Extremely Fast Text Feature Extraction for Classification and Indexing

George Forman, Evan Kirshenbaum

HP Laboratories
HPL-2008-91R1

Keyword(s):
text mining, text indexing, bag-of-words, feature engineering, feature extraction, document categorization, text tokenization

Abstract:

Most research in speeding up text mining involves algorithmic improvements to induction algorithms, and yet for many large scale applications, such as classifying or indexing large document repositories, the time spent extracting word features from texts can itself greatly exceed the initial training time. This paper describes a fast method for text feature extraction that folds together Unicode conversion, forced lowercasing, word boundary detection, and string hash computation. We show empirically that our integer hash features result in classifiers with equivalent statistical performance to those built using string word features, but require far less computation and less memory.

External Posting Date: August 21, 2008 [Fulltext] Approved for External Publication
Internal Posting Date: August 21, 2008 [Fulltext]

To be published and presented at Conference on Information & Knowledge Management, Napa, CA Oct 27, 2008

Extremely Fast Text Feature Extraction for Classification and Indexing

George Forman
Hewlett-Packard Labs
Palo Alto, CA, USA
ghforman@hpl.hp.com

Evan Kirshenbaum
Hewlett-Packard Labs
Palo Alto, CA, USA
evan.kirshenbaum@hp.com

ABSTRACT

Most research in speeding up text mining involves algorithmic improvements to induction algorithms, and yet for many large scale applications, such as classifying or indexing large document repositories, the time spent extracting word features from texts can itself greatly exceed the initial training time. This paper describes a fast method for text feature extraction that folds together Unicode conversion, forced lowercasing, word boundary detection, and string hash computation. We show empirically that our integer hash features result in classifiers with equivalent statistical performance to those built using string word features, but require far less computation and less memory.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing. I.5.2 [**Pattern Recognition**]: Design Methodology– *feature evaluation and selection.*

General Terms

Algorithms, Measurement, Performance, Experimentation.

Keywords

text mining, text indexing, bag-of-words, feature engineering, feature extraction, document categorization, text tokenization.

1. INTRODUCTION

Most text analysis—such as document classification or clustering—includes a step of *text feature extraction* to determine the words or terms that occur in each document. This step is straightforward, runs in linear time, and is generally not a topic of study. Research towards fast and scalable methods tends to focus on algorithmic issues such as model induction, typically O(N^2) in the number of training cases. By contrast, scoring documents in production with a learned model is merely O(N)—yet for a different meaning of N, the number of *production* documents, which can be very large for enterprise-scale applications or when analyzing the world's web pages. The scoring time (aka *testing time*) for large scale deployment can easily dominate the induction time, especially for the common case where it is difficult to obtain a large quantity of training examples labeled by domain experts. An example would be an Information Lifecycle Management (ILM) application that periodically applies classifiers to huge document repositories for content management, such as the automatic application of file retention, archiving and security policies. Our recent research in this space found that the time to extract the words from a text file can be roughly on par with the time to fetch the file from a local disk [5]. Finally, full-text indexing also must perform text feature extraction on large volumes of files or web pages. Hence, text feature extraction can take considerable computational resources for large scale systems.

The tremendous increase of online text annually, together with the proliferation of large scale text analysis applications, yields a valuable opportunity to speed up the core text scanning subroutine that is so ubiquitous. We demonstrate a way to perform this scanning step up to two orders of magnitude faster, with little or no effect on the quality of the text analysis. The method, which includes character table lookup and hashing in its inner loop, inexpensively supports forced lowercasing, multi-word phrases, and either word counts or boolean features, as desired. It can also supplant common Unicode conversion, eliminating some buffer copying and character expansion. We demonstrate its use and speedup in applying multiple classifiers to texts, without having to write out different feature vectors for different feature-selected classifiers. The method yields word and phrase features represented as hash integers rather than as strings.

The obvious use for faster feature extraction is to process more text per second, run more classifiers per second, or require fewer servers to handle the text processing load. Alternately, where the rate of text per second is limited, the benefit may be to lower the impact on a user's machine, e.g. where text analysis agents operate constantly in the background to 'understand' the texts the user is reading or writing.

The following section describes both baseline and proposed methods for text feature extraction. Section 3 provides an empirical comparison for speed and classification accuracy. Section 4 analyzes the collision behavior and memory requirements. Section 5 discusses practical matters for real-world use: the effect for end-users and an extension of our method that folds in the processing of text encoded using Unicode UTF-8 at little incremental cost. Section 6 places this work in the broader context of related work. Section 7 gives conclusions and future work.

Table 1. Baseline text word extraction.

```
HashSet<String> extractWordSet(text):
  1 HashSet<String> fv = new HashSet<String>();
  2 StringBuilder word = new StringBuilder();
  3 foreach (char ch: text) {
  4    if (isWord(ch)) {
  5            word.append(toLowerCase(ch));
  6    } else {
  7            if (word.size() > 0) {
  8                    fv.add(word.toString());
  9                    word.clear();
 10            }
 11    }
 12 }
 13 return fv;
```

Table 2. Basic SpeedyFx text feature extraction.

```
boolean[] extractWordSet(text):
  1 boolean fv[] = new boolean[N];
  2 int wordhash = 0;
  3 foreach (byte ch: text) {
  4    int code = codetable[ch];
  5    if (code != 0) {// isWord
  6            wordhash = (wordhash>>1) + code;
  7    } else {
  8            if (wordhash != 0) {
  9                    fv[wordhash % N] = 1;
 10                    wordhash = 0;
 11            }
 12    }
 13 }
 14 return fv;
```

2. METHODS

We begin by laying out the fundamentals of text feature extraction and describe a straightforward baseline method. After this, we describe the steps of our method. An extension for processing Unicode UTF-8 texts is included in Section 5.2.

Text feature extraction depends on some definition of which characters are to be treated as word characters vs. non-word characters. Besides the letters A–Z (in both upper and lower cases variants), word characters may also include accented letters, digits, the underscore, and generally all Unicode characters having one of the Unicode "Letter" general categories (uppercase, lowercase, titlecase, modifier, or other), depending on the application. Let the boolean function isWord(char) determine the status of any given character. For this paper, we take the common approach that a maximal sequence of adjacent word characters in the text stream constitutes a word.

Typical text processing applications normalize the capitalization of each word by forcing each character to lowercase. Let this conversion be determined by the character function toLowerCase(char). The mapping is non-trivial for some Unicode letters. Additionally, if underscores or digits are included among the word characters, then their lower case mapping is the identity function.

For indexing applications, one needs to determine the sequence of each word that appears, but for text classification or clustering applications, one typically distills the text to the well-known *bag-of-words* as the feature vector representation—for each word, the number of times that it occurs, or, for some situations, a boolean value indicating whether or not the word occurs. The feature vector needed by a classifier depends only on the words that occurred in the training set, and in some cases, may use only a subset of these. Feature selection methods may be used to select a subset of the most predictive words in order to improve the accuracy of the trained classifier [4]. This selection depends on the class labels of the training set, and if multiple classifiers are to be applied to the same text, then each classifier may require a different selection of features for its feature vector.

Text classifiers can often be made more accurate if they also include features that represent word phrases, the most benefit coming from 2-word phrases (aka bi-grams) with diminishing returns for longer phrases [12]. By including word phrases, the dictionary of potential terms is greatly increased, emphasizing the need to select only the most useful terms for the classification task at hand. It is not uncommon for a training set to contain hundreds of thousands of distinct terms and for the final classifier to operate with a subset of, say, 1,000 of the best terms.

2.1 Baseline Method

Table 1 presents pseudo-code that performs text word extraction for boolean features in a straightforward manner. Although many variations and minor optimizations are possible, the common thread of all baseline methods is to gather the contiguous sequence of word characters together in an accumulator buffer (line 5) until the end of the word is encountered, and then to register the word String with some feature vector representation (line 8).

This pseudo-code computes the *set* of words, but if instead one required the bag-of-words, then the appropriate hash table would be a HashMap<String,Integer> that maps each occurring word to the number of times it occurred (or to a position in an array of counters for faster incrementing). Note that to add a word of type String to a hash table of words will require a hash function to be computed over the String that is used as an index into the hash table array, and may require at least one String-vs-String comparison in order to verify that the correct hash table entry has been located or to differentiate this word from other words whose hash representations collide in the hash table representation. We are able to eliminate this overhead with our method.

2.2 Our Method: SpeedyFx

Table 2 presents equivalent pseudo-code for the basic skeleton of our method, which we call SpeedyFx. In structure it appears much the same as before, but with two key differences. Foremost is that the current word accumulator is represented by an integer hash (line 2 & 6) instead of a String. (We analyze the strength of our chosen hash function in Section 4.1.) The second major difference is that all function calls have been removed from the inner text processing loop. A lookup (line 4) into a pre-computed table replaces two much slower function calls: isWord(char) and toLowerCase(char). Even if one could get these functions compiled inline, their joint processing overhead would greatly exceed a single table lookup. And the many word occurrences are simply overwritten to the boolean feature array (line 9), avoiding many calls to the hash table add() function inside the loop. Note that there is no need to check whether the particular word hash has already been noted in the feature array. (In systems that want a count of each word, the array can be an array of integers rather

Table 3. Code table pre-compilation for SpeedyFx.

```
prepTable():
  int rand[256] = 256 fixed random integers;
  foreach (ch: 0..255) {
     codetable[ch] = isWord(ch) ?
             rand[toLowerCase(ch)] : 0;
  }
```

than an array of booleans.) In practice, N will be chosen to be a power of two, allowing the modulus operation (line 9) to be replaced by a simple bitwise AND.

If a more compact representation of the resulting feature vector is required, a quick, linear scan of the boolean array can be used to insert each occurring word hash into a hash table or into an array containing the word hashes found. In many cases, though, features are extracted in order to be used immediately for some task, an in such cases the boolean array representation is often sufficient.

A reasonably large bits array of size N=2^{20} is suggested. Larger values of N result in a greater memory demand momentarily for the bits array, whereas smaller values of N effectively restrict the size of the hash space, conflating features that have different 32-bit integer hash values together into a smaller space. Using N=2^{20} is easily sufficient for corpora or training sets with hundreds of thousands of unique terms. For perspective, a corpus of 804,414 Reuters news articles contains 47,236 unique words, and a corpus of 62,369 Arxiv.org astrophysics papers contains 99,757 unique words [8]. Regarding the memory demands, most compilers will allocate one byte per boolean for good performance, rather than packing them tightly into bits. Thus, for N=2^{20} the size of this bits table is 1 MB, which is a fraction of the L2 cache for modern CPU chips. If instead one needs to track the word count for the bag of words representation, this array can instead be used to contain 8-bit or 16-bit word frequency counters.

The pre-computed code table actually serves four different functions simultaneously: word character determination, lowercasing, hash randomization, and, as will be introduced later in Section 5.2, Unicode processing. Table 3 shows pseudo-code to prepare the table appropriate for 8-bit character streams. For any character that is not a word character, the table contains zero. For word characters, it maps the character to its lowercase equivalent and maps the result a distinct fixed random integer code. This randomization substantially improves the hashing function over just using the lowercase character code directly, as the Java String.hashCode() method does. (For reference, Java Strings use multiplicative hashing [10], computing their hashes by iterating *hash* = 31**hash* + *char* over their characters).

It should be noted that while in our tests our word characters were letters and numbers an our normalization is lowercasing, this is all embodied in the creation of the code table. Any other set of rules could be used by providing a different table. The only restriction is that the mapped and normalized character (if any) be decidable from a single character. This can be relaxed, but the mechanisms for doing so are beyond the scope of this paper.

Our hashing function, which we call Mapped Additive Shift Hashing (MASH), consists of a simple arithmetic right shift, a table lookup (from a very small table), and an addition. As such, it is very efficient and easy for a compiler to optimize. Although the rightward direction of the bit-shift operation (line 6) is somewhat non-intuitive, it actually produces substantially better hashes than using a left-shift. The reason is that we end up using only the low order bits of the resulting word hash (line 9), and if a left-shift were used instead, then these lowest order bits would be largely unaffected by the first characters of the word, especially for a longer word. (In the case in which the modulus is a power of two for efficiency, they would be completely unaffected by them.) If the rightmost ten bits are kept, then only the final ten characters can contribute to the hash. By using a right-shift, on the other hand, and by ensuring that each character contributes 32 bits to the value, a character's influence continues (in a degrading fashion) for at least 31 following characters. Even for the occasional word longer than 32 characters, note that we use a sign-extended right-shift, so that the parity of initial letters continues to have some effect on the hash. Of course, this influence does diminish, and so MASH is certainly not suitable for hashing long texts, which is the goal of typical hash function designs such as SHA1, MD5 and Rabin hashing, which are each much more involved to compute.

If one wishes to also produce features for each phrase bi-gram, then at the moment when the hash is registered (line 9), we can also register a phrase hash consisting of the current hash combined with the previous word hash, using a symmetric combination (such as XOR or addition) if it is desired to have features that are independent of the order of the words or an asymmetric combination (such as subtraction or XOR with a shifted value) if order is to be preserved. Working in integer space, such combination is much more efficient than working in string space, in which typically a new string must be constructed (or at least a new heap-based object that refers to the two strings).

2.3 Use in Full-Text Indexing

So far the discussion has primarily focused on the use case of extracting a bag or set of words, which is needed for use in classification or clustering applications. But the speed benefits of our method can also be leveraged for the widespread use case of full-text indexing of large document repositories (either large documents or many of them or both). In this case, the indexing engine needs to extract the sequence of words/terms in a text, and build reverse indices of them for later information retrieval. The key difference for our features would be that the term features are integers rather than Strings. When later a query is submitted, the words that the user enters are submitted to the exact same hash processing, yielding the same integer hashes that the text has previously been indexed by. Note that for this usage, one can use the full 32-bit integer hash space (i.e. without the modulo N and the bit array on lines 1 and 9 of Table 2).

3. EXPERIMENTS

In this section we present a series of experiments to evaluate the speed and other qualities of our method compared with various baselines. In all cases our compiler is Java JDK 1.6. Where speed measurements are concerned, the measurements were performed on a 2.4 GHz Intel® Xeon™ CPU, which is a 64-bit architecture, running a version of Linux derived from Red Hat version 4. Measuring elapsed time for performance can be quite tricky, since other processes and threads regularly interrupt processing for uncontrolled amounts of time, occasionally slowing down the overall or average measurement substantially. For this reason, we measure the throughput for each test over 50 separate iterations and use the measurement that shows the best performance, which excludes non-essential delays for thread interruptions. To exclude the impact and variability of disk transfer speeds, we cache the file bytes in memory before beginning the series of time measurements.

We use two sources for publicly available benchmark texts. To measure speed for tokenization, we use the text of Leo Tolstoy's *War and Peace*, which is 3.1 MB in ASCII and is available from Project Gutenberg.[1] To measure speed over feature extraction and classification, we use the popular 20 Newsgroups dataset, which has roughly 1000 Usenet postings for each of 20 categories.[2] We use the 'bydate' version, which has had duplicates and some headers removed and which prescribes a specific train/test split. The dataset contains 34.2 MB of mostly-ASCII text in 18,846 documents. Thus the average document is about 1.9 KB.

3.1 Raw Tokenization Speed

First we measure the raw speed of text tokenization. For this measurement, we simply produce the sequence of word Strings or word hash integers that occur in the text and discard them without further processing. This is the most basic processing skeleton that needs to be computed regardless of the text analysis application. For example, in a text indexing system, this sequence of terms would be used to build a reverse index by later sorting all <term hash, document ID> pairs.

Table 4 shows the processing speed for a variety of methods measured on the *War and Peace* text. The first line shows our best processing rate of 136.2 MB/second using SpeedyFx with MASH and calling down to a user-provided function on each word hash found. The same speed was attained using an inlined version of Java's hash algorithm. When Rabin fingerprinting was substituted, the performance dropped to 111.9 MB/second..

Typically tokenization is performed by using an iterator-like class that returns successive tokens each time a method is called. When our algorithm was embodied in such a class, returning a word hash each time, the performance dropped to 116.0 MB/second using MASH and Java's hashing and to 87.0 MB/second using Rabin fingerprints. Since both approaches entail a single method call per discovered word, this speedup of approximately 17% appears to imply that the Java compiler is more aggressive about inlining calls to methods within the same object for (necessarily final) anonymous classes than it is to calls of objects of different classes.

All of these numbers compare very favorably with existing baseline approaches to tokenization. Java provides a String-Tokenizer class, which takes a string and a string containing delimiter characters (characters to not be considered part of any word) and provides hasMoreTokens() and nextToken() methods. Besides the overhead of the extra call to find out whether more words are available, this suffers from several problems common to many approaches we looked at.

Table 4. Raw text tokenization speed.

Tokenizer	MB/second	%
SpeedyFx	136.2	100%
SpeedyFx as Iterator	116.0	85%
Table-based tokenizer	64.5	47%
Lucene CharTokenizer	25.7	19%
WEKA alphabetic tokenizer	21.8	16%
Java Pattern	11.1	8%
Java StringTokenizer	4.1	3%
Lucene StandardAnalyzer	1.9	1%

First of all, it requires a string as input rather than being able to work directly from a byte array. This involves converting bytes to characters and creating a string from the characters, both of which imply copying the data. Second, each of the produced tokens needs to be created as a String object.[3] And third, any normalization (in this case lowercasing) needs to be done externally on each produced string. In addition, the StringTokenizer class searches in its delimiter string for each character it sees. The class is designed for use with a small set of delimiters (e.g., commas and tabs), but for our purposes, we need to include all non-alphanumeric characters, and so this check is quite slow. The net result is that StringTokenizer processed *War and Peace* at a rate of only 4.1 MB/second, just 3% that of SpeedyFx.

A slightly more efficient approach is to use a regular expression, in the form of a Java Pattern object matching all Unicode non-alphanumeric characters, to split each line and then lowercase each word.. This processed *War and Peace* at 11.1 MB/second, or 8% of SpeedyFx.

Another approach is to use classes supplied by a library. The Lucene v2.2 library [7] provides two methods. The most commonly used of the two is their StandardAnalyzer class, but it is not strictly an apples-to-apples comparison, since that class incorporates a full lexical analyzer (built from JavaCC [3]), which attempts to recognize more complex objects, such as e-mail addresses and hostnames, which ours cannot. On *War and Peace*, the Lucene StandardAnalyzer performed at 1.9 MB/second, 1% of SpeedyFx.

Lucene also provides set of Tokenizer classes to allow the programmer to specify the rules directly, and a CharTokenizer subclass that defines tokens as sequences of allowable characters, arbitrarily normalized. We tested a subclass that allows alphanumeric characters and normalizes by lowercasing. Unlike tokenizers that require strings as input, this can read directly from a character stream and so performs much better, processing *War and Peace* at a rate of 25.7 MB/second (19%).

[1] Version 11, http://www.gutenberg.org/dirs/etext01/wrnpc11.txt. The size includes Project Gutenberg boilerplate text at the beginning and end.

[2] http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz

[3] In the case of the StringTokenizer class, the tokens are substrings of the input string, which are themselves small, but which refer to the larger string. This would appear to have severe memory implications if the tokens are stored in long-lived structures, as they would prevent the memory for the full text string from being garbage collected.

The popular Weka v3.4 [16] library contains a class StringToWordVector to convert a dataset with String attributes into bag-of-words features. At its core, it uses its own text tokenizer AlphabeticStringTokenizer, which has specialized code for recognizing sequences of (only) ASCII alphabetic characters. When put in a framework that processes a stream and does lowercasing of the words it finds, it processes *War and Peace* at a rate of 21.8 MB/second (16%).

One question that arises is the extent to which the speed-up we see is due to the table-driven approach rather than the decision to use hashing rather than creating strings. To test this, we constructed a tokenizer that used the same tables as SpeedyFx but which constructed a string rather than computing a hash. This processed *War and Peace* at 64.5 MB/second, 47% of the rate at which the full algorithm worked. A similar approach, suggested by a reviewer, is to use pre-computed arrays in place of the isWord() and toLowerCase() functions (to remove the function call overhead) and to yield the hash of the constructed string rather than the string itself. This approach processed *War and Peace* at 31.6 MB/second, 26% of the rate of the full algorithm.

3.2 Feature Vector Extraction Speed

Next, we measure the text processing speed for text feature extraction, i.e. to determine the complete set or bag of terms in a text, which is a task typically performed during clustering analysis and classification, especially during the training phase (i.e. before any feature selection has been performed). For these tests, we use all 18,846 articles of the 20 Newsgroups benchmark and measure the time it takes to extract a representation of the *set* of words contained in each article, where a word is, as before, considered to be the lowercase version of a maximal sequence of alphanumeric characters (or the closest that can be approximated using a given method). Over and above the tokenization task, this task adds the problems of (1) recognizing that a word has already been seen and (2) constructing a representation that can be used to enumerate the words seen and to decide whether a word of interest was seen.

For methods that enumerate strings, a Java HashSet<String> was chosen as the representation, while for methods that enumerate hash values, an array was used. The size of the array was 2^{20} elements.

Table 5 shows the text extraction speeds for extracting sets of words. Again, we see that extraction using a hash-based tokenizer runs many times faster (9–65x) than the existing string-based methods, and the difference is even greater than with tokenization. This is not surprising, as the use of the hash set imposes its own significant overhead over simply writing a value in an array. First of all, a hash value must be computed over the string to be added, which is linear in the length of the string. Then, if there are already entries in the appropriate slot in the hash table's array, each must be compared against the current string, which is a constant-time operation unless the strings' (recorded) hash values match. And when a word is seen for the second or subsequent time, a full string comparison must be performed to determine that a new entry does not need to be added, which is again linear in the length of the string. Finally, whenever the table's capacity threshold is exceeded, it needs to be grown, which is linear in the number of elements currently in the table. This additional overhead due to the hash table explains why even when using the table-based tokenizer, the performance is only marginally better (11% vs. 151% for tokenization) than that of the fastest other string-based extractor.

Table 5. Feature vector extraction speeds.

Extractor	MB/second	%
SpeedyFx (with case index array)	117.1	100%
Table-based tokenizer	15.2	13%
Lucene CharTokenizer	13.7	12%
WEKA alphabetic tokenizer	11.1	9%
Java Pattern	7.3	6%
SpeedyFx (with boolean array)	2.7	2%
Lucene StandardAnalyzer	1.8	2%

Table 5 illustrates another interesting point about extraction using hash-based tokenization. SpeedyFx methods show up both at the top and near the bottom of the table. The numbers at the bottom are for the method essentially as described in Table 2, in which the feature vector representation is an array of boolean values which are set to true each time a hash computation is completed. When run on *War and Peace*, this method had essentially the same performance (116.0 MB/second vs. 120.5 MB/second) as the faster method.

The problem is that while it is, indeed, inexpensive to set the array item to true each time a word is seen, in order to be usable as a feature vector, this requires that all array items are set to false before the process begins. With 2^{20} elements in the array needing to be reset (or initialized) for each of 18,846 cases, this meant that 19.8 billion false values were being set in order to distinguish them from the approximately 6 million hash values actually seen–over three thousand times as many. Indeed, the 18.4 GB of feature vector initialization dwarfed the 34.2 MB of text that had to be processed. Clearly, the cost of initializing the feature vector (not as big a problem with hash sets, whose internal arrays are proportional to the size of their contents) needs to be taken into account when dealing with smaller texts.

We tried two approaches to get around this problem. In the first, we maintained a *trail*–a separate array of indices of the feature array values changed from false to true. This necessitated that before we wrote a feature value, we first checked to see whether we had already done so, but it also meant that to reset the feature vector to prepare it for the next case, we needed merely to walk a list proportional to the number of unique words seen. This improved the performance to 82.2 MB/second, or 70% of our eventual best speed.

The second—and best—approach was to replace the boolean array with an array of values indicating the document in which the corresponding hash value was last encountered. The extractor keeps track of the value indicating the current case and sets the appropriate array element to that value when a hash computation is completed. To check whether the current case contained a word hash, the array element is compared against the current case indicator. If they are the same, the answer is "yes".

For our implementation, the case indicator was a number, incremented with each new case and allowed to wrap on overflow. Doing it this way has an interesting side effect, though. If the last case in which a particular word hash was seen was exactly 256 cases ago (for byte indicators; 65,536 cases ago for short indicators), the feature extractor would incorrectly report that the case contained that word hash. In theory, this should not be expected to happen very often, but if the possibility is unacceptable, the array can be reset every 255 (or 65,535) cases, amortizing the cost of doing the reset. Doing so with byte indicators reduced the speed to 102.1 MB/second (87%).

3.3 Multiple Text Classifier Speed

As described earlier, often one needs to apply many text classifiers to each text, and our method can support this common use case efficiently. To test this, feature selection was performed on the training cases for each of the 20 newsgroups in the 20 Newsgroups dataset to extract the 1,000 most informative features for each using bi-normal separation as the metric and separately for word hash and string features. This resulted in 18,030 selected word hash features and 18,175 selected string features (some of which were selected by multiple classifiers). We then scored each article in the entire 20 Newsgroups dataset for each of the 20 classes, by summing weights associated with features seen (noting each weight only once per article), as would be the case when using an SVM or Naïve Bayes classifier.

In these experiments we compared SpeedyFx-based classifiers with string-feature-based classifiers. For the string-feature-based classifiers, we used hand-crafted classifiers based on the table-based tokenizer that consistently performed the best of the string-based methods in the tokenization and extraction tests. In addition, each of these methods was implemented three ways, each considering articles one at a time. The first two approaches began by performing feature extraction on the article. In the first approach, the resulting feature vector (array or HashSet) was *filtered* (based on the complete list of selected features) to an array of booleans. Each classifier was then asked to score the article based on this array. In the second approach, for each class, the feature vector representation was used directly to score the article.

The third approach skips feature extraction entirely. Instead, a single multi-class classifier is built that contains an *target set*, an internal data structure mapping features selected to the classes that selected them and the associated weights. (This data structure is an array for SpeedyFx and a hash map for the string-based method.) For each, the main extraction loop is performed, but now when a feature is determined, rather than adding it idempotently to a feature vector, the classifier checks the target set to see whether it contains the determined feature. If so, the classifier uses the associated classes and weights to update the appropriate scores. It then removes it from the target set and adds it to a trail, which is used to repopulate the target set to prepare the classifier for the next article. By removing the feature from the target set, the classifier ensures that if it occurs again in the same article, it will be ignored.

Table 6. Classification speeds (MB/second).

Approach	SpeedyFx		String-based	
extract, filter, classify	6.2	11%	1.2	2%
extract, classify	8.1	14%	1.3	2%
inline classify	57.7	100%	23.5	41%

As can be seen in Table 6, in all cases, SpeedyFx clearly dominated the string-based equivalents. Also, for both SpeedyFx- and string-based approaches, inline classification overwhelmingly dominated classification that begins with feature extraction.

3.4 Classification Accuracy

Ultimately the superior speed of any extraction method would be unimportant if the features it produced led to poor characterization of the data. In this section, we demonstrate that, on the contrary, there is substantially no difference in classification accuracy when using traditional bag-of-word features as opposed to using our hash features.

Figure 1 shows the F-measure (harmonic average of precision and recall) for each of the 20 binary "one-class-vs-others" classification tasks defined by the 20 Newsgroups dataset. The classifier used is the WEKA v3.4 Support Vector Machine (SVM) [16] using the best 4096 binary features selected separately for each training set via Bi-Normal Separation [4]. (We chose this

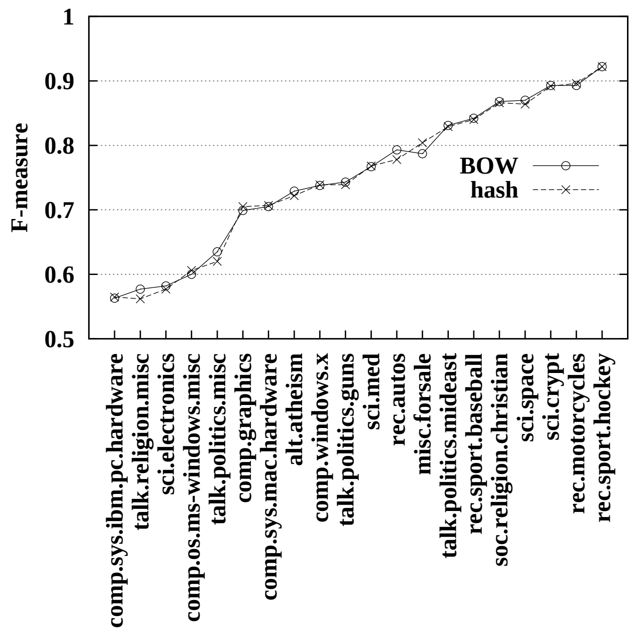

Figure 1. F-measure for each class for an SVM.

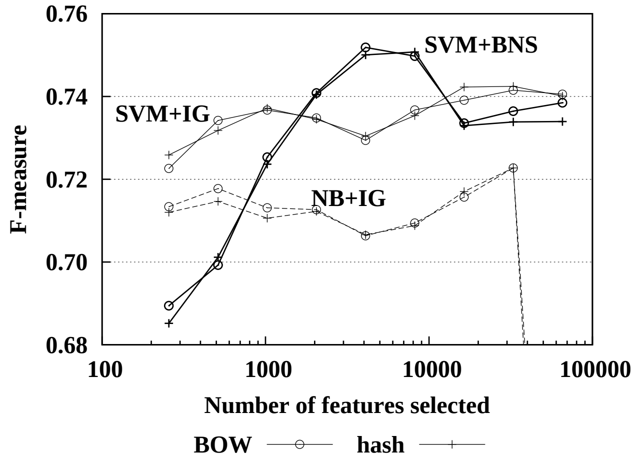

Figure 2. Macro-averaged F-measure is affected more by choice of classifier, number of features selected and the choice of feature selection metric than it is by whether we use bag-of-words features (BOW) or our hash features.

Table 7. Comparison of hash function quality over words.

	MASH	31*h+c	Rabin
Hashes seen	155,620	150,613	113,817
Colliding hashes	12,178	15,661	24,436
P(static collision)	0.0762	0.1059	0.3244
P(dynamic confusion)	0.0012	0.0065	0.0841

number of features to maximize macro-averaged F-measure overall.) The results show essentially no difference in performance between bag-of-words features (BOW curve) and our hash features (hash). A paired t-test over the F-measures for the 20 classification tasks shows that any apparent difference is statistically insignificant (p=0.31). Furthermore, we next show for perspective that other common factors affect the performance much more than the type of features generated.

Figure 2 shows the F-measure under various models macro-averaged over all 20 binary classification tasks. For each model, we see that the performance with bag-of-words features or with our hash features has a small effect (the two curves are close together). Overall, choice of feature generation has a much smaller effect on performance than the choice of classifier (SVM vs. multinomial Naïve Bayes), the number of features selected, or the feature selection metric by which to judge features (Bi-Normal Separation vs. Information Gain). This brings up an additional point of motivation for this work: If in practice one needs to try many different model parameters and select the best via cross-validation, then it may be even more important to the practitioner to have fast text feature extraction.

4. ANALYSES

4.1 Hash Collisions

For a good hash function, accidental hash collisions should be as rare as or rarer than misspellings in the text or naturally occurring homonyms (two words that are written the same but have different meanings, e.g. the 'boxer' dog breed vs. the 'boxer' athlete). Just as misspellings and homonyms occasionally detract from our text analysis goals, so too will hash collisions of unrelated words. We show in this section that our chosen hash function, MASH, is stronger than the ubiquitous multiplicative hashing, used by Java's String.hashCode() function, and the well-known Rabin hash function, which is generally purported to be fast to compute with the use of a pre-computed lookup table.

Table 7 compares the three hash functions in terms of the collisions over words in the 20 Newsgroups dataset, considering only the bottom 20 bits of each hash value. The 20 Newsgroup dataset contains 5,984,079 "words" (maximal sequences of alphanumeric characters) which comprise 168,461 distinct (lowercased) values. The first two rows are the number of distinct hash values seen using the given hash function and the number of those values that are the result of hashing distinct words. (The numbers do not add up to the number of distinct words since a small number of values will be shared among more than two words).

The third line is the resulting probability that a given word will result in a hash value that is shared with some other word in the corpus. This is a *static* probability, which does not take into

Table 8. All pairs of frequent words confused by MASH.

key (3,395) vs (1,075)	pp (443) 96 (199)	assembly (173) responded (107)
research (2,044) easy (841)	xt (393) sharks (207)	senator (155) helsinki (113)
heard (1,607) mu (280)	club (290) nih (141)	brings (120) iq (112)
90 (712) medium (108)	dseg (268) lying (183)	scope (115) badly (107)
vms (645) floppy (482)	discovered (249) watson (218)	
alone (501) duke (424)	drink (175) signals (160)	

account word frequency. As can be seen from the table, words are considerably less likely to share their 20-bit hash values using MASH than using either Java's string hashing or Rabin fingerprinting. Only approximately one in 13 words has a shared hash value for MASH, compared to nearly one in ten for Java string hashing and nearly one in three for Rabin fingerprinting.

But static collisions are not the whole story. In any corpus, the the words will tend to follow a Zipf distribution with the vast majority occurring very seldom and only a small minority occurring with noticeable frequency. Since hash collisions do not depend on the frequencies of the words, this implies that for the vast majority of collisions, one or (much more likely) both will be very infrequent. For instance, using MASH, the words "bit" and "wx3" both hash to 128,278, but the former occurs in 2,597 articles, while the latter appears in only one. The implication for classification is that a colliding hash is either unlikely to be selected as a feature (since both words are infrequent) or will almost always represent the word that led the classifier to select it.

To take this into account, we consider a measurement of a probability of "word confusion" that focuses on the dynamic impact of collisions. This is the conditional probability that if two words chosen at random from the corpus hash to the same value, they are, in fact different words. It is computed as

$$P(\text{confusion}) = 1 - \frac{\sum_{w \in \text{Words}} n_w(n_w - 1)}{\sum_{h \in \text{Hashes}} n_h(n_h - 1)}$$

where n_w and n_h are, respectively, the number of occurrences of words and hashes in the corpus. As is apparent in Table 7, this is over five times as likely to happen with Java string hashing (and 71 times as likely to happen with Rabin fingerprinting) as it is with MASH.

The actual collisions seen appear to be unlikely to cause much problem. Looking at MASH with the "rec.motorcycles" group, the most significant term is a collision between "bikes", clearly a relevant word, which occurs in 233 articles in the corpus and "kom", which occurs once. Of the top 1,024 features for the group, in only two is there a collision between words both of which occur more than ten times. The first such collision, the 1,001st most important feature as selected by Bi-Normal Separation, is a collision between "noemi" (13 articles) and

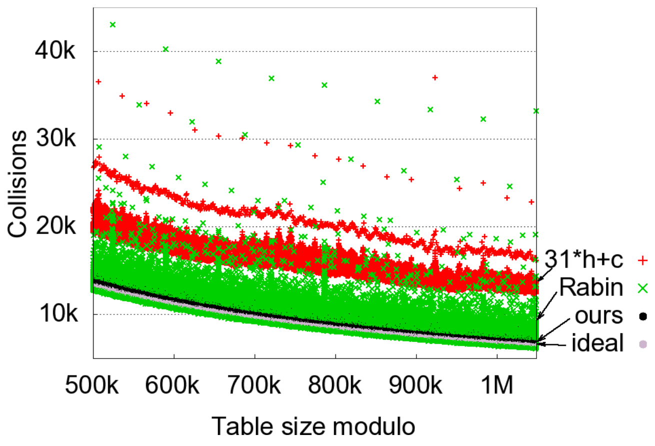

Figure 3. Number of hash collisions modulo table size.

"8mhz" (11). The second such collision is an actual example of what might be considered the introduction of a homonym: "sin" (692 articles) and "executed" (83 articles) both hash to 998,868.

In the entire corpus of 168,461 words, there were only sixteen pairs of words that each occurred in at least 100 articles and which hashed to the same value using MASH. These, shown in Table 8, represent just 0.01% of the 155,620 hash values seen and 0.13% of all (static) collisions. If the threshold is lowered to those that have at least ten instances each, the number of such hashes rises to only 367 (0.24%).

So far, the analyses have all used a bit table space of $N=2^{20}$. Naturally, the collisions will increase if we decrease N, e.g. in order to reduce memory demands for small portable devices. Figure 3 shows the number of collisions produced by the set of words and phrases used previously, as we vary the table size modulus N from 500,000 to 2^{20}. Most noticeably, the 31*h+c multiplicative hash function used by Java has erratic performance. It gets many more collisions for particular values of N—multiples of 31 (the JDK 1.6 HashMap and HashSet implementations avoid such table sizes by always using a power of two). Likewise, the Rabin hash function also shows erratic performance, though it very often has fewer collisions than the multiplicative hash function. Finally, observe that MASH performs consistently well at every table size (labeled 'ours'). For a near-lower bound comparison, we repeated this experiment for a random set of integers and find that their collision performance is only marginally better than MASH (labeled 'ideal').

For indexing applications, in which the task is simply to extract the complete sequence of tokens, it is perfectly reasonable to store much larger hashes of 4 to 8 bytes per token. Using 6-byte hashes, one would not expect to see a single collision in fewer than nineteen million distinct words, assuming hash uniformity. For a web indexing application with a vocabulary of, say, five million words, 48 bits would suffice to give you a probability of 96% of not having a single collision.

4.2 Size

In addition to processing speed, another efficiency consideration is size. In the present context this can be interpreted in two ways: (1) the size of a representation of a classifier having a certain number of selected features and (2) the amount of memory that is needed to perform feature extraction or classification. In what follows, we give a summary of our results. Readers interested in the details of the analysis are invited to see Appendix 10.1. The analysis presumes a straightforward implementation using Java JDK 1.6 on a 32-bit machine. It also assumes the use of 20-bit hashes.

The size of a classifier representation has implications primarily with regard to transmitting the classifier from one machine to another or writing it to permanent storage. This will be especially important in settings in which there are a large number of classifiers being used. With hash-based methods, classifiers can be represented with 7 bytes per selected feature (4 bytes for the weight and 3 bytes sufficient to identify the feature hash). With string-based methods, the number will depend on the distribution of strings. For the 20 Newsgroups dataset, for which words average 6.5 characters, a classifier can be represented on the wire with 11.5 bytes per selected feature on average (4 bytes for the weight and 6.5 UTF-8 bytes on average, plus a null terminator), for an increase of 64% over the use of hash-based features.

The second concern is the memory footprint used temporarily during feature extraction and classification. For SpeedyFx, the memory used will be an array of either one or two bytes per possible hash code, so a 20-bit hash space will require 1 MB or 2 MB. For string-based methods, we analyzed the memory used by the HashSet feature set representation. Our analysis implies that a HashSet of Strings will require for each word approximately 80 bytes + 2 bytes/character, or 93 bytes per word on average for the 20 Newsgroups dataset. The articles in the corpus average 317.5 distinct words apiece, and so processing them should require a HashSet whose overall memory footprint is approximately 29 KB, substantially less than SpeedyFx. For longer texts, the difference is not as clear-cut. The 17,816 different words in *War and Peace* average 7.5 characters long, and overall the HashSet can therefore be expected to require 1.6 MB.

One further consideration is that the arrays used by SpeedyFx are contiguous and self-contained. The HashSets used by string-based methods, on the other hand, comprise hundreds or thousands of individual small objects which will not exhibit the same degree of locality and which will need to be individually garbage collected when no longer needed. In addition, while the 1 or 2 MB size of the SpeedyFx array is larger than the HashSets used by string-based methods for small texts, it is small enough to fit in cache on most modern architectures.

5. DISCUSSION

For perspective on the speed improvement, consider the motivation of enterprise-scale text indexing from the introduction. For each 100GB of enterprise files to be indexed, the off-the-shelf StandardAnalyzer in the Lucene package would spend 15 hours of CPU time just to tokenize the text. Even going to the effort of hand-coding an extension to Lucene's CharTokenizer would only reduce this to just over an hour. By contrast, with SpeedyFx this comes down to less than 13 minutes (from Table 4).

The remainder of this section contains a discussion of the practical impact of occasional hash collisions and an extension to SpeedyFx that folds in the ability to handle Unicode UTF-8 text with minimal loss of efficiency.

5.1 Impact of Hash Collisions

Hash collisions, when not too common, behave similarly to homographs. Consider first information retrieval: a search for documents containing the word 'shift' will return some about 'gear shift,' some about 'night shift,' and others about 'shift' in the slang sense of 'move it.' We usually discover the confusion after viewing the initial search results, and then use natural means to cope with these situations. We may add disambiguating search terms such as 'shift AND (gear OR transmission)', we may place the word in an unambiguous search phrase such as 'gear shift,' or rarely, we may give relevance feedback to refine the hits without caring which terms are involved. The major difference for hash features is that our background knowledge does not provide intuition for which words will have troublesome pseudo-homographs due to random hash collisions. If we search for 'cat' and we get an equal mix of documents containing 'strawberry,' then we may be surprised, but the same coping techniques apply as above. With our intuition, we might know that 'shift' is going to be ambiguous and plan for it from the very first query, but for the 'cat' example, we could not know in advance. Since most words occur rarely according to a Zipf distribution, it is much more likely that the pseudo-homograph would be some more obscure word. For example, if we search for 'cat' and we also get documents containing 'schadenfreude,' then the precision of our results may hardly be noticeably polluted, especially when considering the already difficult signal-to-noise problems in everyday, practical searches.

Next, consider the homograph-collision problem for text classification: the classifier is sure to use a variety of terms together to predict the class. Thus, a small amount of term confusion (typically from rarely occurring words) is unlikely to impact classification accuracy significantly. Nevertheless, it could occasionally happen that a particularly important word for classification is conflated with a very common word, such as the common stopword 'the', which ruins the predictive value of the joint feature. So, although there may be no substantial degradation for the *average* text classification problem, hash collisions may impair classification accuracy for a single, specific classification task at hand. A potential work-around would be to try a different initial hash seed to avoid the unfortunate collision. This workaround is not practical for the information retrieval scenario, where the queries of interest are not known until after the large document repositories have already been indexed with a fixed hash function.

5.2 Unicode extension

Although traditionally most text processing has been done on 7-bit ASCII or 8-bit characters of various encodings, more and more one must process texts encoded using the Unicode character set [15], either in the fixed-width two-byte UCS-2 encoding or in the variable-length UTF-8 encoding. Typical software for Unicode text processing, especially in Java, first calls a function to translate a UTF-8 byte stream into a buffer of 16-bit Unicode characters, upon which all remaining text processing is done. But this requires a substantial amount of copying and writing to a large block of buffer RAM (writing is slower than reading for most modern CPUs).

The table-driven MASH algorithm can be easily modified to handle both the 16-bit characters and the variable-length UTF-8 encoding, working directly on the byte stream and with only a minor reduction in throughput. The modifications do not entail the construction of a 65,536-element code table, but take advantage of the fact that most texts contain characters from only one or a few code pages, often ASCII or Latin-1 (on page 0) and

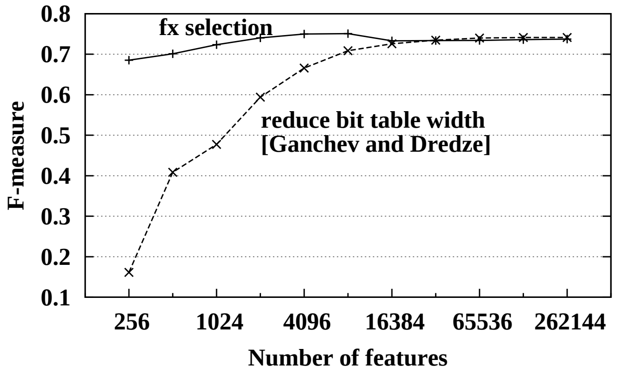

Figure 4. Eliminating features by selection vs. collisions.

one other, e.g., Greek characters on code page 3 or Devanagari characters on code page 9. Because of this, the table can be sparse and constructed lazily. The reader is invited to see the full description in Appendix 10.2.

6. RELATED WORK

The restricted problem of searching text for a previously selected set of useful features for a classifier may be cast in a broader context that is generally familiar in computer science. The problem of searching for a limited number of known query strings in large volumes of text is a traditional *string search* problem. The most well-known algorithms that treat multiple strings simultaneously and do not first require extensive indexing of the target text are the Aho-Corasick algorithm [1], the Commentz-Walter algorithm [2], and the Rabin-Karp algorithm [9]. Although they each have an expected run time linear in the size of the target text, they were designed in a historical context where the number of query patterns was in the dozens (e.g. information retrieval via UNIX grep). Research towards scalable methods that handle thousands of query patterns continues (e.g. [14]), but all these algorithms are expected to operate under general-purpose conditions, i.e. no restrictions on the query strings. In our setting, we can be sure that a query string is a word (or a bi-gram), which eliminates a great deal of dictionary checking. Another major difference is that for many statistical text processing applications, we can accept a certain risk of false positive matches in the form of hash collisions, considering that the natural misspellings, homographs, and variations in word choice in real-world text. Thus, for our purposes, traditional string search algorithms are not competitive, nor do they address the more general problem of text feature extraction for unrestricted words.

Just recently there have been two papers that promote hash features for classification, though neither focuses on text extraction speed, supervised feature selection, or applications besides classification, as we do. A short paper by Ganchev and Dredze [6] suggests using Java's String.hashCode() function modulo N for text classification features. Their focus is to obtain small classifiers for mobile devices with little RAM: the description of a linear classifier then needs only retain an array of weights—the index implicitly names the associated feature rather than a String word. The main result is that the number of features N was reduced to 10,000–20,000 without substantial loss in classification accuracy. To this we note that feature selection methods provide a principled approach for limiting the number of features, and classifiers with the best 500–2000 features are often more accurate than models using all features [4]. We illustrate this point in Figure 4, which compares the F-measure of an SVM using our hash features where we select the top features via BNS (the top curve as Figure 2), vs. simply reducing the width N of the bit table N and using all the features created. We see that as N decreases, the increasing number of random collisions generally harm the predictive quality of the features. Instead, it is much better to produce features in a large bit array and then select the best of these, sometimes resulting in improved performance.

The other recent work promoting hashes for classification is by Rajaram and Scholz [13]. In their case, their goal is to compute *unsupervised, similarity-preserving* hashes for texts such that 1000–4000 bits could be a sufficient feature vector for many different classification tasks. Their computations are heavyweight by comparison to ours, and first rely on a text word extractor. SpeedyFx could be used as their raw text scanning subroutine, since they only depend on the statistical properties of the features, not the actual letter sequences.

7. CONCLUSIONS & FUTURE WORK

We have shown that using SpeedyFx integer hashes in place of actual words is faster, requires less memory for transmission and use of multiple classifiers, and has an effect on classification performance that is practically noise compared to the effect of other common parameters in model selection. We showed that MASH has strong, uniform performance for words, though not appropriate for long blocks of text. Moreover, we have demonstrated the ability to classify a text by many classifiers many times faster than is possible with typical code.

The primary implication of this is that it allows significantly more efficient indexing and classification of large document repositories, e.g. to support information retrieval over all enterprise file servers with frequent file updates. It also enables lower impact to a user's computer, e.g. when there are many classification agents examining all text the user sees or types, for the purpose of 'understanding' what the user is thinking about and providing appropriate support. The savings in computation may also make it feasible to run on certain small mobile devices and/or consume less battery power.

Whenever a CPU bottleneck is resolved by more efficient algorithms, it often exposes the I/O transfer speed as the next bottleneck. Although with SpeedyFx we can easily process over a hundred megabytes per second per processor core, if there is only a single disk attached, the data transfer bottleneck may be just 2–70 MB/sec. Multiple disks or a 100 gigabit Ethernet feed from many client computers may certainly increase the input rate, but ultimately (multi-core) processing technology is getting faster faster than I/O bandwidth is getting faster.

One potential avenue for future work is to push the general-purpose text feature extraction algorithm closer to the disk hardware. That is, for each file or block read, the disk controller itself could distill the bag-of-words representation and then transfer only this small amount of data to the general-purpose processor. This could enable much higher indexing or classification scanning rates than is currently feasible.

Another potential avenue is to investigate varying the hash function to improve classification performance, e.g. to avoid a particularly unfortunate collision between an important, predictive feature and a more frequent word that masks it. At the least, this could amount to trying different random seeds and selecting the most favorable via cross-validation. In the limit, *perfect hashing* techniques could attempt to generate a hash function that produces dense, unique positive integers for each of the selected features, and negative values for features to be discarded.

8. ACKNOWLEDGMENTS

We wish to thank Kave Eshghi, Rajan Lukose, Shyam Rajaram, Martin Scholz, and Craig Soules for discussions and motivations in this area.

9. REFERENCES

[1] Aho, AV. and Corasick, MJ. 1975. Efficient string matching: an aid to bibliographic search. *Communications of the ACM* 18 (6): 333–340.

[2] Commentz-Walter, B. 1979. A string matching algorithm fast on the average. In *Proc. of the 6th Colloquium, on Automata, Languages and Programming* (July 16-20, 1979). Lecture Notes in Comp. Sci.,v.71, Springer-Verlag, 118-132.

[3] Copeland, T. 2007. *Generating Parsers with JavaCC.* Centennial Books, Alexandria, VA.

[4] Forman, G. 2003. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.* 3 (Mar. 2003), 1289-1305.

[5] Forman, G. and Rajaram, S. 2008. Scaling up text classification for large file systems. In *Proc. ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining* (Las Vegas, USA, August 24-27, 2008). KDD '08, 239–246.

[6] Ganchev, K. and Dredze, M. 2008. Small statistical models by random feature mixing. In *Workshop on Mobile Language Processing, Annual Meeting of the Association for Computational Linguistics* (June 20, 2008). ACL'08.

[7] Gospodnetic, O., and Hatcher, E. 2004. *Lucene in Action* (In Action Series). Manning Publications Co.

[8] Joachims, T. 2006. Training linear SVMs in linear time. In *Proc. ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining* (Philadelphia, USA, August 20 - 23, 2006). KDD '06, 217-226.

[9] Karp, RM and Rabin, MO. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31 (2), 249-260.

[10] Knuth, DE. 1973. *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA.

[11] McKenzie, B. J., Harries, R., and Bell, T. 1990. Selecting a hashing algorithm. *Software Practice and Experience* 20, 2 (Feb. 1990), 209-224.

[12] Mladenic, D. and Grobelnik, M. 1998. Word sequences as features in text-learning. In *Proc. 17th Electrotechnical and Computer Science Conference* (ERK98), Slovenia.

[13] Rajaram, S. and Scholz, M. 2008. Client-friendly classification over random hyperplane hashes. In *European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* (ECML/PKDD'08).

[14] Salmela, L., Tarhio, J., and Kytöjoki, J. 2007. Multipattern string matching with *q*-grams. *J. Exp. Algorithmics* 11 (Feb. 2007), 1.1.

[15] The Unicode Consortium. 2006. *The Unicode Standard, Version 5.0*. Addison-Wesley.

[16] Witten, I., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques* (Second Edition). Morgan Kaufmann, June 2005.

10. APPENDICES

Here we give the details of how we estimated the memory usage for a bag-of-words feature vector implemented with the Java JDK HashSet, and we give a full exposition of SpeedyFx, including the extension to process Unicode text encoded in UTF-8.

10.1 Bag-of-Words HashSet Memory Size

A reasonable implementation for a bag-of-words data structure in Java would be a HashMap<String,Integer>, which maps each word to the Integer count of the number of times it occurs. For the fairly common situation where one dispenses with the count and simply keeps track of the *set* of words that appear in a document, the natural representation is a HashSet<String> of words. Of course, there are a wide variety of data structures that could be used, but these are the most straightforward given the Java JDK library, and we have seen them used in practice.

In Java JDK 1.6, a HashSet is implemented as a HashMap in which the elements of the set all map to a distinguished value. So the size of the HashSet will be the size of the underlying HashMap. And since there is only a single, reused Integer object allocated in memory for the low integers, the HashMap data structure will consume roughly the same amount of memory as the HashSet version for most short documents. The remainder of the analysis focuses on the memory consumption of the HashMap itself.

A HashMap contains an array of "buckets," each representing elements whose keys have the same hash when masked to the low-order bits. The array is kept at a power of two and is doubled whenever the number of elements in the map exceeds a factor of the size of the array (by default three-fourths). Thus the size of the array will be between 4/3 and 8/3 of the number of elements in the set, which yields two buckets (references) per stored element on average.

Within each bucket is a chain of Entries, where each Entry contains a reference to the key, a reference to the value, the hash of the key (as an int), and a reference to the next Entry in the same bucket.

The key itself will be a String object which will contain three integers, and a reference to an array, which will contain an integer and some number of characters. The precise number will be corpus-dependent.

Thus, the whole size of the HashSet as a function of the number of elements is therefore six (4-byte) references, five (4-byte) integers, some number of (2-byte) characters, and the overhead of three objects. This last will vary based on implementation, but appears to be typically 12 bytes per object. So the total size of the HashSet is proportional to 80 bytes per element plus the characters themselves. In the 20 Newsgroups dataset, there are, on average 6.5 characters per unique word, for a total factor of 93 bytes per element. For *War and Peace*, there are an average of 7.5 characters per unique word, for a total factor of 95 bytes per element.

We verified this analysis by measuring the memory used by a HashSet<String> loaded with 1000 seven-letter words using the Sun Java JVM and JRE 1.6.0: 95,960 bytes. And for a HashMap<String,Integer> loaded with the same words where the counts fell in the range [0,128]: 95,944 bytes.

10.2 Inline Unicode Processing for SpeedyFx

The basic SpeedyFx algorithm is as follows:

```
int pos = start;
while (pos < end) {
  byte b = text[pos++];
  int code = codetable[b];
  if (code != 0) {
    hash = (hash >> 1) + code
  } else if (hash != 0) {
    process hash
    hash = 0;
  }
}
```

with a final step, invoked when text has been completely processed (which may involve several calls to the basic algorithm) of

```
if (hash != 0) {
  process hash
}
```

to handle the case in which the text ended without a terminating non-word character.

The "process hash" step will depend on the particular task at hand. For tokenization, it will be a callback (or faster, a "call-down" to an overridden inlined method) to a provided function; for feature extraction, it will involve setting an array element to indicate that the feature with that hash has been seen; and for in-line classification, it will involve adjusting weights on various classes based on having seen the hash.

The code table is used to map single-byte characters to codes, where a code of zero signals the end of a "word". The code table serves three functions: identifying word characters, normalizing them, and mapping each normalized character onto a full integer value. The construction of this table can be thought of as being parameterized by a function ***mapChar***(), which maps one (one-byte) character to another or to the zero character for non-word characters. The mapping function used in the paper mapped letters to their lower-case equivalents, digits to themselves, and all other characters to zero.

The creation of the code table is a two-step process:

```
int[] random = new int[256];
Random r = new Random(seed);

for (int i=0; i<256; i++) {
  random[i] = r.nextInt();
}

int[] codetable = new int[256];
for (int i=0; i<256; i++) {
  byte mapped = mapChar(i);
  if (mapped != 0) {
    codetable[i] = random[mapped];
  }
}
```

with care taken when choosing the seed so that none of the elements in the random array are zero.

In UTF-8, each character is encoded by one, two, or three bytes. One-byte characters are signified by the high-order bit being 0, two-byte characters by the high-order bits of the first byte being

110, and three-byte characters by the high-order bits of the first byte being 1110. All following bytes have the 10 for their high-order bits. To compute the 16-bit character value, the remaining unspecified bits are simply concatenated. A key feature of the UTF-8 encoding is that 7-bit ASCII characters, which necessarily have a high-order bit of 0, simply encode as themselves, so any purely ASCII text is automatically legal UTF-8-encoded Unicode.

As a first cut, the UTF-8 version of the SpeedyFx algorithm is simply

```
int pos = start;
while (pos < end) {
  byte b = text[pos++];
  char ch;
  if (b is single byte char) {
    ch = b;
  } else if (b is first byte of two-byte char) {
    byte b2 = text[pos++] & 0x3F;
    ch = ((b1 & 0x1F)<<6) | b2;
  } else if (b is first byte of three-byte char) {
    byte b2 = text[pos++] & 0x3F;
    byte b3 = text[pos++] & 0x3F;
    ch = ((b1 & 0x0F)<<12) | (b2<<6) | b3;
  } else {
    // ill-formed UTF-8.  Ignore.
    ch = 0;
  }
  int code = codetable[ch];
  if (code != 0) {
    hash = (hash >> 1) + code
  } else if (hash != 0) {
    process hash
    hash = 0;
  }
}
```

with the code table expanded to hold mappings for all 65,536 possible characters.

Unfortunately, there are several problems with implementing it this way. First, and probably most important, the code for checking whether the initial byte is the beginning of a one-byte, two-byte, or three-byte character involves a bit mask and a test against zero, which, while not exactly expensive, is more than is required and more than we would like, especially in the common case in which the vast majority of characters are one-byte characters. To handle this, we note that we have to do a table lookup anyway, so we might as well use that lookup to let us know that we have multi-byte characters. We create a second, 256-element array, the *dispatchTable*, as follows: elements 00 through 7F are copied from the main code table, elements C0 through DF are set to a special flag value *TWO_BYTE_CHAR*, elements E0 through EF are set to *THREE_BYTE_CHAR*, and all other elements are set to *ILLEGAL_CHAR*. The values of these constants are the top three values that an integer can take, and that no element of the random array has any of these values. Thus, it is efficient to test for any of the special flag values with a single comparison against an integer threshold.

The code then becomes

```
int pos = start;
while (pos < end) {
  byte b = text[pos++];
  int code = dispatchTable[b];
  if (code > THRESHOLD) {
    char ch;
    if (code == TWO_BYTE_CHAR) {
      byte b2 = text[pos++] & 0x3F;
      ch = ((b & 0x1F)<<6) | b2;
    } else if (code == THREE_BYTE_CHAR) {
      byte b2 = text[pos++] & 0x3F;
      byte b3 = text[pos++] & 0x3F;
      ch = ((b & 0x0F)<<12)|(b2<<6)| b3;
    } else {
      // ill-formed UTF-8.  Ignore.
      ch = 0;
    }
    code = codetable[ch];
  }
  if (code != 0) {
    hash = (hash >> 1) + code
  } else if (hash != 0) {
    process hash
    hash = 0;
  }
}
```

The second problem is that due to the variable-length characters, it may be the case that a given byte array might end in the middle of a two- or three-byte character. This can happen in the common case that the array represents a buffer of bytes read from a file or socket. We outline two ways to handle this.

First, if main loop is nested within a loop reading from a source into a single array, the inner loop can note that it has run out of bytes and signal that the partial character is to be moved from the end of the array to the beginning, as

```
offset = 0;
while ((nRead = src.read(text, offset,
                         len-offset)) != 0)
{
  end = nRead + offset;
  start = offset = 0;
  main loop (with changes)
  for (int i=0; i<offset; i++) {
    text[i] = text[len-offset+i];
  }
}
```

with the main loop augmented to check for overflow and set the offset if necessary, as

```
    } else if (code == THREE_BYTE_CHAR) {
      if (pos == end) {
        offset = 1;
        break;
      }
      byte b2 = text[pos++] & 0x3F;
      if (pos == end) {
        offset = 2;
        break;
      }
      byte b3 = text[pos++] & 0x3F;
      ch = ((b1 & 0x0F)<<12)|(b2<<6)| b3;
    }
```

In text with many two- and three-byte characters, these checks may be a significant performance hit. To remove them, we can note that the only time that we can run off the end of the array is if we're looking for the initial byte of a character in the last two bytes of the array. Therefore, if the main loop is performed twice, as

```
int earlyEnd = end-2;
while (pos < earlyEnd) {
  main loop without checks
}
while (pos < end) {
  main loop with checks
}
```

then the overhead of the checks (and possible break) need only occur on at most two iterations of the loop.

When SpeedyFx is to be called with arbitrary byte arrays without control over the outer loop that reads from the source stream as above, the object encapsulating the algorithm needs to be able to keep track of the fact that a prior invocation resulted in a partial character. To do that, we give the object two members, *state*, which takes an enumeration of *LIVE*, *ONE_OF_TWO*, *ONE_OF_THREE*, and *TWO_OF_THREE*, and *partial*, which contains the partial character built up so far. With such an arrangement, two modifications to the code are needed. First, in the main loop (in the second version with checks), the end-of-array tests become

```
} else if (code == THREE_BYTE_CHAR) {
  if (pos == end) {
    state = ONE_OF_THREE;
    partial = (b&0x0F) << 12;
    break;
  }
  byte b2 = text[pos++] & 0x3F;
  if (pos == end) {
    state = TWO_OF_THREE;
    partial = ((b&0x0F)<<12)|(b2<<6);
    break;
  }
  byte b3 = text[pos++] & 0x3F;
  ch = ((b1 & 0x0F)<<12)|(b2<<6)| b3;
}
```

Second, any partial characters must be finished the before the first loop:

```
if (pos == end) { return; }
if (state != LIVE) {
  char ch;
  if (state == ONE_OF_TWO) {
    byte b2 = text[pos++] & 0x3F;
    ch = partial | b2;
  } else if (state == ONE_OF_THREE) {
    byte b2 = text[pos++] & 0x3F;
    if (pos == end) {
        state = TWO_OF_THREE;
        partial |= (b2<<6);
        return;
    }
    byte b3 = text[pos++] & 0x3F;
    ch = partial |(b2<<6)| b3;
  } else if (state == TWO_OF_THREE) {
    byte b3 = text[pos++] & 0x3F;
    ch = partial |(b2<<6)| b3;
  }

  state = LIVE;
  int code = codetable[ch];

  if (code != 0) {
    hash = (hash >> 1) + code
  } else if (hash != 0) {
    process hash
    hash = 0;
  }
}
```

For runtime situations where memory is at a premium and the 64K code table poses a concern, a further modification may be worthwhile. We take advantage of the fact that most texts take their characters from relatively small contiguous subsets (typically fewer than 256 characters) of the Unicode space. (The exception would be texts in languages like Chinese and Japanese.) If we think of a 16-bit Unicode character as comprising an 8-bit *page* indicator and an 8-bit index within the page, instead of having one table of 64K elements, we instead have a 256-element array of references to lazily-constructed 256-element tables. We are now faced with two tasks: (1) determining the page number and index for the next character in a stream of bytes and (2) obtaining the appropriate code table for that page.

A two-byte UTF-8 character will specify five bits in the first byte and six in the second, for a total of eleven bits. Thus, there are eight possible pages that can be referred to by two-byte characters. These eight pages cover all of the Latin-derived scripts, Greek, Cyrillic, Arabic, Hebrew, and several others. Instead of a single value of *TWO_BYTE_CHAR*, we reserve eight special values starting at *TWO_BYTE_BASE*, each indicating a page from 0 through 7. To handle the very common case of successive two-byte characters being taken from the same page, we keep track of the most recently used two-byte table and the dispatchTable code that referred to it. Similarly, we keep track of the last three-byte page and the code that referred to it. The multi-byte lookup (ignoring end-of-array) thus becomes

```
if (code > THRESHOLD) {
  if (code == last2BCode) {
    byte b2 = text[pos++] & 0x3F;
    code = last2BTable[((b&0x03)<<6)|b2];
  } else if (code >= TWO_BYTE_BASE) {
    last2BCode = code;
    int page = code - TWO_BYTE_BASE;
    last2BTable = getTable(page);
    byte b2 = text[pos++] & 0x3F;
    code = last2BTable[((b&0x03)<<6)|b2];
  } else if (code == THREE_BYTE_CHAR) {
    byte b2 = text[pos++] & 0x3F;
    int page=((b&0x0F)<<4)|(b2&0x3C)>>2);
    if (page != last3BPage) {
      last3BPage = page;
      last3BTable = getTable(page);
    }
    byte b3 = text[pos++] & 0x3F;
    code = last3BTable[((b2&0x03)<<6)|b3;
  } else {
    // ill-formed UTF-8.  Ignore.
    code = 0;
  }
}
```

With the state-based implementation described above for handling end-of-array, we can ensure that *last2BTable* and *last3BTable* are set to the correct tables, and so only the partial index into the table needs to be preserved.

When constructing the tables, we use the same process as described above, calling a *mapChar* function to map the character and using tables of random numbers to map the result. The table of pre-computed random numbers could be a full 64K-element array, but it suffices to have two 256-elment arrays whose values are XORed, the high and low bytes being taken from the respective tables. To do this, it is important that the XOR of any pair of values from the two tables not be zero or any of the special flag codes, but this turns out to not be difficult.

10.3 Calculating Probability of Collision

The probability of getting through n b-bit hashes without a single collision can be computed as

$$P_n = P_{n-1}\frac{2^b - n}{2^b}$$

which reduces in closed form to

$$\frac{2^b!}{2^{bn}(2^b - n)!}$$

Table 9 shows the results of computing this value for various values of b, assuming a random hash distribution. The top rows of the table show the cutoff value of n before the value of P drops below various thresholds. For example, with 40 bits (5 bytes) per hash, one can add just over 1.2 million entries before it is more likely than not that there will be a collision. Unfortunately, the Java floating point types make it difficult to perform this computation for hash sizes of more than 53 bits, but from the table, which appears to go up by a factor of 16 for every byte, it is clear that with full 64-bit hashes one could expect to handle around 112 million entries at the 90% threshold or around 315 million at the 50% probability.

Table 9. Probability of No Collisions

P	Bits Per Hash			
	32	40	48	53
99%	9,290	148,663	2,378,620	13,455,509
95%	20,990	335,849	5,373,597	30,397,661
90%	30,083	481,341	7,701,473	43,566,113
80%	43,780	700,497	11,207,972	63,401,867
50%	77,162	1,234,603	19,753,661	> 100M
20%	117,578	1,881,272	30,199,389	
10%	140,636	2,250,207	36,003,334	
5%	160,414	2,566,647	41,006,375	
1%	198,890	3,182,273	50,916,406	
N				
1M		63.5%	100.0%	100.0%
2M		16.2%	99.3%	100.0%
5M		1.7%	95.7%	99.9%
10M			83.7%	94.7%
20M			49.1%	97.8%
50M			1.2%	87.0%
100M				57.9%

The bottom part of the table gives the probability of not getting a collision for various values of *n*. For an *n* of five million, for example, with 48 bits, the probability of having zero collisions is approximately 95.7%.

Of course, this whole calculation presumes that there is a reason why one would be concerned with the expectation of getting even a single collision. This was a point raised by a reviewer, and so we cover it here, but it seems to us to be a relatively unimportant point, as we discuss in section 4.1 above. While too-high a collision rate is likely to be problematic, the distribution of collisions is also important. Although it is true that in query applications false positives are more annoying than with classification applications (which are less likely to rely on a single extracted feature), one must also keep in mind the likelihood that neither of a pair of colliding words will ever be part of a query, rendering such a collision non-problematic, as well as the likelihood that the false positive will go unnoticed.