Available online at www.sciencedirect.com
SCIENCE @ DIRECT®
ELSEVIER
Journal of Discrete Algorithms 2 (2004) 137–159
JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

Adaptive text mining: inferring structure from sequences

I.H. Witten

Department of Computer Science, University of Waikato, Hamilton, New Zealand

Abstract

Text mining is about inferring structure from sequences representing natural language text, and may be defined as the process of analyzing text to extract information that is useful for particular purposes. Although hand-crafted heuristics are a common practical approach for extracting information from text, a general, and generalizable, approach requires adaptive techniques. This paper studies the way in which the adaptive techniques used in text compression can be applied to text mining. It develops several examples: extraction of hierarchical phrase structures from text, identification of keyphrases in documents, locating proper names and quantities of interest in a piece of text, text categorization, word segmentation, acronym extraction, and structure recognition. We conclude that compression forms a sound unifying principle that allows many text mining problems to be tacked adaptively.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Text mining; Phrase hierarchies; Keyphrase extraction; Generic entity extraction; Text categorization; Word segmentation; Acronym extraction; Compression algorithms; Adaptive techniques

1. Introduction

Text mining is about inferring structure from sequences representing natural language text, and may be defined as the process of analyzing text to extract information that is useful for particular purposes—often called “metadata”. Compared with the kind of data stored in databases, text is unstructured, amorphous, and contains information at many different levels. Nevertheless, the motivation for trying to extract information from it is compelling—even if success is only partial. Despite the fact that the problems are diffi-

E-mail address: ihw@cs.waikato.ac.nz (I.H. Witten).

1570-8667/$ – see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S1570-8667(03)00084-4

cult to define clearly, interest in text mining is burgeoning because it is perceived to have enormous potential practical utility.

Text compression is about identifying patterns that can be exploited to provide a more compact representation of the text. A relatively mature technology, it offers key insights for text mining. Research in compression has always taken the pragmatic view that files need to be processed whatever they may contain, rather than the normative approach of classical language analysis which generally assumes idealized input. Modern compression methods avoid making prior assumptions about the input by using adaptive techniques. In practice text—particularly text gathered from the Web, the principal source of material used today—is messy, and many useful clues come from the messiness. Adaptation is exactly what is required to deal with the vagaries of text universally encountered in the real world.

This paper studies the way in which the adaptive techniques used in text compression can be applied to text mining.

One useful kind of pattern concerns the repetition of words and phrases. So-called "dictionary" methods of compression capitalize on repetitions: they represent structure in terms of a set of substrings of the text, and achieve compression by replacing fragments of text by an index into a dictionary. A recent innovation is "hierarchical" dictionary methods, which extend the dictionary to a non-trivial hierarchical structure which is inferred from the input sequence [21]. As well as fulfilling their original purpose of forming an excellent basis for compression, such hierarchies expose interesting structure in the text that is very useful for supporting information-browsing interfaces, for example [23]. Section 2 describes schemes for generating phrase hierarchies that operate in time linear in the size of the input, and hence are practical on large volumes of text.

Keyphrases are an important kind of metadata for many documents. They are often used for topic search, or to summarize or cluster documents. It is highly desirable to automate the keyphrase extraction process, for only a small minority of documents have author-assigned keyphrases, and manual assignment of keyphrases to existing documents is very laborious. Appropriate keyphrases can be selected from the set of repeated phrases mentioned above. In order to do so we temporarily depart from our theme of text compression and, in Section 3, look at simple machine learning selection criteria and their success in keyphrase assignment.

Returning to applications of text compression, "character-based" compression methods offer an alternative to dictionary-based compression and open the door to new adaptive techniques of text mining. Character-based language models provide a promising way to recognize lexical tokens. Business and professional documents are packed with loosely structured information: phone and fax numbers, street addresses, email addresses and signatures, tables of contents, lists of references, tables, figures, captions, meeting announcements, URLs. In addition, there are countless domain-specific structures—ISBN numbers, stock symbols, chemical structures, and mathematical equations, to name a few. Tokens can be compressed using models derived from different training data, and classified according to which model supports the most economical representation. We will look at this application in Section 4.

There are other areas in which compression has been used for text mining: text categorization, segmentation into tokens, and acronym extraction. We review these in Section 5, concluding with more speculative material on structure recognition.

2. Generating phrase hierarchies

Dictionary-based compression methods capitalize on repetitions. In simplest form, they replace subsequent occurrences of a substring with references to the first instance. Standard compression methods are non-hierarchical, but hierarchical dictionary-based schemes have recently emerged that form a grammar for a text by replacing each repeated string with a production rule.

Such schemes usually operate online, making a replacement as soon as repetition is detected. "Online" algorithms process the input stream in a single pass, and begin to emit compressed output long before they have seen all the input. Historically, virtually all compression algorithms have been online, because main memory has until recently been the principal limiting factor on the large-scale application of string processing algorithms for compression. However, offline operation permits greater freedom in choosing the order of replacement. Offline algorithms can examine the input in a more considered fashion, and this raises the question of whether to seek *frequent* repetitions or *long* repetitions—or some combination of frequency and length.

This section describes three algorithms for inferring hierarchies of repetitions in sequences that have been developed recently for text compression. Surprisingly, they can all be implemented in such a way as to operate in time that is linear in the length of the input sequence. This is a severe restriction: apart from standard compression algorithms that produce non-hierarchical structure (e.g., [35]) and tail-recursive hierarchical structure (e.g., [36]), no linear-time algorithms for detecting hierarchical repetition in sequences were known until recently.

2.1. SEQUITUR*: an online technique*

Online operation severely restricts the opportunities for detecting repetitions, for there is no alternative to proceeding in a greedy left-to-right manner. It may be possible to postpone decision-making by retaining a buffer of recent history and using this to improve the quality of the rules generated, but at some point the input must be processed greedily and a commitment made to a particular decomposition—that is inherent in the nature of (single-pass) online processing.

SEQUITUR is an algorithm that creates a hierarchical dictionary for a given string in a greedy left-to-right fashion [21]. It builds a hierarchy of phrases by forming a new rule out of existing pairs of symbols, including non-terminal symbols. Rules that become non-productive—in that they do not yield a net space saving—can be deleted, and their head replaced by the symbols that comprise the right-hand side of the deleted rules. This allows rules that concatenate more than two symbols to be formed. For example, the string *abcdbcabcdbc* gives rise to the grammar

$$S \to AA$$

$$A \to aBdB$$
$$B \to bc.$$

Surprisingly, SEQUITUR operates in time that is linear in the size of the input [22]. The proof sketched here also contains an explanation of how the algorithm works. SEQUITUR operates by reading a new symbol and processing it by appending it to the top-level string and then examining the last two symbols of that string. Zero or more of the three transformations described below are applied, until none applies anywhere in the grammar. Finally, the cycle is repeated by reading in a new symbol.

At any given point in time, the algorithm has reached a particular point in the input string, and has generated a certain set of rules. Let r be one less than the number of rules, and s the sum of the number of symbols on the right-hand side of all these rules. Recall that the top-level string S, which represents the input read so far, forms one of the rules in the grammar; it begins with a null right-hand side. Initially, r and s are zero.

Here are the three transformations. Only the first two can occur when a new symbol is first processed; the third can only fire if one or more of the others has already been applied in this cycle.

1. The digram comprising the last two symbols matches an existing rule in the grammar. Substitute the head of that rule for the digram. s decreases by one; r remains the same.
2. The digram comprising the last two symbols occurs elsewhere on the right-hand side of a rule. Create a new rule for it and substitute the head for both its occurrences. r increases by one; s remains the same (it increases by two on account of the new rule, and decreases by two on account of the two substitutions).
3. A rule exists whose head occurs only once in the right-hand sides of all rules. Eliminate this rule, substituting its body for the head. r decreases by one; s decreases by one too (because the single occurrence of the rule's head disappears).

To show that this algorithm operates in linear time, we demonstrate that the total number of rules applied cannot exceed $2n$, where n is the number of input symbols. Consider the quantity $q = s - r/2$. Initially 0, it can never be negative because $r \leqslant s$. It increases by 1 for each input symbol processed, and it is easy to see that it must decrease by at least $1/2$ for each rule applied. Hence the number of rules applied is at most twice the number of input symbols.

2.2. *Most frequent first*

SEQUITUR processes the symbols in the order in which they appear. The first-occurring repetition is replaced by a rule, then the second-occurring repetition, and so on. If online operation is not required, this policy can be relaxed. This raises the question of whether there exist heuristics for selecting substrings for replacement that yield better compression performance. There are two obvious possibilities: replacing the most frequent digram first, and replacing the longest repetition first.

The idea of forming a rule for the most frequently-occurring digram, substituting the head of the rule for that digram in the input string, and continuing until some terminating condition is met, was proposed a quarter century ago by Wolff [33] and has been reinvented many times since then. The most common repeated digram is replaced first, and the process continues until no digram appears more than once. This algorithm operates offline because it must scan the entire string before making the first replacement.

Wolff's algorithm is inefficient: it takes $O(n^2)$ time because it makes multiple passes over the string, recalculating digram frequencies from scratch every time a new rule is created. However, Larsson and Moffat [16] recently devised a clever algorithm, dubbed RE-PAIR, whose time is linear in the length of the input string, which creates just this structure of rules: a hierarchy generated by giving preference to digrams on the basis of their frequency. They reduce execution time to linear by incrementally updating digram counts as substitutions are made, and using a priority queue to keep track of the most common digrams.

For an example of the frequent-first heuristic in operation, consider the string *babaabaa baa*. The most frequent digram is *ba*, which occurs four times. Creating a new rule for this yields the grammar

$$S \to AAaAaAa$$
$$A \to ba.$$

Replacing *Aa* gives

$$S \to ABBB$$
$$A \to ba$$
$$B \to Aa,$$

a grammar with eleven symbols (including three end of rule symbols). This happens to be the same as the length of the original string (without terminator).

2.3. Longest first

A second heuristic for choosing the order of replacements is to process the longest repetition first. In the same string *babaabaabaa* the longest repetition is *abaa*, which appears twice. Creating a new rule gives

$$S \to bAbaA$$
$$A \to abaa.$$

Replacing *ba* yields

$$S \to bABA$$
$$A \to aBa$$
$$B \to ba,$$

resulting in a grammar with a total of twelve symbols.

Bentley and McIlroy [2] explored the longest-first heuristic for very long repetitions, and removed them using an LZ77 pointer-style approach before invoking *gzip* to compress shorter repetitions. This is not a linear-time solution.

Suffix trees [12] provide an efficient mechanism for identifying longest repetitions. In a suffix tree, the longest repetition corresponds to the deepest internal node, measured in symbols from the root. The deepest non-terminal can be found by traversing the tree, which takes time linear in the length of the input because there is a one-to-one correspondence between leaf nodes and symbols in the string.

We are left with two problems: how to find all longest repetitions, and how to update the tree after creating a rule. Farach-Colton and Nevill-Manning (private communication) have shown that it is possible to build the tree, and update it after each replacement, in time which is linear overall. The tree can be updated in linear amortized time by making a preliminary pass through it and sorting the depths of the internal nodes. Sorting can be done in linear time using a radix sort, because no repetition will be longer than $n/2$ symbols. The algorithm relies on the fact that the deepest node is modified at each point.

2.4. Discussion

It is interesting to compare the performance of the three algorithms we have described: SEQUITUR, most frequent-first, and longest-first [24]. It is not hard to devise short strings on which any of the three outperforms the other two. In practice, however, longest-first is significantly inferior to the other techniques; indeed, simple artificial sequences can be found on which the number of rules it produces grows linearly with sequence length whereas the number of rules produced by frequent-first grows only logarithmically. Experiments on natural language text indicate that in terms of the total number of symbols in the resulting grammar, which is a crude measure of compression, frequent-first outperforms SEQUITUR, with longest-first lagging well behind.

There are many applications of hierarchical structure inference techniques in domains related more closely to text mining than compression [24]. For example, hierarchical phrase structures suggest a new way of approaching the problem of familiarizing oneself with the contents of a large collection of electronic text. Nevill-Manning et al. [23] presented the hierarchical structure inferred by SEQUITUR interactively to the user. Users can select any word from the lexicon of the collection, see which phrases it appears in, select one of them and see the larger phrases in which it appears, and so on. Larus [17] gives an application in program optimization, where the first step is to identify frequently-executed sequences of instructions—that is, paths that will yield the greatest improvement if optimized. Martin [19] has used these techniques to segment the input for speech synthesis, so that phonemes can be attached to rules at the appropriate levels.

3. Extracting keyphrases

Automatic keyphrase extraction is a promising area for text mining because keyphrases are an important means for document summarization, clustering, and topic search. Only a minority of documents have author-assigned keyphrases, and manually assigning keyphrases to existing documents is very laborious. Therefore, it is highly desirable to automate the keyphrase extraction process.

The phrase extraction techniques described above provide an excellent basis for selecting candidate keyphrases. In order to go further and decide which phrases are keyphrases, we need to step outside the area of compression and use techniques from machine learning. We have combined phrase extraction with a simple procedure based on the "naive Bayes" learning scheme, and shown it to perform comparably to the state-of-the-art in keyphrase extraction [9]. Performance can be boosted even further by automatically tailoring the extraction process to the particular document collection at hand, and experiments with a large collection of technical reports in computer science have shown that the quality of the extracted keyphrases improves significantly if domain-specific information is exploited.

3.1. Background

Several solutions have been proposed for generating or extracting summary information from texts [3,14,15]. In the specific domain of keyphrases, there are two fundamentally different approaches: keyphrase *assignment* and keyphrase *extraction*. Both use machine learning methods, and require for training purposes a set of documents with keyphrases already identified. In keyphrase assignment, there is a predefined set from which all keyphrases are chosen—a controlled vocabulary. Then the training data provides, for each keyphrase, a set of documents that are associated with it. For each keyphrase, a classifier is created from all training documents using the ones associated with it as positive examples and the remainder as negative examples. A new document is processed by each classifier, and is assigned the keyphrases associated with those that classify it positively [7]. Here, the only keyphrases that can be assigned are ones that are in the controlled vocabulary. In contrast, keyphrase extraction, which forms the basis of the method described here, employs linguistic and information retrieval techniques to extract phrases from a new document that are likely to characterize it. The training set is used to tune the parameters of the extraction algorithm, and any phrase in the new document is a potential keyphrase.

Turney [29] describes a system for keyphrase extraction, GenEx, that is based on a set of parametrized heuristic rules which are fine-tuned using a genetic algorithm. The genetic algorithm optimizes the number of correctly identified keyphrases in the training documents by adjusting the rules' parameters. Turney compares GenEx to the straightforward application of a standard machine learning technique—bagged decision trees [4]—and concludes that GenEx performs better. He also shows that it generalizes well across collections: trained on a collection of journal articles it successfully extracts keyphrases from a collection of web pages on a different topic. This is an important feature because training GenEx on a new collection is computationally very expensive.

3.2. Keyphrase extraction

Keyphrase extraction is a classification task. Each phrase in a document is either a keyphrase or not, and the problem is to correctly classify phrases into one of these two categories. Machine learning provides off-the-shelf tools for this problem. In the terminology of machine learning, the phrases in a document are "examples" and the learning problem is to find a mapping from the examples to the classes "keyphrase" and "not-keyphrase". Learning techniques can automatically generate this mapping if they are provided with a set of training examples—that is, examples that have class labels assigned to them. In the context of keyphrase extraction this is simply a set of phrases which have been identified as either being keyphrases or not. Once the learning scheme has generated the mapping given the training data, it can be applied to unlabeled data, thereby extracting keyphrases from new documents.

Not all phrases in a document are equally likely to be keyphrases *a priori*. In order to facilitate the learning process, most phrases can be eliminated from the examples that are presented to the learning scheme. We have experimented with many ways of doing this, most involving one of the hierarchical phrase extraction algorithms described above. Following this process, all words are case-folded, and stemmed using the iterated Lovins method. This involves using the classic Lovins stemmer [18] to discard any suffix, and repeating the process on the stem that remains, iterating until there is no further change. The final step in preparing the phrases for the learning scheme is to remove all stemmed phrases that occur only once in the document.

Once candidate phrases have been generated from the text, it is necessary to derive selected properties from them. In machine learning these properties are called the "attributes" of an example. Several potential attributes immediately spring to mind: the number of words in a phrase, the number of characters, the position of the phrase in the document, etc. However, in our experiments, only two attributes turned out to be useful in discriminating between keyphrases and non-keyphrases. The first is the distance into the document of the phrase's first appearance. The second, and more influential, is the "term frequency times inverse document frequency", or TF $\times$ IDF, score of a phrase [32]. This is a standard measure used in information retrieval which favors terms that occur frequently in the document ("term frequency") but disfavors ones that occur in many different documents ("inverse document frequency") on the grounds that common terms are poor discriminators.

Both these attributes are real numbers. We use the "naive Bayes" learning method because it is simple, quick, and effective: it conditions class probabilities on each attribute, and assumes that the attributes are statistically independent. In order to make it possible to compute conditional probabilities, we discretize the attributes prior to applying the learning scheme, quantizing the numeric attributes into ranges so that each value of the resulting new attribute represents a range of values of the original numeric attribute. Fayyad and Irani's [8] discretization scheme, which is based on the Minimum Description Length principle, is suitable for this purpose.

The naive Bayes learning scheme is a simple application of Bayes' formula. It assumes that the attributes—in this case TF $\times$ IDF and distance—are independent given the class. Making this assumption, the probability that a phrase is a keyphrase given that it has discretized TF $\times$ IDF value T and discretized distance D is easily computed from the probability that a keyphrase has TF $\times$ IDF score T, the probability that a keyphrase has distance D, the *a priori* probability that a phrase is a keyphrase, and a suitable normalization factor. All these probabilities can be estimated by counting the number of times the corresponding event occurs in the training data.

This procedure is used to generate a Bayes model from a set of training documents for which keyphrases are known (for example, because the author provided them). The resulting model can then be applied in a straightforward way to a new document from which keyphrases are to be extracted.

First, TF $\times$ IDF scores and distance values are calculated for all phrases in the new document using the procedure described above, using the discretization obtained from the training documents. (Both attributes, TF $\times$ IDF and distance, can be computed without knowing whether a phrase is a keyphrase or not.) The naive Bayes model is then applied to each phrase, giving the estimated probability of this phrase being a keyphrase. The result is a list of phrases ranked according to their associated probabilities. Finally, the r highest ranked phrases are output, where r is a user-determined parameter.

3.3. Experimental results

We have evaluated this keyphrase extraction method on several different document collections with author-assigned keyphrases. The criterion for success is the extent to which the algorithm produces the same stemmed phrases as authors do. This method of evaluation is the same as used by Turney [29], and on comparing our results with GenEx we conclude that both methods perform at about the same level.

An interesting question is how keyphrase extraction performance scales with the amount of training data available. There are two ways in which the quantity of available documents can influence performance on fresh data. First, training documents are used in computing the discretization of the attributes TF $\times$ IDF and distance, and the corresponding parameters for the naive Bayes model. It is essential that these documents have keyphrases assigned to them because the learning method needs labeled examples. Second, training documents are used to calculate the document frequency of each phrase, which in turn is used to derive its TF $\times$ IDF score. In this case, unlabeled documents are appropriate because the phrase labels are not used.

To investigate these effects we performed experiments with a large collection of computer science technical reports (CSTR) from the New Zealand Digital Library. The results show that keyphrase extraction performance is close to optimum if about 50 training documents are used for both generating the classifier and computing the global frequencies. In other words, 50 labeled documents are sufficient to push performance to its limit. However, we will see in the next subsection that if domain-specific information is exploited in the learning and extracting process, much larger volumes of labeled training documents prove beneficial.

3.4. Exploiting domain-specific information

A simple modification of the above procedure enables it to exploit collection-specific knowledge about the likelihood of a particular phrase being a keyphrase. To do this, just keep track of the number of times a candidate phrase occurs as a keyphrase in the training documents and use this information in the form of an additional, third, attribute in the learning and extraction process.

The new attribute only makes sense if the documents for which keyphrases are to be extracted are from the same domain as the training documents. Otherwise, biasing the extraction algorithm towards phrases that have occurred as author-assigned keyphrases during training cannot possibly have any beneficial effect. In order to make use of the information provided by the new attribute, it is therefore necessary to re-train the extraction algorithm if keyphrases are to be extracted from documents on a different topic. Training time becomes a critical factor.

We have empirically verified that exploiting domain-specific information increases the number of correctly extracted keyphrases by performing experiments with the CSTR collection mentioned above [9]. In order to isolate the effect of changing the number of documents for computing the keyphrase-frequency attribute, we used a separate set of documents—the keyphrase-frequency corpus—for counting the number of times a phrase occurs as a keyphrase. We found that the use of the keyphrase-frequency attribute improved keyphrase extraction markedly when the size of the keyphrase-frequency corpus increased from zero (i.e., no keyphrase-frequency attribute) to 100, and improved markedly again when increased from 100 to 1000. The actual set of 130 training documents was held constant; also, the same set of 500 test documents was used throughout this experiment.

3.5. Discussion

We conclude that a simple algorithm for keyphrase extraction, which filters phrases extracted using a hierarchical decomposition scheme such as those described in Section 2 based on the naive Bayes machine learning method, performs comparably to the state of the art. Furthermore, performance can be boosted by exploiting domain-specific information about the likelihood of keyphrases. The new algorithm is particularly well suited for making use of this information because it can be trained up very quickly in a new domain. Experiments on a large collection of computer science technical reports confirm that this modification significantly improves the quality of the keyphrases extracted.

4. Generic entity extraction

We now return to our main theme: using the adaptive techniques developed in text compression for the purposes of text mining. In this section and the next, we will review applications of character-based compression methods. Throughout this work, the well-known PPM text compression scheme is used [1,6], with order 5 (except where otherwise mentioned) and escape method D [13]. However, the methods and results are not particularly sensitive to the compression scheme used, although character-based prediction is assumed.

“Named entities” are defined as proper names and quantities of interest in a piece of text, including personal, organization, and location names, as well as dates, times, percentages, and monetary amounts [5]. The standard approach to extracting them from text is manual: tokenizers and grammars are hand-crafted for the particular data being extracted. Commercial text mining software includes IBM’s *Intelligent Miner for Text* [28], which uses specific recognition modules carefully programmed for the different data types, Apple’s *Data Detectors* [20], which uses language grammars, and the *Text Tokenization Tool* of [11].

An alternative approach to generic entity extraction is to use compression-based training instead of explicit programming to detect instances of sublanguages in running text [31].

4.1. An example

In order to assess the power of language models to discriminate tokens, experiments were conducted with information items extracted (manually) from twenty issues of a 4-page, 1500-word, weekly electronic newsletter. Items of the kind that readers might wish to take action on were classified into generic types: people's names; dates and time periods; locations; sources, journals, and book series; organizations; URLs; email addresses; phone numbers; fax numbers; and sums of money. These types are subjective: dates and time periods are lumped together, whereas for some purposes they should be distinguished; personal and organizational names are separated, whereas for some purposes they should be amalgamated. The methodology we describe accommodates all these options: there is no commitment to any particular ontology.

4.2. Discriminating isolated tokens

The first experiment involved the ability to discriminate between different token types when the tokens are taken in isolation. Lists of names, dates, locations, etc. in twenty issues of the newsletter were input to the PPM compression scheme separately, to form ten compression models. Each issue contained about 150 tokens, unevenly distributed over token types. In addition, a plain text model was formed from the full text of all these issues. These models were used to identify each of the tokens in a newsletter that did not form part of the training data, on the basis of which model compresses them the most. Although the plain text model could in principle be assigned to a token because it compresses it better than all the specialized models, in fact this never occurred.

Of the 192 tokens in the test data, 40% appeared in the training data (with the same label) and the remainder were new. 90.6% of the total were identified correctly and the remaining 9.4% incorrectly; all errors were on new symbols. Three of the "old" symbols contain line breaks that do not appear in the training data: for example, in the test data `Parallel␣Computing\nJournal` is split across two lines as indicated. However, these items were nevertheless identified correctly. The individual errors are easily explained; some do not seem like errors at all. For example, the place names `Norman` and `Berkeley` were "mis"-identified as people's names, time periods like `Spring␣2000` were mis-identified as sources (because of confusion with newsgroups like `comp.software.year-2000`), people's names were confused with organizational names, and so on.

4.3. Distinguishing tokens in context

When tokens appear in text, contextual information provides additional cues for disambiguating them. Identification must be done conservatively, so that strings of plain text are not misinterpreted as tokens—since there are many strings of plain text, there are countless opportunities for error.

Context often helps recognition: e.g., email addresses in this particular newsletter are always flanked by angle brackets. Conversely, identification may be foiled by misleading context: e.g., some names are preceded by `Rep.`, which reduces the weight of the capitalization evidence for the following word because capitalization routinely follows a period.

The second experiment evaluated the effect of context by assuming that all tokens have been *located* in the test issue, and the task is to *identify* their types *in situ*. If a stretch of text is identified as a token of the appropriate type it will compress better using the specialized model; however, begin- and end-token markers must be coded to indicate this fact. To investigate this, all tokens in the data were replaced by a surrogate symbol that was treated by PPM as a single character (different from all the ASCII characters). A different surrogate was used for each token type. A new model was generated from the modified training data, and the test article was compressed by this model to give a baseline entropy of e_0 bits. Then each token in turn, taken individually, was restored into the test article as plain text and the result recompressed to give entropy e bits. This will (likely) be greater than e_0 because the information required to represent the token itself (almost certainly) exceeds that required to represent its type. Suppose e_m is the token's entropy with respect to model m. Then the net space saved by recognizing this token as belonging to model m is $e - (e_0 + e_m)$ bits. This quantity was evaluated for each model to determine which one classified the token best, or whether it was best left as plain text. The procedure was repeated for each token.

When context is taken into account the error rate per token actually increases from 9.4% to 13.5%. However, almost all these "errors" are caused by failure to recognize a token as different from plain text, and the rate of actual mis-recognitions is only 1%—or just two mis-recognitions, one of which is the above-mentioned `Berkeley` being identified as a name.

To mark up a string as a token requires the insertion of two extra symbols: begin- and end-token, and it is this additional overhead that causes the above-noted failures to recognize tokens. However, the tradeoff between actual errors and failures to identify can be adjusted by using a non-zero threshold when comparing the compression for a particular token with the compression when its characters are interpreted as plain text. This allows a small increase in the number of errors to be sacrificed for a larger decrease in identification failures.

4.4. Locating tokens in context

Tokens can be located by considering the input as an interleaved sequence of information from different sources. Every token is to be bracketed by *begin-token* and *end-token* markers; the problem is to "correct" text by inserting such markers appropriately. The markers also identify the type of token in question—thus we have *begin-name-token*, *end-name-token*, etc., written as `<n>`, `</n>`. Whenever *begin-token* is encountered, the encoder switches to the compression model appropriate to that token type, initialized to a null prior context. Whenever *end-token* is encountered, the encoder reverts to the plain text model that was in effect before, replacing the token by a single symbol representing that token type.

The algorithm takes a string of text and works out the optimal sequence of models that would produce it, along with their placement. It works Viterbi-style [30], processing the input characters to build a tree in which each path from root to leaf represents a string of characters that is a possible interpretation of the input. The paths are alternative output strings, and *begin-token* and *end-token* symbols appear on them. The entropy of a path can be calculated by starting at the root and coding each symbol along the path according to the model that is in force when that symbol is reached. The context is re-initialized to a unique starting token whenever *begin-token* is encountered, and the appropriate model is entered. On encountering *end-token*, it is encoded and the context reverts to what it was before.

What causes the tree to branch is the insertion of *begin-token* symbols for every possible token type, and the *end-token* symbol—which must be for the currently active token type so that nesting is properly respected. To expand the tree, a list of open leaves is maintained, each recording the point in the input string that has been reached and the entropy value up to that point. The lowest-entropy leaf is chosen for expansion at each stage. Unless the tree and the list of open leaves are pruned, they grow very large very quickly. A beam search is used, and pruning operations are applied that remove leaves from the list and therefore prevent the corresponding paths from growing further.

To evaluate the procedure for locating tokens in context, we used the training data from the same issues of the newsletter as before, and the same single issue for testing. The errors and mis-recognitions noted above when identifying tokens in context (rates of 1% and 12.5%, respectively) also occur when locating tokens. Inevitably there were a few incorrect positive identifications—2.6% of the number of tokens—where a segment of plain text was erroneously declared to be a token. In addition, 8% of tokens suffered from incorrect boundary placement, where the algorithm reported a token at approximately the same place as in the original, but the boundaries were slightly perturbed. Finally, a further 4.7% of tokens suffered discrepancies which were actually errors made inadvertently by the person who marked up the test data.

4.5. Discussion

We find these initial results encouraging. There are several ways that they could be improved. The amount of training data—about 3000 tokens, distributed among ten token types—is rather small. The data certainly contains markup errors, probably at about the same rate—4.7% of tokens—as the test file. Many of the mistakes were amongst very similar categories: for example, fax numbers contained embedded phone numbers and were only distinguished by the occurrence of the word *fax*; several times they were confused with phone numbers and this counted as an error. Some of the mistakes were perfectly natural—`Norman` as a name instead of a place, for example. In addition, improvements could likely be made to the pruning algorithm.

5. Other text mining tasks

Character-based compression can be applied in many other ways to text mining tasks. Here are some examples.

5.1. *Text categorization*

A central feature of the approach to generic entity extraction described in the previous section is the basic assumption that a token can be identified by compressing it according to different models and seeing which produces the fewest bits of output. We now examine whether this extends to text categorization—the assignment of natural language texts to predefined categories based on their content. Already-classified documents, which define the categories, are used to build a model that can be used to classify new articles.

Text categorization is a hot topic in machine learning. Typical approaches extract "features", generally words, from text, and use the feature vectors as input to a machine learning scheme that learns how to classify documents. This "bag of words" model neglects word order and contextual effects. It also raises some problems: how to define a "word", what to do with numbers and other non-alphabetic strings, and whether to apply stemming. Because there are so many different features, a selection process is applied to determine the most important words, and the remainder are discarded.

Compression seems to offer a promising alternative approach to categorization, with several potential advantages:

- it yields an overall judgement on the document as a whole, and does not discard information by pre-selecting features;
- it avoids the messy problem of defining word boundaries;
- it deals uniformly with morphological variants of words;
- depending on the model (and its order), it can take account of phrasal effects that span word boundaries;
- it offers a uniform way of dealing with different types of documents—for example, files in a computer system;
- it minimizes arbitrary decisions that inevitably need to be taken to render any learning scheme practical.

We have performed extensive experiments on the use of PPM for categorization using a standard dataset [10]. Best results were obtained with order 2; other values degraded performance in almost all cases—presumably because the amount of training data available is insufficient to justify more complex models.

5.1.1. *The benchmark data*

All our results are based on the Reuters-21578 collection of newswire stories, the standard testbed for the evaluation of text categorization schemes. In total there are 12,902 stories averaging 200 words each, classified into 118 categories. Many stories are assigned to multiple categories, and some are not assigned to any category at all. The distribution among categories is highly skewed: the ten largest—*earnings*, *corporate acquisitions*, *money market*, *grain*, *crude oil*, *trade issues*, *interest*, *shipping*, *wheat*, and *corn*—contain 75% of stories, an average of around 1000 stories each.

5.1.2. Pairwise discrimination

Applying a straightforward compression methodology to the problem of text categorization quickly yields encouraging results. In the two-class case, to distinguish documents of class A from documents of class B we form separate models M_A and M_B from the training documents of each class. Then, given a test document (different from the training documents), we compress it according to each model and calculate the gain in per-symbol compression obtained by using M_A instead of M_B. We assign the document to class A or B depending on whether this difference is positive or negative, on the principle that M_A will compress documents of class A better, and similarly for M_B. Encouraging results are obtained.

5.1.3. Building positive and negative models

To extend to multiply-classified articles, we decide whether a model belongs to a particular category independently of whether it belongs to any other category. We build positive and negative models for each category, the first from all articles that belong to the category and the second from those that do not.

5.1.4. Setting the threshold

Deciding whether a new article should in fact be assigned to category C or not presents a tradeoff between making the decision liberally, increasing the chance that an article is correctly identified but also increasing the number of "false positives"; or conservatively, reducing the number of false positives at the expense of increased "false negatives". This tradeoff is captured by the standard information retrieval notions of *precision*, that is the number of articles that the algorithm *correctly* assigns to category C expressed as a proportion of the documents that it assigns to this category, and *recall*, that is the number of articles that the algorithm correctly assigns to category C expressed as a proportion of the articles that actually have this category. To allow comparison of our results with others, we maximize the average of recall and precision—a figure that is called the "breakeven point".

The basic strategy is to calculate the predicted probability $Pr[C|A]$ of article A having classification C, compare it to a predetermined threshold, and declare the article to have classification C if it exceeds the threshold. We choose the threshold individually for each class, to maximize the average of recall and precision for that class. To do this the training data is further divided into a new training set and a "validation set", in the ratio 2:1. The threshold t is chosen to maximize the average of recall and precision for the category (the breakeven point) on the validation set. Then maximum utility is made of the information available by rebuilding the positive and negative models from the full training data.

As an additional benefit, threshold selection automatically compensates for the fact that the positive and negative models are based on different amounts of training data. In general, one expects to achieve better compression with more data.

5.1.5. Results

Elsewhere [10] we have compared this method with results reported for the naive Bayes and Linear Support Vector Machine methods [7]. The compression-based method outperforms naive Bayes on the six largest categories (*grain* is the only exception) and is worse on the four smallest ones. It is almost uniformly inferior to the support vector method, *money market* being the only exception.

Compared to LSVM, compression-based categorization produces particularly bad results on the categories *wheat* and *corn*, which are (almost) proper subsets of the category *grain*. Articles in *grain* summarize the result of harvesting grain products—for example, by listing the tonnage obtained for each crop—and all use very similar terminology. Consequently the model for *wheat* is very likely to assign a high score to *every* article in *grain*.

The occurrence of the word "wheat" is the only notable difference between an article in *grain* that belongs to *wheat* and one that does not. The presence of a single word is unlikely to have a significant effect on overall compression, and this is why the new method performs poorly on these categories. Support vector machines perform internal feature selection, and can focus on a single word if that is the only discriminating feature of a category. In comparison, naive Bayes performs badly on the same categories as the new method, because it too has no mechanism for internal feature selection.

5.1.6. Modifications

Our initial results were obtained quickly, and we found them encouraging. We then made many attempts to improve them, all of which met with failure.

To force compression models that are more likely to discriminate successfully between similar categories, we experimented with a more costly approach. Instead of building one positive and one negative model, we built one positive and 117 negative models for each of the 118 categories. Each negative model only used articles belonging to the corresponding category that did not occur in the set of positive articles. During classification, an article was assigned to a category if the positive model compressed it more than all negative models did. Results were improved slightly for categories like *wheat* and *corn*. However, the support vector method still performed far better. Moreover, compared to the standard compression-based method, performance deteriorated on some other categories.

We also experimented with several modifications to the standard procedure, none of which produced any significant improvement over the results reported above:

- not rebuilding the models from the full training data;
- using the same number of stories for building the positive and negative models (usually far more stories are available for the negative one);
- priming the models with fresh Reuters data from outside the training and test sets;
- priming the models with the full training data (positive and negative articles);
- artificially increasing the counts for the priming data over those for the training data and *vice versa*;
- using only a quarter of the original training data for validation;
- using a word model of order 0, escaping to a character model of order 2 for unseen words.

5.1.7. Discussion

Compared to state-of-the-art machine learning techniques for categorizing English text, the compression-based method produces inferior results because it is insensitive to subtle differences between articles that belong to a category and those that do not. We do not believe our results are specific to the PPM compression scheme. If the occurrence of a single word is what counts, any compression scheme will likely fail to classify the article correctly. Machine learning schemes fare better because they automatically eliminate irrelevant features.

Compared to word-based approaches, compression-based methods avoid *ad hoc* decisions when preparing input text for the actual learning task. Moreover, these methods transcend the restriction to alphabetic text and apply to arbitrary files. However, feature selection seems to be essential for some text categorization tasks, and this is not incorporated in compression methods.

5.2. Segmentation into tokens

Conventional text categorization is just one example of many text mining methods that presuppose that the input is somehow divided into lexical tokens. Although "words" delimited by non-alphanumeric characters provide a natural tokenization for many items in ordinary text, this assumption fails in particular cases. For example, generic tokenization would not allow many date structures (e.g., *30Jul98*, which is used throughout the newsletters of Section 4) to be parsed. In general, any prior segmentation into tokens runs the risk of obscuring information.

A simple special case of this scheme for compression-based entity extraction can be used to divide text into words, based on training data that has been segmented by hand. An excellent testbed for this research is the problem of segmenting Chinese text, which is written without using spaces or other word delimiters. Although Chinese readers are accustomed to inferring the corresponding sequence of words as they read, there is considerable ambiguity in the placement of boundaries which must be resolved in the process. Interpreting a text as a sequence of words is necessary for many information retrieval and storage tasks: for example, full-text search and word-based compression.

Inserting spaces into text can be viewed as a hidden Markov modeling problem. Between every pair of characters lies a potential space. Segmentation can be achieved by training a character-based compression model on pre-segmented text, and using a Viterbi-style algorithm to interpolate spaces in a way that maximizes the overall probability of the text.

For non-Chinese readers, we illustrate the success of the space-insertion method by showing its application to English text in Table 1, which is due to Teahan [26]. At the top is the original text, including spaces in the proper places, then the input to the segmentation procedure, and finally the output of the PPM-based segmentation method.

In this experiment PPM was trained on a sample of English, and its recall and precision for space insertion were both 99.52%. Corresponding figures for a word-based method that does not use compression-based techniques [25] were 93.56% and 90.03%, respectively, a result which is particularly striking because PPM had been trained on only a small fraction of the amount of text used for the other scheme.

PPM performs well on unknown words: although *Micronite* does not occur in the Brown Corpus, it is correctly segmented in Table 1. There are two errors. First, a space was not inserted into *LoewsCorp* because the single "word" requires only 54.3 bits to encode,

Table 1
Segmenting words in English text

text	the unit of New York-based Loews Corp that makes Kent cigarettes stopped using crocidolite in its Micronite cigarette filters in 1956.
input	theunitofNewYork-basedLoewsCorpthatmakesKentcigarettess toppedusingcrocidoliteinitsMicronitecigarettefiltersin1956.
output	the unit of New York-based LoewsCorp that makes Kent cigarettes stopped using croc idolite in its Micronite cigarette filters in 1956.

whereas *Loews Corp* requires 55.0 bits. Second, an extra space was added to *crocidolite* because that reduced the number of bits required from 58.7 to 55.3.

Existing techniques for Chinese text segmentation are either word-based, or rely on hand-crafted segmentation rules. In contrast, the compression-based methodology is based on character-level models formed adaptively from training text. Such models do not rely on a dictionary and fall back on general properties of language statistics to process novel words. Excellent results have been obtained with the new scheme [27].

5.3. *Acronym extraction*

Identifying acronyms in documents—which is certainly also about looking for patterns in text—presents a rather different kind of problem. Webster defines an “acronym” as

> a word formed from the first (or first few) letters of a series of words, as *radar*, from *ra*dio *d*etecting *a*nd *r*anging.

Acronyms are often defined by preceding or following their first use with a textual explanation—as in Webster’s example. Finding all acronyms, along with their definitions, in a particular technical document is a problem that has previously been tackled using *ad hoc* heuristics. The information desired—acronyms and their definitions—is relational, and this distinguishes it from the text mining problems discussed above.

It is not immediately obvious how compression can assist in locating relational information such as this. Language statistics certainly differ between acronyms and running text, because the former have a higher density of capital letters and a far higher density of non-initial capital letters. However, it seems unlikely that acronym definitions will be recognized reliably on this basis: they will not be readily distinguished from ordinary language by their letter statistics.

We have experimented with coding potential acronyms with respect to the initial letters of neighboring words, and using the compression achieved to signal the occurrence of an acronym and its definition [34]. Our criterion is whether a candidate acronym could be coded more efficiently using a special model than it is using a regular text compression scheme. A phrase is declared to be an acronym definition if the discrepancy between the number of bits required to code it using a general-purpose compressor and the acronym model exceeds a certain threshold.

We first pre-filter the data by identifying acronym candidates: for initial work we decided to consider words in upper case only. Then we determined two windows for each candidate, one containing 16 preceding words and the other 16 following words. This range covered all acronym definitions in our test data.

5.3.1. Compressing the acronyms

Candidate acronyms are coded using a group of models that express the acronym in terms of the leading letters of the words on either side. This group comprises four separate models. The first tells whether the acronym precedes or follows its definition. The second gives the distance from the acronym to the first word of the definition. The third identifies a sequence of words in the text by a set of offsets from the previous word. The fourth gives the number of letters to be taken from each word. Each of these models is an order-0 PPM model with a standard escape mechanism.

After compressing the acronym candidates with respect to their context, all legal encodings for each acronym are compared and the one that compresses best is selected. For comparison, we compress the acronym using the text model, taking the preceding context into account. The candidate is declared to be an acronym if

$$\frac{\text{bits}_{\text{acronym model}}}{\text{bits}_{\text{text model}}} \leqslant t$$

for some predetermined threshold t. Although subtracting the number of bits seems more easily justified than using the ratio between them, in fact far better results were obtained using the ratio method. We believe that the reason for this is linked to the curious fact that, using a standard text model, longer acronyms tend to compress into fewer bits than do shorter ones. While short acronyms are often spelt out, long ones tend to be pronounced as words. This affects the choice of letters: longer acronyms more closely resemble “natural” words.

5.3.2. Experimental results

To test these ideas, we conducted an experiment on a sizable sample of technical reports, and calculated recall and precision for acronym identification. The operating point on the recall/precision curve can be adjusted by varying t. While direct comparison with other acronym-extraction methods is not possible because of the use of different text corpora, our scheme performs well and provides a viable basis for extracting acronyms and their definitions from plain text. Compared to other methods, it reduces the need to come up with heuristics for deciding when to accept a candidate acronym—although some prior choices are made when deciding how to code acronyms with respect to their definitions.

5.4. Structure recognition

We have shown that while compression is a useful tool for many token classification tasks, it is less impressive for document categorization. As a discriminant, overall compression tends to weaken as the size of individual items grows, because a single holistic measure may become less appropriate. Some decisions depend on the occurrence or non-occurrence of a few special words, which makes feature selection essential. Even in token discrimination, different kinds of token may be distinguishable only by the context in which they occur—for example, author's names and editor's names no doubt enjoy identical statistical properties, but are distinguished in bibliographic references by local context.

The size of individual tokens can often be reduced by extending the techniques described above to work hierarchically. This allows more subtle interactions to be captured. Names are decomposable into forenames, initial, surname; email addresses into username, domain, and top-level domain; and fax numbers contain embedded phone numbers. After analyzing the errors made during the generic entity extraction experiments of Section 4, we refined the markup of the training documents to use these decompositions. For instance:

Name	`<n><f>Ian</f>␣<i>H</i>.␣<s>Witten</s></n>`
Email	`<e><u>ihw</u>@<d>cs.waikato.ac</d>.<t>nz</t></e>`
Fax	`<f><p>+64-7-856-2889</p>␣fax</f>`

We use the term "soft parsing" to denote inference of what is effectively a grammar from example strings, using exactly the same compression methodology as before. During training, models are built for each component of a structured item, as well as the item itself. For example, the *forename* model is trained on all forenames that appear in the training data, while the *name* model is trained on patterns like *forename* followed by space followed by *middle initial* followed by period and space followed by *surname*, where each of the lower-level items—*forename, middle initial* and *surname*—are treated by PPM as a single "character" that identifies the kind of token that occurs. When the test file is processed to locate tokens in context, these new tags are inserted into it too. The algorithm described in Section 4 accommodates nested tokens without any modification at all.

Initial results are mixed. Some errors are corrected (e.g., some names that had been confused with other token types are now correctly marked), but other problems remain (e.g., the fax/phone number mix-up) and a few new ones emerge. Some are caused by the pruning strategies used; others are due to insufficient training data. Despite inconclusive initial results, we believe that soft parsing will prove invaluable in situations with strong hierarchical context (e.g., references and tables).

It is possible that the technique can be extended to the other kinds of tasks considered above. For example, we could mark up an acronym, with its definition. Webster's *radar* example above might look like

Acronym ... `of␣a␣series␣of␣words,␣as␣<a>radar,␣from`
`<d>radio␣detecting␣and␣ranging</d></a>.`

To capture the essential feature of acronyms—that the word being defined is built from characters in the definition—the search algorithm needs to be extended to consider this possibility.

In text categorization, important features could be highlighted. The word "wheat", which distinguishes articles on *wheat* from other articles in the *grain* category, could be marked in the training data—or by an automatic feature selection process—and the markup inferred in the test data. Such techniques may allow compression-based generalization to tackle problems that require feature selection.

6. Conclusions

Text mining is a burgeoning new area that is likely to become increasingly important in future. This paper has argued, through examples, that compression forms a sound unifying principle that allows many text mining problems to be tacked adaptively.

Word-based and character-based compression methods can be applied to different kinds of text mining tasks. Phrase hierarchies can be extracted from documents using recently-developed algorithms for inferring hierarchies of repetitions in sequences, all of which have been proposed for text compression. Although we have not focused on applications of phrase hierarchies in this paper, they are beginning to be applied in a diverse range of areas, including browsing in digital libraries.

The extraction of different kinds of entities from text is commonly approached through the use of hand-tailored heuristics. However, adaptive methods offer significant advantages in construction, debugging, and maintenance. While they suffer from the necessity to mark up large numbers of training documents, this can be alleviated by priming the compression models with appropriate data—lists of names, addresses, etc.—gathered from external sources.

Other kinds of text-mining problems also succumb to compression-based techniques: examples are word segmentation and acronym extraction. Some, notably text categorization, seem less well-suited to the holistic approach that compression offers. However, hierarchical decomposition can be used to strengthen context, and perhaps even to incorporate the results of automatic feature selection.

Adaptive text mining using compression-based techniques is in its infancy. Watch it grow.

Acknowledgements

Much of the research reported in this paper here has been undertaken in conjunction with others. In particular, Zane Bray, John Cleary, Eibe Frank, Stuart Inglis, Malika Mahoui, Gordon Paynter, Craig Nevill-Manning, Bill Teahan, Ying Ying Wen and Stuart Yeates have all contributed greatly to the work described here.

References

[1] T.C. Bell, J.G. Cleary, I.H. Witten, Text Compression, Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] J. Bentley, D. McIlroy, Data compression using long common strings, in: Proc. Data Compression Conference, IEEE Press, Los Alamitos, CA, 1999, pp. 287–295.

[3] R. Brandow, K. Mitze, L.F. Rau, Automatic condensation of electronic publications by sentence selection, Information Processing and Management 31 (5) (1995) 675–685.
[4] L. Breiman, Bagging predictors, Machine Learning 24 (2) (1996) 123–140.
[5] N.A. Chinchor, Overview of MUC-7/MET-2, in: Proc. Message Understanding Conference MUC-7, 1999.
[6] J.G. Cleary, I.H. Witten, Data compression using adaptive coding and partial string matching, IEEE Trans. Comm. 32 (4) (1984) 396–402.
[7] S.T. Dumais, J. Platt, D. Heckerman, M. Sahami, Inductive learning algorithms and representations for text categorization, in: Proceedings of the 7th International Conference on Information and Knowledge Management, 1998.
[8] U.M. Fayyad, K.B. Irani, Multi-interval discretization of continuousvalued attributes for classification learning, in: Proc. Internat. Joint Conference on Artifical Intelligence, 1993, pp. 1022–1027.
[9] E. Frank, G.W. Paynter, I.H. Witten, C. Gutwin, C. Nevill-Manning, Domain-specific keyphrase extraction, in: Proc. Internat. Joint Conference on Artificial Intelligence, Stockholm, Sweden, 1999, pp. 668–673.
[10] E. Frank, C. Chiu, I.H. Witten, Text categorization using compression models, in: Proc. Data Compression Conference (Poster paper), IEEE Press, Los Alamitos, CA, 2000. Full version available as Working Paper 00/2, Department of Computer Science, University of Waikato.
[11] C. Grover, C. Matheson, A. Mikheev, TTT: Text Tokenization Tool, 1999, http://www.ltg.ed.ac.uk/.
[12] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, Cambridge, UK, 1997.
[13] P.G. Howard, The design and analysis of efficient lossless data compression systems, PhD Thesis, Brown University, 1993.
[14] F.C. Johnson, C.D. Paice, W. Black, A. Neal, The application of linguistic processing to automatic abstract generation, J. Document and Text Management 1 (1993) 215–241.
[15] J.M. Kupiec, J. Pedersen, F. Chen, A trainable document summarizer, in: Proc. ACM SIGIR Conference on Research and Development in Information Retrieval, ACM Press, 1995, pp. 68–73.
[16] N.J. Larsson, A. Moffat, Offline dictionary-based compression, in: Proc. Data Compression Conference, IEEE Press, Los Alamitos, CA, 1999, pp. 296–305.
[17] J.R. Larus, Whole program paths, in: Proc. SIGPLAN 99 Conf. on Programming Languages Design and Implementation, 1999.
[18] J. Lovins, Development of a stemming algorithm, Mech. Transl. Comput. Linguistics 11 (1968) 22–31.
[19] A.R. Martin, Intelligent speech synthesis using the sequitur algorithm and graphical training: server software, M.S. Thesis, Engineering Science, University of Toronto, 1999.
[20] B.A. Nardi, J.R. Miller, D.J. Wright, Collaborative, programmable intelligent agents, Comm. ACM 41 (3) (1998) 96–104.
[21] C.G. Nevill-Manning, I.H. Witten, Identifying hierarchical structure in sequences: a linear-time algorithm, J. Artificial Intelligence Res. 7 (1997) 67–82.
[22] C.G. Nevill-Manning, I.H. Witten, Phrase hierarchy inference and compression in bounded space, in: J.A. Storer, M. Cohn (Eds.), Proc. Data Compression Conference, IEEE Press, Los Alamitos, CA, 1998, pp. 179–188.
[23] C.G. Nevill-Manning, I.H. Witten, G.W. Paynter, Lexically-generated subject hierarchies for browsing large collections, Internat. J. Digital Libraries 2 (2–3) (1999) 111–123.
[24] C.G. Nevill-Manning, I.H. Witten, Online and offline heuristics for inferring hierarchies of repetitions in sequences, Proc. IEEE 88 (11) (2000) 1745–1755.
[25] J.M. Ponte, W.B. Croft, Useg: a retargetable word segmentation procedure for information retrieval, in: Proc. on Document Analysis and Information Retrieval, Las Vegas, Nevada, 1996.
[26] W.J. Teahan, Modelling English text, PhD Thesis, University of Waikato, NZ, 1997.
[27] W.J. Teahan, Y. Wen, R. McNab, I.H. Witten, A compression-based algorithm for Chinese word segmentation, Comput. Linguistics 26 (3) (2000) 375–393.
[28] D. Tkach, Text mining technology: Turning information into knowledge, IBM White paper, 1997.
[29] P. Turney, Learning algorithms for keyphrase extraction, Information Retrieval 2 (4) (2000) 303–336.
[30] A.J. Viterbi, Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, IEEE Trans. Inform. Theory (1967) 260–269.
[31] I.H. Witten, Z. Bray, M. Mahoui, W.J. Teahan, Text mining: a new frontier for lossless compression, in: Proc. Data Compression Conference, IEEE Press, Los Alamitos, CA, 1999, pp. 198–207.

[32] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, second ed., Morgan Kaufmann, San Francisco, CA, 1999.

[33] J.G. Wolff, An algorithm for the segmentation of an artificial language analogue, British J. Psychol. 66 (1975) 79–90.

[34] S. Yeates, D. Bainbridge, I.H. Witten, Using compression to identify acronyms in text, in: Proc. Data Compression Conference (Poster paper), IEEE Press, Los Alamitos, CA, 2000. Full version available as Working Paper 00/1, Department of Computer Science, University of Waikato.

[35] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory IT-23 (3) (1977) 337–343.

[36] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inform. Theory IT-24 (5) (1978) 530–536.