Using Superimposed Coding Of N-Gram Lists For Efficient Inexact Matching

William B. Cavnar and Alan J. Vayda

Environmental Research Institute of Michigan
PO Box 134001
Ann Arbor, MI 48113-4001

Abstract

N-gram encoding for strings provides good inexact matching performance, but pays a large storage penalty. We describe a method for superimposed N-gram coding that provides more than a 40-60% reduction in the storage space required over the typical N-gram implementation while giving good speed and matching performance. Our system uses bit vectors to represent the occurrence of sets of N-grams in lexicon strings. The space savings comes from the notion of superimposing sets of N-grams onto single bit vectors. We also discuss an extension of the superimposed coding idea which encodes every N-gram with an ensemble of bit vectors in such a way as to yield even greater space savings.

1. Introduction

Retrieving street and city name information from the USPS database and business name information from a business name database for an arbitrary mail piece is a difficult task. This is largely because of inconsistencies between data values as they appear on a mailpiece and the actual stored values in the database. These inconsistencies are the result of numerous interfering factors, including:

- patron addressing errors
- patron abbreviations
- image quality problems
- character recognition problems
- incorrect or incomplete database records

All of these inconsistencies push us in the direction of finding "discrepancy-tolerant" or "inexact" matching techniques that are also reasonably efficient for our algorithm development purposes. We are currently using an approach to inexact matching that satisfies both the tolerance and efficiency needs. This approach is based on the use of N-grams to break both database records and query strings into matchable pieces. In this paper, we use the term *N-gram* to refer to a slice of N contiguous characters from a longer string. Our method uses bi-grams (N = 2) and tri-grams (N = 3) together. Also, we use only nonpositional N-grams. That is, the only fact we keep track of is whether a given string contains an N-gram, not where it falls in the string.

Typical implementations of N-gram-based matching require large amounts of storage. The approach we describe here requires significantly less storage, and also allows the database designer to tune the system by further trading off matching performance and storage.

In this paper we will cover the following topics:

- Section 2: Storage analysis for various implementations of N-gram-based matching
- Section 3: Implementation of N-gram-based matching with superimposed bit vectors
- Section 4: Performance of N-gram-based matching with superimposed bit vectors
- Section 5: Implementation of N-gram-based matching with Bloom-style encoded bit vectors

- Section 6: Performance of N-gram-based matching with Bloom-style encoded bit vectors
- Section 7: Applications of N-gram-based matching in processing ASCII addresses

2. Analysis Of Storage Requirements

In this paper we will use as our principal example the USPS City-State-ZIP database. The version of the database that we used contained 79674 unique City-State-ZIP records. To illustrate the storage requirements for different implementations of N-gram-based matching, let us look at the storage requirements for just the city name field alone using different approaches which we will discuss:

Approach	Storage Computed As	Total Bytes
Raw City Name Data	79674 recs * 28 chars	2.2M
Naive Bit Vector Implementation	79674 recs * 52022 N-gram bits / 8 bits per byte	518.1M
Better Bit Vector Implementation	79674 recs * 9406 N-gram bits / 8 bits per byte	93.7M
Record Pointer Lists	79674 recs * 20.77 N-gram ptrs per rec * 4 bytes per ptr	6.6M
Superimposed Bit Vectors	79674 recs * 510 N-gram buckets / 8 bits per byte	5.1M
Bloom-Type Encoding	79674 recs * 256 bits / 8 bits per byte	2.5M

Table 1: Summary of Storage Requirements For N-gram-based Methods

Let us look at each of these in slightly more detail:

- The naive bit vector implementation has a bit for each of the 52022 possible bi-grams and tri-grams in each of the 79674 records in the city name data.
- The better bit vector implementation has a bit only for those N-grams that actually occur.
- The record pointer list approach stores a record pointer for each N-gram that occurs in a record. On average, a city name contains 20.77 different N-grams (counting both bi-grams and tri-grams). One could represent the actual record pointers more compactly than with a 4-byte pointer, but there will be a processing penalty.
- The superimposed bit vector scheme is like the better bit vector implementation, except that it superimposes bit vectors for unrelated N-grams into composite vectors called *buckets*. We chose the number of N-gram buckets, 510, empirically. If we had fewer buckets, we began to have problems with many false positives coming up in the best-20 list of matches. On the other hand, using more than 510 took more space, but did not appreciably improve matching results.
- Bloom-type encoding allows further superimposition of bit vectors by representing each N-gram as an ensemble of bits occurring in different vectors.

We will discuss each of the various bit-vector implementations mentioned in Table 1 in the sections to follow. We will not discuss the record pointer list implementations other than to point out that it is tricky to make such schemes work efficiently.

3. Implementation Of N-Gram-Based Matching With Superimposed Bit Vectors

3.1. The Naive Bit Vector Implementation

One way to implement a database retrieval scheme based on N-gram matching is with bit vectors. The key idea is to represent each N-gram with a bit vector. Consider the diagram in Figure 1 (tri-grams have been omitted for clarity). It depicts matching the misspelled city name "GLENNBROOK" against a list of real city names. Every possible N-gram has its own bit vector. If there are R records in the database,

each vector will be R bits long. Each bit in an N-gram's bit vector is on (1) if the corresponding data record contains the N-gram, and off (0) if it does not.

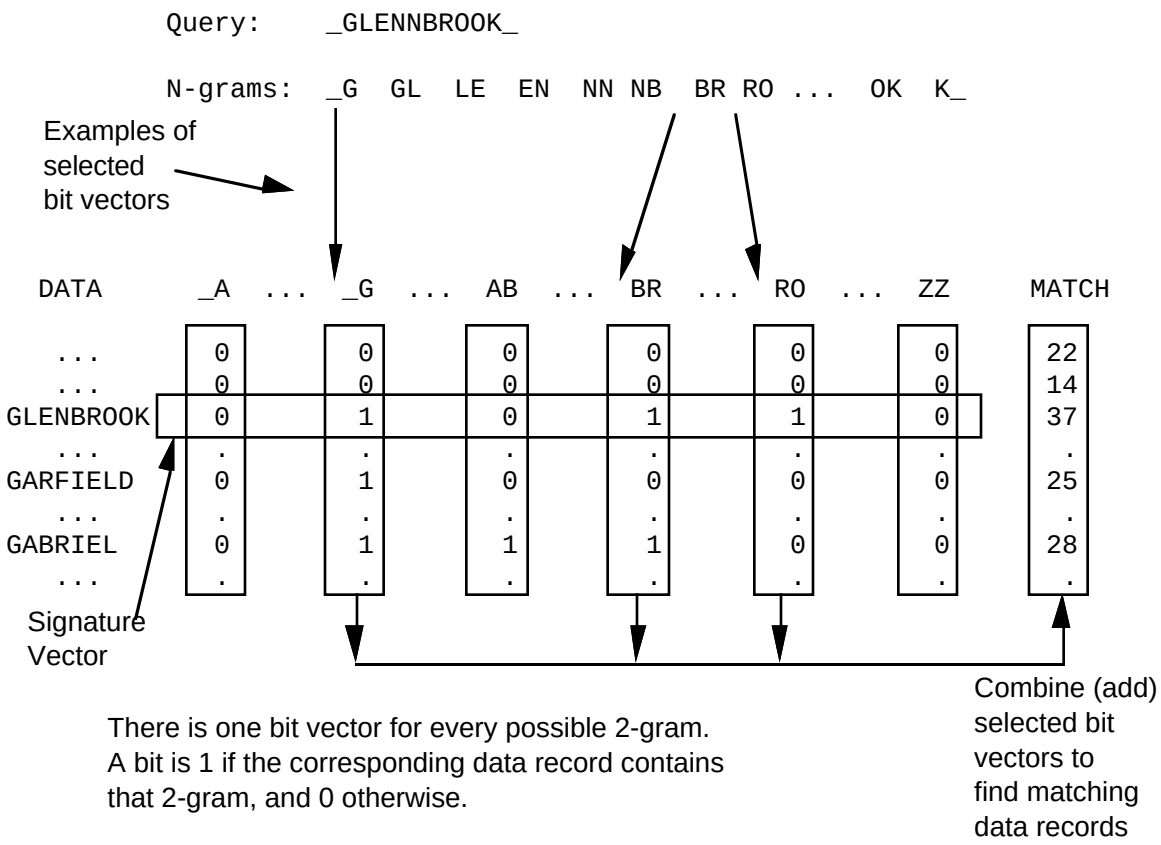

Figure 1: Retrieving The Best Matches to a Query String from an Exhaustive N-gram Representation

Normally we view the bit-table (the assembly of bit vectors taken together) along the vertical axis, that is, by focusing on the bit vectors themselves (shown as columns in the figure). Alternatively, we can view the bit-table along the horizontal axis. If we take the *k*th bits of all of the bit vectors in order from "_A" to "ZZ", they constitute a *signature vector* (shown as a row in the figure) for the *k*th data record. This signature provides a useful, fixed-length representation of any string.

A retrieval using N-gram-based matching proceeds as follows:

- break the query string into its constituent N-grams (e.g., _G, GL, LE, etc.)
- select the corresponding N-gram bit vectors (the columns in the figure above)
- combine those selected N-gram bit vectors to construct a vector representing the matching records.
- retrieve the matching records from the database

The third step can take several forms. If we want to do a simple exact match, we can intersect (logical bit-wise AND) the bit vectors together yielding a single bit vector. This resulting bit vector will have a one (1) in the *k*th position if and only if all of the *k*th bits in the selected bit vectors was also one (1). The one bits in the result vector correspond to database records which are likely matches. (Recall that a string matching on all of the N-grams in the query string is only a *likely* match, not a guaranteed match.)

Alternatively, we can compute a match score by "adding" together the selected N-gram bit vectors. This is logically equivalent to counting the relevant bits in the signature vectors of all of the database records. If we do this for all of the signature vectors (rows) in the bit-table, the resulting vector of integer values are the match scores for all of the database records as shown in Figure 1 above.

3.2. A Space-Efficient Representation of an N-gram Database

Of course, the major drawback to an exhaustive N-gram database is the enormous amount of space that it takes stored as bit vectors. Using an alphabet of {A-Z, 0-9, blank}, there are $37^2 = 1369$ possible bi-grams, and $37^3 = 50653$ possible tri-grams. The problem is that many or even most of these, e.g., "QX" or "VTC", simply never occur in normal English text or names. Their corresponding bit vectors would be completely empty.

One answer would be to keep bit vectors only for the N-grams that actually occur. For example, in the city name field in the City-State-ZIP database there are 9406 bi-grams and tri-grams that actually occur. Only these would need bit vectors. [HULL82] describes a related idea (linear marginal indexing) involving traversing a tree structure to find the appropriate bit vector for a given N-gram; only those N-grams that actually occur have bit vectors in this tree.

Another answer would be to represent the database not as bit vectors, but as lists of record pointers. The problematic rare N-grams, such as "QX", would thus have zero-length lists. While this is certainly a more efficient use of space, it also requires somewhat more complicated processing for efficiently performing various list operations.

[OWOLABI88] suggests yet another way to save space by grouping database records into sets of 16 records and then recording a bit to represent the presence or absence of an N-gram in the whole set. One has to then examine the entire set if it is selected by intersecting the N-gram bit vectors.

A different approach to representation is to use a superimposed coding scheme that effectively "folds" a set of many different N-gram bit vectors onto a single bit vector. While this superimposition of bit vectors does imply a loss of information, there are two mitigating factors:

- If we can correctly distribute the superimposition assignments, every resulting bit vector will represent an even mix of a few common N-grams and a lot of rare or unused ones. Consequently there are fewer real collisions between N-grams than one might expect.
- Every character in a data record participates in several N-grams. In our scheme, we use both bi-grams and tri-grams. Thus every character is part of two bi-grams and three tri-grams. We can take the joint occurrence of on (1) bits in the bit vectors for those N-grams as strong evidence that they were all in fact present in the data record. We can think of this as a kind of distributed representation for each character in its own context.

We have implemented a superimposition scheme that successfully incorporates both of these factors to provide good match performance with a reasonable query time using a reasonable amount of disk space. The key idea is that we use a hashing function to map every N-gram to a bit vector (See Figure 2 where tri-grams have been omitted for clarity).

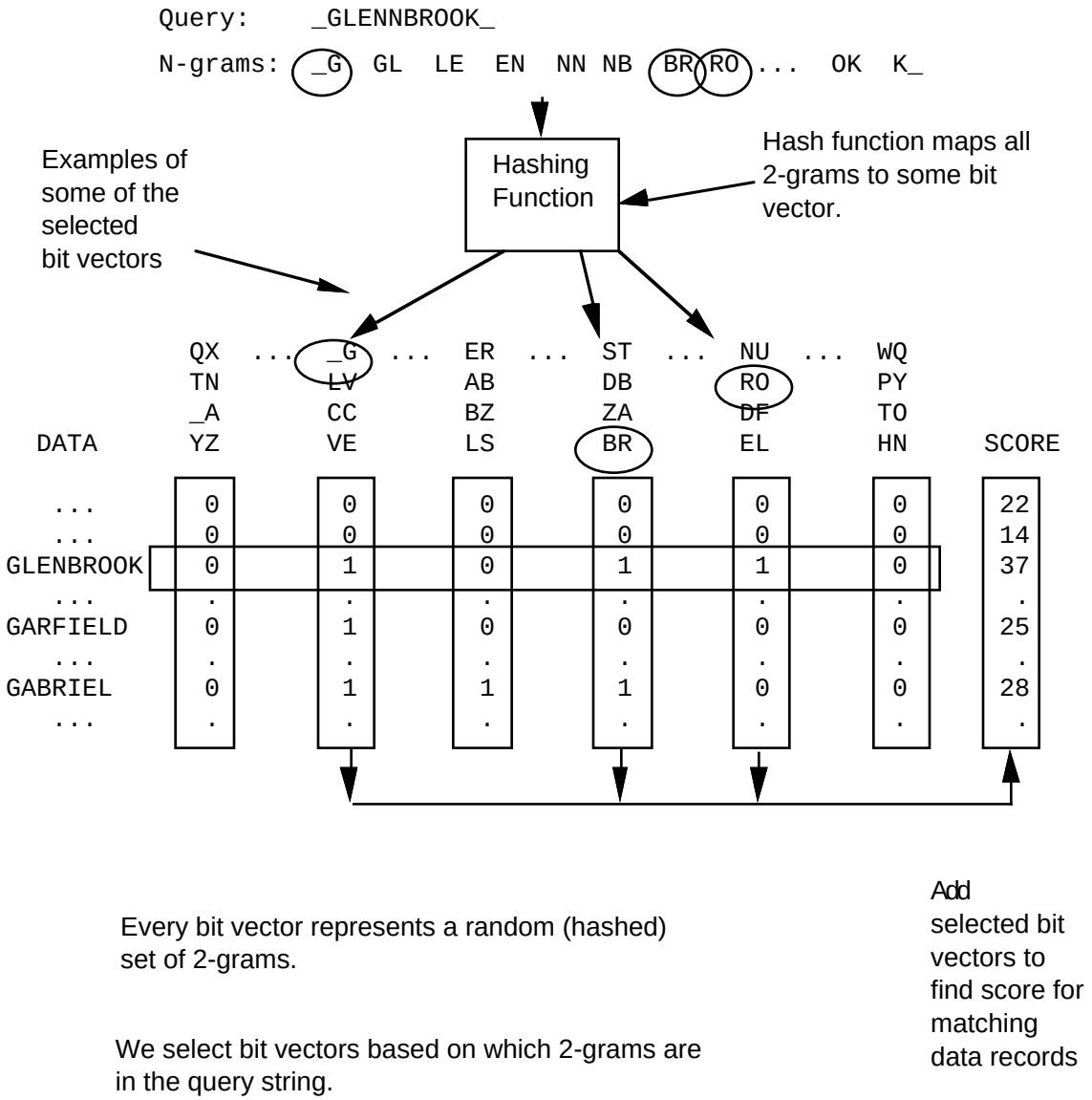

Figure 2: Retrieving the Best Matches to a Query String from a Superimposed N-gram Database

Notice how the hashing function maps the circled N-grams in the query string to the appropriate bit vectors. The hash function may map a number of different N-grams to the same bit vector, which we term a *bucket*. There is a tradeoff between the width of the bit-table (the number of buckets which the hashing function maps onto) and retrieval performance. If we make the bit-table narrower, we raise the average bit-density (the percentage of on-bits in the table) of each signature vector. This reduces the overall table size, but at the cost of reducing its ability to differentiate dissimilar records. This is because each bucket in the narrower table represents a larger number of N-grams, and thus provides greater chance for confusion. For the purposes of the USPS work we have done so far, we have used bit-table widths ranging from 37 bits (for a state code) to 510 bits (for a city name). Contrast these widths to the 52022 bits needed for the exhaustive 2 and tri-gram representation. Our bit densities range from 2.89% (for the ZIP field in a ZIP-street database) to 15.2% (for the state code field in the city-state-ZIP database).

3.3. A Field-Oriented N-gram Database

In many inexact matching applications, one would like to match not just strings, but data records having multiple fields. For example, in matching the city-state-ZIP line of a mailpiece, there are three distinct fields which must match conjointly. In our system, we build separate but parallel bit-tables for each field, and then match on all of them simultaneously. The separate field tables prevent confusion between the hashed N-grams in one field, say the city name, which might collide with those of another field, say the ZIP Code. Having separate fields also lets us individually tune the bit density of each field's bit-table, thus controlling its retrieval performance.

In all of the USPS applications of inexact matching, we use the ZIP Code as one of the matchable fields. Although the ZIP Code may not seem a good candidate for inexact matching, doing so provides us with several benefits. First, many times the ZIP Code on the mailpiece is actually wrong, but it is in the correct 3-digit ZIP area. The inexact matching helps localize the match by strengthening the scores of other database records in the same 3-digit ZIP. Another difficulty is that character recognition difficulties may have damaged one or more of the digits in the ZIP. Nevertheless, the inexact matching helps narrow the search by improving the scores for the records that do share the correct ZIP digits in those digit positions not in error.

Consider the example of a multi-field query against the city-state-ZIP database in Figure 3. In this example we are simulating the effects of matching a mailpiece in which the city name is misspelled (it is actually GLENBROOK with only one N) and character recognition incorrectly identifies one of the digits in the ZIP Code.

```
looking for 'GLENNBROOK/IN/46815'
with maximum possible score = 40
score cityname                    state ZIP
 35   GLENBROOK                     IN  46805
 29   GLENWOOD PARK                 IN  46815
 26   SUNNYBROOK ACRES              IN  46835
 24   BROOKSIDE ESTATES             IN  46835
 24   GREYBROOK LAKE                IN  47868
 23   MEADOWBROOK                   IN  46774
 23   GREENDALE                     IN  46815
 23   MAPLEWOOD PARK                IN  46815
 22   COUNTRYSIDE ESTATES           IN  46815
 22   GOLDEN ACRES                  IN  46815
 22   NORTHWOOD                     IN  46815
 22   NORTHWOOD PLZ                 IN  46815
 22   GLENBROOK                     OR  97456
 21   GLENBROOK                     CT  06906
 21   GWYNNBROOK                    MD  21117
 21   STEUBENVILLE                  IN  46705
 21   GREENVIEW                     IN  46815
 21   MAPLEWOOD PLZ                 IN  46815
 21   WEST BROOK ACRES              IN  47006
 21   WEST BROOK DOWNS              IN  47401
```

Figure 3: Example of Multi-Field Inexact Matching

The maximum possible score is 40, which is the total number of possible bi-grams and tri-grams in the three fields together. Notice that the best match has a strong match in all three fields. Lesser matches differ significantly in one or more of the fields. For our application, we currently generate the 20 best matches to an inexact query.

3.4. N-gram Statistics

When we built the index for the 79674 records in the City-State-Zip database described in the previous section, we computed the following statistics for the three fields:

Field Name	Number of Occurrences	Number of Unique N-grams	Number of Buckets	Occurrences per N-gram	N-grams per Bucket
City Name	1654617	9406	510	175.91	18.44
State Code	477866	245	37	1950.47	6.62
ZIP Code	950476	1317	100	721.70	13.17

Table 2: N-gram Statistics For City-State-ZIP database

where

Number of Occurrences	= how many N-grams appeared in that field altogether
Number of Unique N-grams	= how many unique N-grams appeared in the field
Number of Buckets	= how many superimposed bit vectors we used to index the field
Occurrences per N-gram	= the how many times the average N-gram appeared in that field
N-grams per Bucket	= the average number of N-grams that map to the same bucket and thus could possible collide.

For each, field we also show here the top 20 most frequent N-grams. The tilde character represents a blank in an N-gram. Recall that since we pad both ends of a string with blanks, an N-gram from the first part of a string will always start with a blank, and similarly, an N-gram from the end will always end with a blank. In addition, if a string contains multiple words, the first and last N-grams of each word will also contain blanks. These lists show both bi-grams and tri-grams.

City N-gram	Num of Occurrences
E~	18932
N~	16259
E~~	14636
LE	12900
ON	12864
N~~	11937
~C	11812
LL	11706
AN	11630
ER	11453
~S	11265
IN	10395
S~	10325
OR	9732
IL	9191
~B	8674
T~	8568
AR	8096
~M	8043
ON	7988

State N-gram	Num of Occurrences
A~	18807
A~~	18807
~N	12959
~M	11153
Y~~	9104
Y~	9104
~I	7834
L~~	6702
L~	6702
~C	6664
~NY	6141
NY~	6141
NY	6141
N~~	5781
N~	5781
~O	5589
~T	5169
~W	5121
~P	4902
PA	4706

ZIP N-gram	Num of Occurrences
1~	10206
1~~	10206
~4	10167
~1	9524
0~	8941
0~~	8941
~5	8544
~7	8445
2~~	8429
2~	8429
~2	8356
3~~	8216
3~	8216
4~~	7956
4~	7956
~9	7877
5~~	7875
5~	7875
~0	7726
6~~	7602

Table 3: N-gram Counts From City-State-ZIP Database

Using these counts, we can make some interesting observations, such as:

- there are many city names words that start with 'C', 'S', 'B' or 'M' or that end in 'E', 'N' or 'T'
- state codes starting with 'N', 'M' and 'I' dominate.
- 'NY' dominates the City-State-Zip database.
- ZIP N-grams are more evenly distributed than city name or state code N-grams, since the curve for number of occurrences is much flatter
- trailing digits in the ZIP field had a very definite, near-perfectly-ascending order in frequency, marred only by the fact that the USPS tends to make ZIPs end with '1' more often than they do with '0'

4. Performance Of N-Gram-Based Matching With Superimposed Bit Vectors

In order to get some measure of the actual performance of the superimposed bit vector implementation, we performed an experiment with simulated spelling errors. For this, we wrote a program which performed matches on strings containing simulated spelling errors using this implementation. In this experiment we:

- selected 5000 records randomly from the city-state-ZIP database
- produced for each record a number of different simulated erroneous city name strings to use as queries to the inexact match (See next paragraph for details.)
- performed an inexact match query (using our superimposed coding N-gram scheme) on those strings, returning the top 20 matches

- computed statistics on the resulting matches by finding the rank of the original "correct" city name record among the top 20 matches returned. (This is known as measuring *truth inclusion.*)

We used the four classic types of spelling errors, singly and doubly, at randomly selected positions in the strings in generating the erroneous city names:

- insertion (e.g., DALLAS becomes DALLIAS)
- deletion (e.g., DALLAS becomes DALAS)
- substitution (e.g., DALLAS becomes DALLOS)
- transposition (e.g., DALLAS becomes DALALS)

The point of using these simulated errors is to model some of the effects of different error sources. The deletion and substitution errors together simulate some kinds of character recognition errors. The four types of errors taken together cover the most frequently occurring kinds of problems that can occur during keyboard data entry. However, there are other kinds of patron errors, such as unusual abbreviations or phonetic misspellings, that these error types can only loosely approximate.

As mentioned in a previous section, our N-gram-based matcher is field-oriented, and has separate fields for city name, state code and ZIP code. We used only the generated city name and the original state code for the match, but supplied no value for the ZIP code field. We did this so that our match results would not be unduly biased by having strong matches based on state and ZIP codes alone. Also, we considered one of the returned records to be a match if it had the same city name and state code as the original undistorted record. We did not consider ZIP code to determine if a retrieved record was a match.

In Figure 4, we show the resulting truth inclusion curves for zero, one and two errors. Each curve shows the percent of truth inclusion as a function of the rank of the match. For example, for single errors the correct record was in the top four choices over 96% of the time. These curves give us a feel for how many entries we should keep in the top-ranked list to meet a particular inclusion performance level.

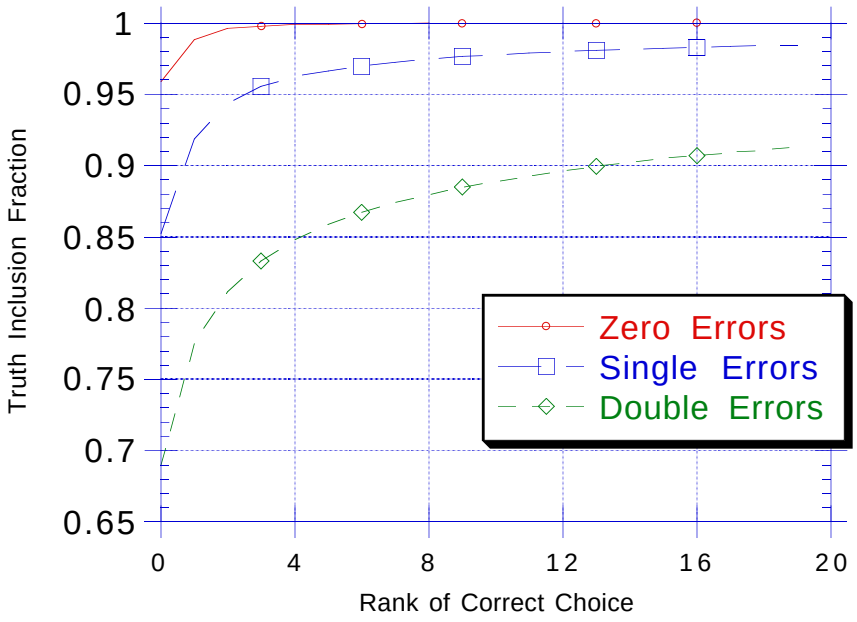

Figure 4: Truth Inclusion Fraction For Zero, One and Two Errors

One apparent anomaly worth noting is that even for zero errors, the correct record was not always ranked at the top. This is due, in part, to using unnormalized scores for this test. For example, if we are looking for "HUNTINGTON NY," the following 3 matches all have the same score, but only the third one is correct:

- EAST HUNTINGTON NY
- WEST HUNTINGTON NY
- HUNTINGTON NY

We could have normalized the match score by dividing the N-gram count by the length of the database string. This would have given the first two records a lower score and the third record would have been correctly ranked first.

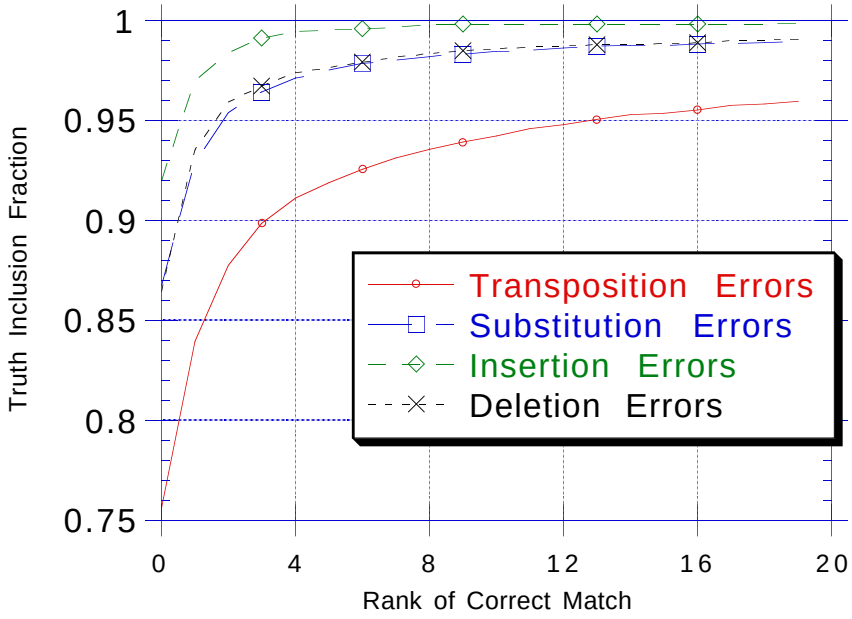

Figure 5: Truth Inclusion Curves For Single Errors

It is instructive to examine more closely the truth inclusion curves for single errors, as shown in Figure 5. Here we have split out the results for each of the four types of errors. As expected, transposition errors produce poorer truth inclusion because they are essentially double character errors, and thus cause more damage in the string. The differences between the other curves require some explanation. Consider the city name DALLAS, which consists of the following:

- bi-grams: _D, DA, AL, LL, LA, AS, S_
- tri-grams: _DA, DAL, ALL, LLA, LAS, AS_

Now consider how the following errors cause changes in the N-gram match score for this name by impacting different sets of N-grams:

Error	Resulting String	Impacted N-grams	Score Change
Insertion	DALLIAS	LA, LLA, LAS	3
Deletion	DALLS	LA, AS, LLA, LAS	4
Substitution	DALLOS	LA, AS, LLA, LAS	4
Transposition	DALALS	LL, LA, AS, ALL, LLA,LAS	6

Table 4: Impact of Errors on N-gram Scores

Thus we see that deletions and substitutions have basically the same impact on this string (score change = 4), while insertions have less (score change = 3) and transpositions have more (score change = 6). The curves in Figure 5 reflect the results of these score changes on matches. Of course one would get different results if the error impacted N-grams which were duplicated in the string. However, this is fairly rare.

5. Implementation Of N-Gram-Based Matching With Bloom-Style Encoding

5.1. The Bloom Filter For Dictionary Lookup

To achieve higher compression for the indices for inexact matching, we have recently been experimenting with the so-called Bloom filter, which was first introduced by Burton H. Bloom [BLOOM70] as a way to compactly represent hash coded information by allowing a small but tolerable error rate. To describe the key ideas, we will first describe a typical application of a Bloom filter, such as in an online spelling checker in a word processing package. In this application, when a user finishes a word (by typing, say, a space) the word processor would check the word's spelling and notify the user if there was a problem. Clearly, for this to be practical, the dictionary must be fast, which usually means that it must be in memory, and that implies in turn that it must be small. We can explain Bloom's system with the simplified diagram in Figure 6:

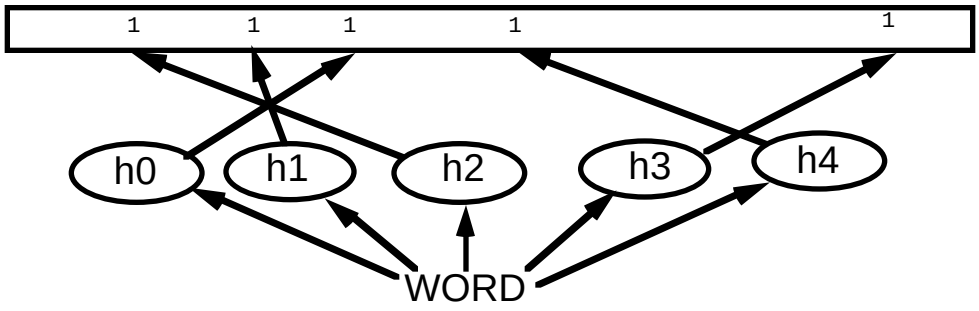

Figure 6: Bloom-type Representation Of A Word In A Compact Dictionary

The bar at the top of this diagram represents a long bit vector of, say, m bits. The bubbles represent a set of hash functions h_1 through h_k each of which maps a word to a number between 0 and m-1. To record a word in the dictionary, we perform the following steps:

- compute the values of h_1 through h_k for the word
- turn on the corresponding k bits in the bit vector. If a bit is already on, leave it on.

We perform these steps on every word we wish to store. In effect, the Bloom filter represents every word we store in the dictionary as a (hopefully unique) ensemble of bits. When we are done, we can use the

resulting bit vector to check if a particular word is supposed to be in the dictionary. To check for dictionary membership, we perform the following steps:

- compute the values of h_1 through h_k for the query word
- check the corresponding k bits in the bit vector. If all of the bits are on, the word probably is in the dictionary. *If any one bit is off, we are certain that the word is not in the dictionary.* These k bits constitute a test set for the word.

We can depict a failing membership test with the diagram in Figure 7:

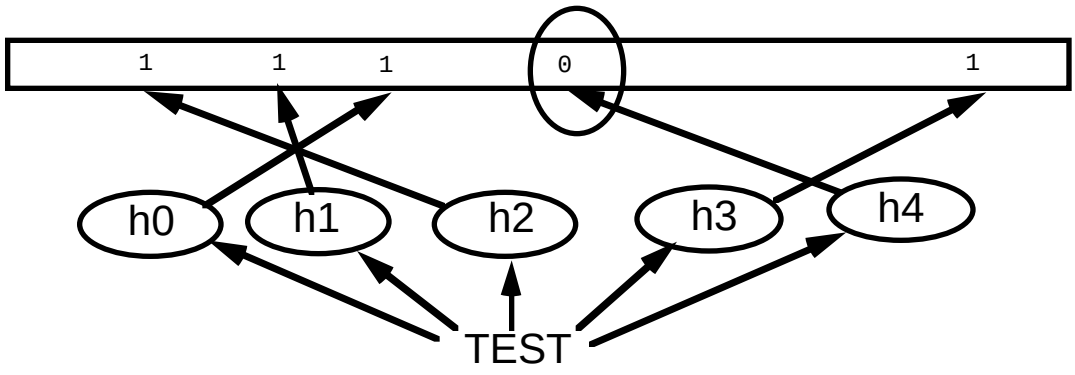

Figure 7: A Bloom-type Membership Test That Fails Because Of A Missing Bit

Thus we see that the Bloom-type encoding provides a probabilistic test for whether a given query word is in the dictionary. It is always right when it says NO. However, since it allows overwriting bits, there is a slight chance that the method will incorrectly says YES for a given word. This is because it could happen to hash the query word to a set of test bits that were all turned on by hashing and storing the bits from other words.

Before we go on, let us translate this scheme back to our inexact matching task. The N-gram set for the string will play the role of the dictionary, and each N-gram plays the role of a word in that dictionary. We will then represent the string with an m-bit vector, and we will hash each N-gram k different ways and turn on the corresponding bits. Thus, we can represent each N-gram as a ensemble of k bits in the bit vector. Likewise, if we want to test for the presence of a particular N-gram, we can examine its set of test bits and see if they are all on. If so, the N-gram is probably present in the string. If any one bit is off, we know that the N-gram is not present in the string. In this scheme, every string in our database would have its own bit vector.

5.2. Estimating Probability Of False Positives

The interesting question for such a probabilistic scheme, then, is this: To store n items (words, N-grams, etc.) reliably, how big should m (the number of bits in the bit vector) and k (the number of hash functions) be? Donald Knuth examined this question some in [KNUTH73]. In this discussion, he gives the following formula for estimating the probability of a false positive for an entry in a Bloom-type encoded dictionary:

$$P_{fp} = (1 - e^{-k*n/m})^k$$

where

k = number of hash functions (typically, $1 \le k \le 10$)

n = number of items to store in each bit vector (typically, $n \approx 20$)

m = length of bit vector in bits (typically, 100 ≤ m ≤ 400)

Using this approximation, we can derive the following curves in Figure 8 for n = 20:

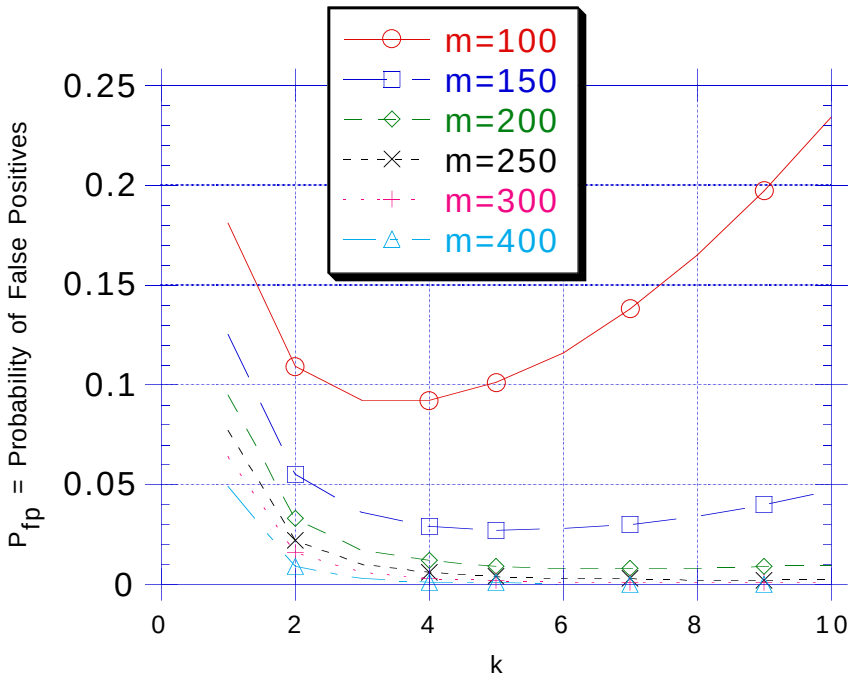

Figure 8: Estimates For The Probability Of A False Positive On A Single Item For n = 20

These curves, especially the ones for m=100 and m=150, illustrate a general property of Bloom encoding, namely, that for a given value of n and m, there is an optimal value for k. If k is lower than this value, there is too high a chance of duplicate test bit sets for different items, yielding more false positives. If k is higher than the optimal value, the loading factor (the percentage of 1 bits in the vector) becomes too high, and there are significant numbers of collisions between overlapping test sets, also yielding more false positives. The curves for m=200, 250, 300 and 400 don't show the full range for this behavior since we did not take k high enough to see P_{fp} start to climb again. By the way, the average number of N-grams in a city name string is 20.77, so the actual performance for the corresponding curves is slightly less.

Although we have not explored all of the tradeoffs implicit in Knuth's formula, we have adopted for now the heuristic of simply setting m equal to the number of bits in the original string. Thus we want our new Bloom-style index to take no more space than the original data. For the city name data, this works out to be 232 bits. Since we end up rounding the length of the bit vectors to fit in arrays of unsigned long (32-bit) words, this in turns works out to 256 bits. The curve for m=256 becomes essentially flat at k=5, so that is the value we used. By way of comparison, we can examine our previous superimposed coding approach in the light of these estimates. For the city name index, we were using m=510 and k=1. Using Knuth's formula, these values yield a P_{fp} of 0.038. Even with errors that high, we got good results. Thus we should expect to do at least as well with the new Bloom-type scheme.

6. Performance Of N-Gram-Based Matching With Bloom-Style Encoding

To test the performance of this Bloom-style encoding, we ran the same kind of test that we did for our previous inexact matching scheme. To find the best matches using this method, one must perform the following steps:

- build an index by constructing a Bloom-style bit vector D_i for each string in the database
- construct a Bloom-style bit vector Q for the query string
- perform a logical AND operation between Q and each of the D_i. This yields a bit vector R with a 1 in every place that has a 1 in both Q and D_i.
- count the number of bits in R. This becomes the estimated score for D_i.
- keep the top 30 records according to the estimated score. (We had to keep the top 30 rather than 20 in order to ensure good truth inclusion in the top 20. This is because we used the Bloom-encoding bit count to set the threshold for the top entry list, but then used the actual shared N-gram count to order that list. Another way to say this is that the Bloom-encoding count was fast to compute, but tended to call something a match when it was not. See the next paragraph for further discussion.)
- order the top 30 list by the actual shared N-gram count between Q and the selected D_i

Since we are simply counting the number of bits in R to get the estimated score rather than specifically checking for each of Q's N-gram test bit sets, we are taking a hit in accuracy. This is because, using the multiple-bit representation for an N-gram, there are k bits which must be present to be relatively certain that the N-gram is present. If only, say, 80% of those bits are on, it is *not* the case that the N-gram is 80% present in the string. Nonetheless, we can use this bit count as a useful thresholding mechanism for our top 30 list. Once we have the top list, we can compute a much better order for those records by using the real shared N-gram count. In fact, in some ways this produces superior results to our previous method, which does not include a post-processing stage to compute the true shared N-gram count for the best records. ([OWOLABI88] also advocates using a fast, but possibly coarse, primary method to prune the lexicon, and then a slower but finer method to sort through the remaining best choices.) Using this new composite method, the resulting truth inclusion curves for zero, one and two errors and for the separate single errors look very much the same as the curves for the previous method shown in Figures 4 and 5, and for the same reasons.

To summarize the discussion about Bloom-type superimposed coding, we see that it produces similar results while taking significantly less space. Furthermore, using Knuth's formula, we now have a way to engineer the design of the encoding by trading off the storage (length of the bit vector) against the false positive probabilities. In general, this type of encoding falls into the larger arena of distributed representations such as one finds in the neural network literature, and in fact shares some of their properties, such as the ability to generalize (find similarity) and to withstand some local damage. These properties suggest many more directions in which to take this line of research.

7. Applications of Inexact Matching in Processing ASCII Addresses

One of our USPS application systems uses ASCII address information (such as that taken from a raw mailing list) and tries to generate the correct 9-digit ZIP code [VAYDA92]. The goal of this system is to test ideas about database organization and address matching logic without having to deal with character recognition issues. This test system has a database covering 25 SCFs (i.e., 3-digit ZIP areas) which together constitute about 10% of the national database.

Currently this system uses the N-gram-based inexact matching approach in three places. The first is in city-state-ZIP matching. Since we have a national level database for this matching, we can usually generate reasonable matches even if there is a significant problem with either the city name, state code or the ZIP Code.

The second application is in ZIP-street lookups. One significant detail for this database is that we concatenate the street pre-directional, street name, street post-directional and suffix to form a single field. We use this concatenated field notion both in building the database and in constructing the query strings. The effect of this concatenation is to get around one of the trickier problems in database consistency. Sometimes it is very difficult to correctly label a word on a mailpiece as being either a street word or as a street suffix. For example, in the name "DEER CREEK" one might easily conclude that "CREEK" was the street suffix, whereas in the name "DEER CREEK RD" it is a street name word. By concatenating the

fields we get around that difficulty for matching. We also can sometimes match predirectional words against postdirectional words in a simple way using the same strategy. Currently we have a single database for all of the 183292 unique ZIP-street records in the 25 SCF database we are using for testing this system. The third application is in ZIP-business name lookups. Since there are over a million ZIP-business name combinations in our 25 SCF database, we ended up splitting this database in 25 pieces, one for each SCF, in order to get a reasonable query speed and to reduce the number of responses from outside the locality specified in the query.

We tested this system in December 1991 with a set of ASCII address blocks from 1200 mailpieces provided by the USPS along with a manually generated "correct" ZIP code to compare against. Using this ASCII data, our system was then able to generate 9-digit ZIP+4 codes for 91% of the mailpieces and 5-digit ZIP codes for the remainder. Compared with the manual truth values, our system generated the wrong ZIP code in only 4.6% of the mailpieces. The inexact matching approach described in this article was a key component in the system's success.

8. Acknowledgments

This work was sponsored by the Office of Advanced Technology of the United States Postal Service under contract number 104230-86-H-0042.

9. References

[BLOOM70] Bloom, Burton H., "Space/time tradeoffs in hash coding with allowable errors"*Comm. ACM* 13,7 (July 1970), 422-426.

[HULL82[Hull, Jonathan J., and Srihari, Sargur N., "Experiments in Text Recognition with Binary n-Gram and Viterbi Algorithms", *IEEE. Trans. on PAMI* 4, 5 (Sept. 1982), 520-530.

[KNUTH73] *The Art Of Computer Programming*, Vol. 3, pp. 561-562.

[OWOLABI88] Owolabi, O., and McGregor, D. R., "Fast Approximate String Matching", *Software Practice And Experience* 18,4 (April 1988) 387-393.

[VAYDA92] Vayda, Alan J., and Cavnar, William B., "A System for ASCII Address Interpretation", *Proceedings of the Fifth Advanced Technology Conference*, Wash. D.C., 1992.