking each one to see if it is in the table. In other words, see if the slot (bit) corresponding to the N-gram contains a 1. If it does, add 1 to the line's score.

If we make the hash table big enough, we can make the probability of collisions arbitrarily small. For example, consider that there are only 52,022 possible bi-grams and tri-grams for the alphabet consisting of A-Z, 0-9, and space. A hash table size of, say, 20,000 slots (625 32-bit words or 2500 bytes) provides far more than ample space to avoid collisions. If there is a collision between the hash of an N-gram in the query string and that of some different N-gram in the line, its only effect is to cause that line's score to be one higher than it should be. It is even more unlikely that two or more of the N-grams would also collide. Thus by allowing collisions in the hash table, we get a simple, quick algorithm at the cost of a small probability of small distortions in the line scores. See [2] for a further discussion of how to estimate the probability of these kinds of collisions. Note that we include leading and trailing spaces for words in counting N-grams. Also recall that in our full implementation of this we process tri-grams the same way and at the same time. The match score between two strings includes the count of both matching bi-grams and matching tri-grams.

To get an additional gain in efficiency, we can carry this hash table scheme for counting N-grams another step by using parallelism to match a single line from a document against a number of query strings all at the same time. Figure 3 illustrates this scheme. Again, matching is a two-step process:

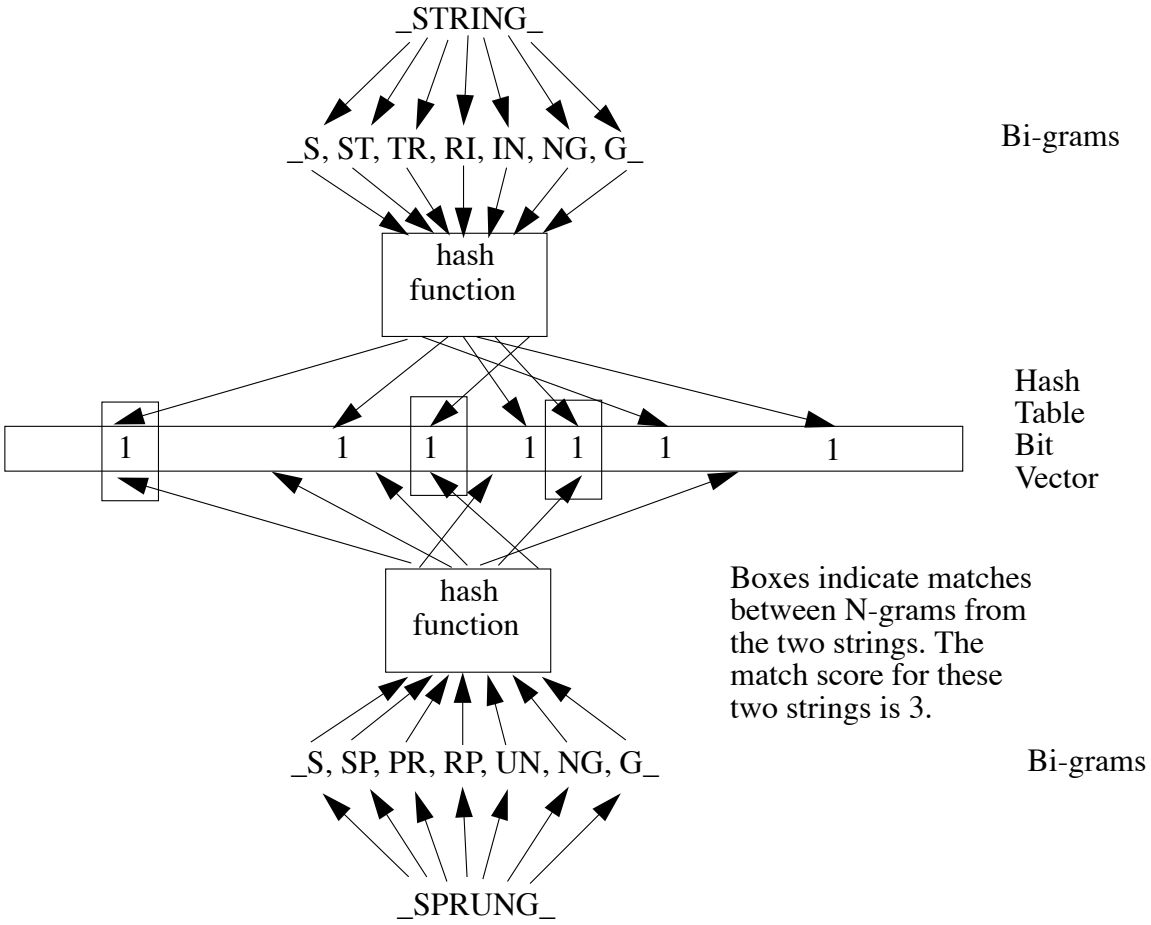

Figure 2: Using Bi-Grams to Compare "STRING" and "SPRUNG"

- Initialize a hash table for the entire set of query strings. Each slot in the hash table again represents one N-gram, but it is now a bit vector in which the kth bit indicates whether the N-gram occurs in the kth query string.

- For every line in the document, hash each N-gram to find the corresponding bit-vector. Then add that bit vector to the current score vector for the document. The kth element of this score vector keeps the current match score for the document against the kth query string.

One other significant aspect of this multi-query-string implementation is that we also represent the final score vector for all of the queries as a set of parallel bit-vectors. This effectively allows us to perform 32 additions in parallel by adding one 32-bit word of an N-gram bit-vector from the table to one 32-bit word representing the 1-bits of 32 query string scores, and then propagate the carry bits up to bit-vectors representing the higher order bits of the scores. Effectively, this lets us compute scores for 32 query strings in close to the same amount of time as it takes to compute the score for a single query string.

2.4 Scoring and Ranking Documents

After the system computes the N-gram match scores of a single line against all of the query strings, it applies a threshold criterion. If the score for a particular query string is below its threshold, the system treats it as a zero score. The threshold is defined as a percentage of the maximum possible N-gram score for the query string. For example, if a particular query string has a maximum possible score of 20 matching N-grams, and the threshold percentage (as determined by a command line argument to the program) is 70%, then the system will ignore any line whose score is less than 14. There is a similar command line argument which determines the threshold percentage for negation query strings (those tagged with a NOT prefix in the query string set). If a document has a line whose score for a negation query string exceeds its threshold, the system will discard the document. For most runs of the system, we got the best results with a match threshold at 70% and a negation threshold at 95%. In other words, we were willing to accept a line as a match for a given query string if it had 70% of the

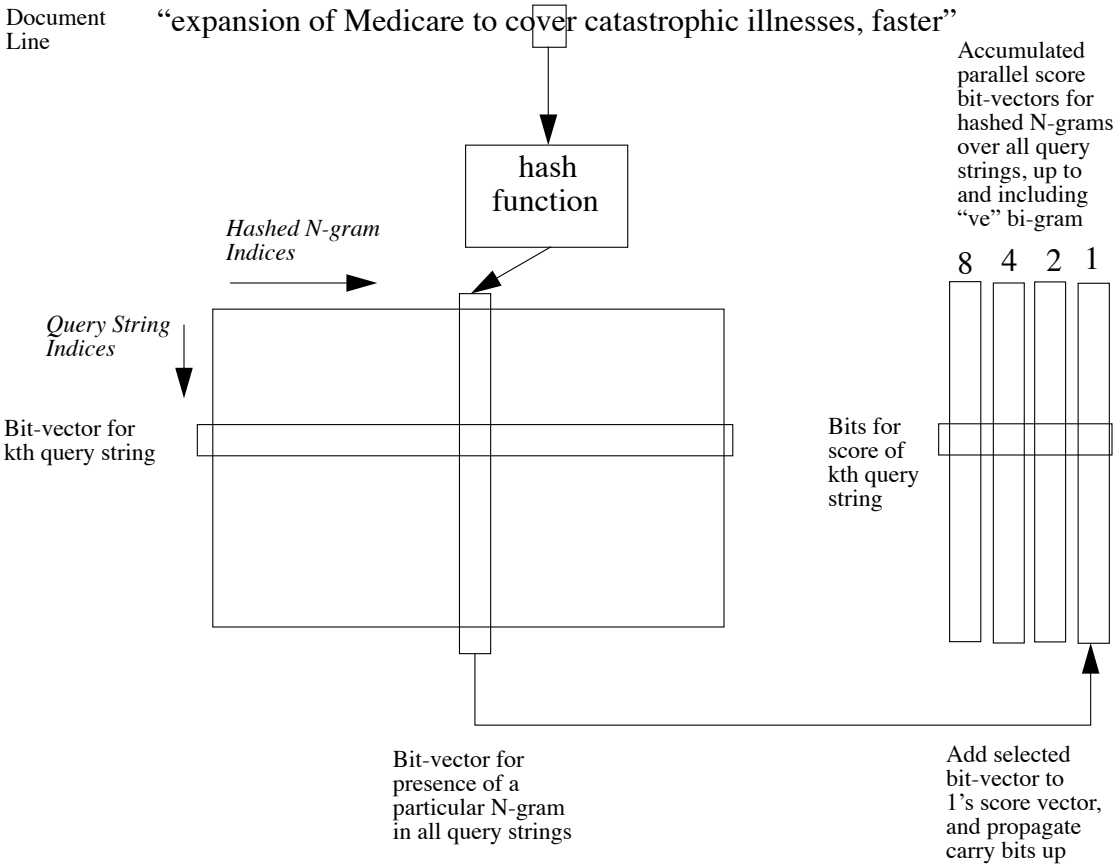

Figure 3: Parallel Implementation for Multiple Query Strings

requisite N-grams, but needed a nearly exact match in order to discard it for matching a negation query.

To compute the scores for a whole document, we accumulated the line scores for each query string, but applied a cap at twice the maximum possible N-gram score. In other words, if a document mentions a particular query string twice, that is twice as good as mentioning it once. However mentioning it three or more times is of no additional value. We put this cap into the system after observing that initial versions of the system would overrate some documents because they mentioned some low ranking query string for a given topic many times, but completely missed any of the higher ranking query strings for the topic.

To generate the final scores for a document against a topic, the system accumulates the scores for each query string in the topic, multiplying them by the weighting factor computed earlier. If this aggregate weighted score exceeds a hard-coded threshold of 40, the system then writes out this score as a relevance judgement for the document. Later a separate program reads the file containing the relevance scores, computes the top 1000 scores for each topic, and writes them out as the final system result.

3.0 Multi-Processor Execution

Even with all of this attention to efficient implementation, the system still required a large amount of computation. Fortunately, ERIM had available a network of Sun workstations over which we could distribute the processing load. This network consisted of two Sun SPARCstations and five or six Sun SPARCstation 2 machines. We used a very simple partitioning scheme, assigning each machine a fixed set of documents to evaluate against all of the topics. Given this arrangement, a full run of topic set 3 against Disks 1 and 2 took 12 to 13 hours if all of the machines were fully available. When there was significant contention on some of the machines, the run might take as much as 24 hours. An obvious drawback with using the fixed partitioning scheme was that often one or more of the machines would not finish until many hours after all of the others had. This was usually due to unexpected heavy contention on one or more machines. If we had used a more intelligent and dynamic document file assignment mechanism (implemented in, say, the Linda coordination language), we would have had better elapsed execution times.

TABLE 1. Selected TREC-2 Test Results

Test Number	Note	Test Type	Query Thresh.	Negation Thresh.	Average Precision	At 10 Docs	At 100 Docs	R-Precision
erima1	official	ad hoc	80	80	0.1589	0.3960	0.3016	0.2179
erima2	official	ad hoc	75	80	0.1885	0.4480	0.3426	0.2494
a30	official	ad hoc	70	95	0.1604	0.4040	0.3038	0.2198
a31	official w/ X queries	ad hoc	70	95	0.1153	0.4000	0.259	0.1904
a33	spanning lines	ad hoc	70	95	0.0734	0.15	0.1484	0.1333
a34	span lines; exp. decay weighting	ad hoc	90	95	0.1099	0.314	0.2276	0.1776
a35	span lines; exp. decay weighting	ad hoc	80	95	0.1336	0.3000	0.2442	0.2004
a35-AP	AP only	ad hoc	80	95	0.2717	0.4740	0.2838	0.3140
a36	span lines; exp. decay weighting	ad hoc	70	95	0.0792	0.1780	0.1678	0.1387
erimr1	official	routing	80	80	0.1219	0.3580	0.2304	0.1814
erimr2	official	routing	75	80	0.1415	0.4240	0.2524	0.2031
r5	official	routing	70	95	0.1225	0.3600	0.2310	0.1818
r6	span lines; exp. decay weighting	routing	80	95	0.1015	0.2880	0.1954	0.1604
r7	span lines; exp. decay weighting	routing	70	95	0.0602	0.2200	0.1388	0.1123

4.0 Results

To test our system, we ran it a number of times, varying different system parameters. After the conference, we were also able to make a few changes and run it again. Table 1 above summarizes some highlights of our results, which show a number of interesting points:

- The tests numbered erima1, erima2, erimr1 and erimr2 were the official results turned in on June 1. The first two were the ad hoc results, and the second two were the routing results.

- The tests numbered a30, a35, a36, r5, and r7 were all attempts to determine good settings for the query threshold and negation threshold parameters. Unfortunately, the results from these test runs simply do not provide anywhere near enough data to perform a complete sensitivity analysis for these parameters. Also, we noticed in some of our testing on the TREC-1 queries that there is considerable difference in the optimum values of these thresholds for different topic sets/data set combinations.

- One of the motivations for using N-gram-based matching was that it provides good matching performance in the face of textual errors. To test this idea, we ran test a31 using deliberately damaged query strings. In this test, we took each query string produced by *gen_query*, and replaced the third character with the letter "X". This works out to be an effective character recognition error rate of 4.5% over the whole body of query strings. Although the system took a considerable hit in performance, the interesting thing was that it still functioned at all. Many of the other systems in the TREC evaluation would most likely have completely failed in an analogous test, since they depend heavily on exact word matches.

- One serious drawback to the original system was that it did not span lines when matching. That is, if the text that

a particular query should match was split across two document lines, then the match would fail. After the conference, we found a relatively easy way to rectify this design deficiency and were able to run tests a33 though a36, and r6 and r7. Unfortunately, this change greatly affected the system's response to the query and negation thresholds, and we were not able to run enough tests to find the optimum values for these parameters. The end result is that the best results we have for this improved system are still not as good as the official results we reported.

- Another serious problem with the original system came to light during the official conference. As Donna Harman pointed out in her closing talk, no one has yet really explored the full ramifications of changes in term weighting strategies. Our original system used a very simple-minded linear-falloff weighting scheme. Our assumption was that concept strings appeared in reverse order of importance. However, we began to suspect, based on different concepts that were mentioned in a number of the conference presentations, that this was overly simplistic. We decided to implement a straightforward exponential-decay weighting scheme. In this approach, the first query string gets a weight of 100, the second gets a weight of, say, 90, the third a weight of 81, the fourth a weight of 72, and so on, with each succeeding weight taking 90% of its predecessor's value. Unfortunately, we did not have time to tune the system's response to this change either, and its results (a34, a35, a36, r6, and r7) are worse than the official results as well. However, it appears that there is plenty of room for experimentation with different term weighting schemes and we will continue working with them.

- In our early work with the system before sending in the official results, we did a fair amount of testing using just the Associated Press document set. We were perhaps misled by how well the system did on these documents, and missed some chances to improve the system earlier. The test labelled a35-AP shows what the results look like for a35 if we restrict the new system to just returning AP documents, and restrict the relevance judgements to just AP documents. Even with this imperfectly tuned new version, we see that the system is capable of significantly better performance. It is unclear why there should be such variation between the retrievability of the AP documents and the other document collections.

At its best, our system performed as well as most of the systems that participated in TREC-1. However, there is ample room for improvement, as we have noted above, especially in comparison to many of the systems that came back for TREC-2.

5.0 Further Research

The TREC-2 task is the first real application for our N-gram-based multiple-query system. As in any experiment of this nature, the results and problems suggest many more possible avenues of research. These ideas fall into two categories.

5.1 Analyzing the Current System's Performance

Further analysis of the existing system will allow us to better understand its behavior and limitations. Some ways to do that include:

- It is likely that generating query strings from the topic concept strings may have significantly limited performance. For example, Topic 74 about instances where the U.S. government propounds conflicting policies completely failed to mention terms such as *policy* or *regulation* in the concept list. Thus, our system had only a very small chance whatsoever of finding matching documents. Zimmerman's filtering system [4] did well with handcrafted queries, so we should also try manually generated queries.

- Currently the system has a hard-coded cutoff threshold of 40 for the weighted aggregate score. The purpose of the threshold was to prevent the system from returning results that were guaranteed to be noise because of their very low score. This value was set more or less arbitrarily, so we should experiment with changing this threshold to determine its true effect. In all likelihood, it could be a fair amount higher, preventing the system from generating other useless low-scoring results.

- Currently the system sets a cap of three times the maximum N-gram score for any query string score. Again, this value was determined only by a very rough empirical process, so we should experiment with changing this cap, to see how much impact it has.

5.2 Extending the System

We can also make some significant changes to the system to explore possibilities for other performance improvements.

- Currently the system treats upper and lower case alike for both documents and queries. Since acronyms and brand names have different meanings sometimes from uncapitalized words having the same letters, perhaps there is a way to take the case of letters into account when computing a match. That is, we could count a

match between two words with different capitalizations as a good match, but a match between two words with the same capitalizations as a better one.

- Our system is basically just a text filter. As such, it is already close to being useful for various routing tasks. However, to use N-gram-based matching for on-line retrieval, we will have to implement a true index-based system. Ideally, we would like to integrate the text retrieval capability of our *zview* system with the flexibility of the multi-query filtering system described here. Although we have an initial design sketched out, it will take an investment of further time and machine resources to implement this idea and test it. We will also be using some of our new compact N-gram representation techniques to reduce the large amount of index storage and computation required.

Acknowledgments

We would like to thank our sponsors at DARPA/SISTO for allowing us the opportunity to participate in TREC-2. We would also like to thank Donna Harman and her staff at NIST for all of their help. Finally, we are grateful for the United States Postal Service's sponsorship of the original N-gram-based matching technology that inspired our TREC research.

References

[1] W. B. Frakes. Stemming Algorithms. In William B. Frakes and Ricardo Baeza-Yates, Editors. *Information Retrieval: Data Structures & Algorithms,* pages 131-160. Prentice Hall, Inc. Englewood Cliffs, NJ, 1992.

[2] William B. Cavnar and Alan J.Vayda. Using superimposed coding of N-gram lists for efficient inexact matching. In *Proceedings of the Fifth USPS Advanced Technology Conference*, pages 253–267, Washington, DC, 1992.

[3] William B. Cavnar and Alan J.Vayda. N-gram-based matching for multi-field database access in postal applications. In *Proceedings of the 1993 Symposium On Document Analysis and Information Retrieval*, pages 287–297, University of Nevada, Las Vegas.

[4] Mark Zimmerman. Proximity-correlation for document ranking: The PARA Group's TREC Experiment. In *Proceedings of the First Text REtrieval Conference (TREC-1)*, NIST Special Publication 500-207, 1992.