

# Index Construction for Linear Categorisation

Vaughan R. Shanks

Hugh E. Williams

School of Computer Science and Information Technology, RMIT University  
GPO Box 2476V, Melbourne 3001, Australia  
{shanks,hugh}@cs.rmit.edu.au

May 29, 2003

## Abstract

Categorisation is a useful method for organising documents into subcollections that can be browsed or searched to more accurately and quickly meet information needs. On the Web, category-based portals such as Yahoo! and DMOZ are extremely popular: DMOZ is maintained by over 56,000 volunteers, is used as the basis of the popular Google directory, and is perhaps used by millions of users each day. Support Vector Machines (SVM) is a machine-learning algorithm which has been shown to be highly effective for automatic text categorisation. However, a problem with iterative training techniques such as SVM is that during their learning or training phase, they require the entire training collection to be held in main-memory; this is infeasible for large training collections such as DMOZ or large news wire feeds. In this paper, we show how inverted indexes can be used for scalable training in categorisation, and propose novel heuristics for a fast, accurate, and memory efficient approach. Our results show that an index can be constructed on a desktop workstation with little effect on categorisation accuracy compared to a memory-based approach. We conclude that our techniques permit automatic categorisation using very large training collections, vocabularies, and numbers of categories.

**Keywords** Categorisation, index construction, efficiency, support vector machines.

## 1 Introduction

Text categorisation is useful and popular for the management of large volumes of information. Indeed, the prevalence of hierarchical directories at online web portals such as Yahoo!<sup>1</sup> suggests that categorisation is a popular alternative to query-based searching. These hierarchical directory services are manually categorised by human maintainers who file each submitted web site into its appropriate categories. However, manual categorisation does not scale well to large numbers of documents, and has the disadvantage that categorisation decisions are subjective.

Automatic text categorisation using machines is a scalable, effective, and consistent alternative to manual categorisation. It may also be complementary: augmenting manually-constructed directories with automatically-labelled documents is an important potential application of automatic text categorisation. In such a process, automatic categorisation could be used to recommend each document be forwarded to a particular maintainer, who then performs the manual category allocation [27]. There are many other applications for automatic text categorisation, such as fighting the battle against spam emails, and prioritising incoming email or voicemail in a high-volume system.

Categorisation is typically approached as a machine-learning problem [9, 14, 31]. This works accurately and efficiently when applied to a small number of documents where all category models can be concurrently held in main-memory. However, when the number of categories and terms in the system is large — such as when categorising millions of documents for a web directory — a scalable, disk-based solution is required.

---

<sup>1</sup>See: <http://www.yahoo.com>

With few exceptions, recent work in the field of text categorisation has focused mostly on categoriser accuracy. In particular, *Support Vector Machines* (SVM) categorisers have been shown to be very effective for text categorisation tasks [19, 15, 33]. Beckerman et al. [1] have combined SVM with feature selection algorithms to reduce the dimensionality of training examples, and thus reduce SVM training time while maintaining accuracy. A related approach was proposed by Chakrabarti et al. [3], and aimed at producing a scalable categoriser comparable to SVM. In this approach, a *linear discriminant* is used to derive a lower-dimensional subspace onto which training examples are projected. The training examples are then used to construct a decision tree using Weka [31]. The resulting categoriser is scalable, and only slightly less accurate than SVM.

We have previously considered efficiency and scalability issues for categorisation. In particular, we have investigated how category information can be compactly represented in the inverted index structure that is used in all practical query-based text retrieval systems [24]. In this previous work, we have shown that indexes can be used for fast categorisation. However, we have not investigated efficient index construction for training algorithms which require category vectors be held in main-memory concurrently. In practice, this means our existing index construction techniques cannot be used on large training sets, but are a successful technique for fast categorisation after training is complete.

In this paper, we investigate the scalable and efficient construction of inverted indexes for text categorisation. We explore two main themes: first, the application of index construction techniques used in query-based retrieval to text categorisation; and, second, heuristics to avoid keeping all category models concurrently in main-memory. Our aim is to develop techniques that permit training with main-memory use that is independent of collection size, while also being fast enough to train on large collections.

Our results show that training of a text categoriser on large training collections with many categories is both efficient and scalable using our approaches. For a large collection of news stories, our scheme permits training on almost one gigabyte of data, where main-memory use is no longer proportional to collection size. Importantly, accuracy is typically unaffected and training proceeds at more than 10 Mb of data per minute in our research prototype with an SVM categoriser; a production implementation is likely to be much faster.

## 2 Background

In this section, we present an introduction to text categorisation, and focus on the application of Support Vector Machines (SVMs) to categorisation. We also describe our previous work in applying inverted indexing to categorisation.

### 2.1 Text Categorisation

Text categorisation is a two-phase process [23]. During the *training phase*, example documents are used to train *category models* for a series of predefined categories. During the *categorisation phase*, the system categorises unlabelled documents, using the information in the category models generated during the training phase.

Category models are learned using a *training set* of examples, where each of the documents in the training set is labelled with zero or more categories from a predefined set. For a particular category, the documents which are labelled with that category are *positive examples*, while the rest of the collection documents are *negative examples*. A *training algorithm* is used to analyse the example documents and form generalisations about the nature of documents in each category, and these generalisations are then summarised by a category model. This approach to training is a feature of all state-of-the-art categorisers, including Support Vector Machines (which we discuss later), Winnow [5], and Rocchio [17].

The accuracy of a categoriser can be tested during the categorisation phase by applying it to a test set of documents with zero or more known labels. Documents in the test set are treated as unlabelled documents, and the categoriser is used to apply category labels to these documents using the category models learned during the training phase. The accuracy of the categoriser is then measured by comparing the labels assigned by the categoriser with the known labels. This approach is commonly used in categorisation work [23] and we use it in this work.

The units of information extracted from a document are the document *features*. Many definitions of features have been proposed for the purposes of text retrieval and categorisation, including single words, word bigrams, word stems, phrases, synonyms, and collocations [13]. We use single words as the features of a document, so that each document is reduced to an unordered set of words or *terms*. Our particular definition of a term is the class ALNUM [29], where each term is delimited by whitespace and may contain at most one hyphen, one apostrophe, and no more than one consecutive digit.

## 2.2 The Vector Space Model

In text retrieval, the vector space model [20] is often used to represent a document as a vector of term weights. In this model, terms are the bag of words in the document, and each weight indicates the importance of the term in that document. Conventionally, two basic components are considered in computing such weights [20, 21]: first, the term frequency, that is, the frequency of the term in the document; and, second, the inverse document frequency, that is, the reciprocal of the number of collection documents that contain the term. In addition, weights are often adjusted using other metrics: for example, the length of the document [25] is used to normalise the weights for long or short documents, and text from hypertext links in web collections is used to add weight to terms or augment documents [4].

A popular term weighting metric in query-based retrieval that incorporates the term frequency, inverse document frequency, and document length factors is the Okapi BM25 measure [16]. In this scheme, each term  $t$  has a weight  $w_{d,t}$  in a document  $d$  computed using the following:

$$w_{d,t} = \frac{f_{d,t}}{0.5 + 1.5 \times (dl/dl_{avg} + f_{d,t})} \times \frac{\ln((N + 0.5)/f_t)}{\ln(N + 1)}$$

where the term frequency within the document  $d$  is  $f_{d,t}$ , the number of documents in which a term occurs in the training set is  $f_t$ , the number of documents in the training set is  $N$ , the document length is  $dl$ , and the average length of documents in the training set is  $dl_{avg}$ .

After computing weights, the vector space model [20] can be used to represent as a vector of length  $n$  the term weights of the  $n$  distinct terms in the document. This approach can be generalised for a category: a *linear categoriser* represents a category as a vector, in the same space as the vectors of the example documents from that category that are used in training. During the training phase, positive and negative example vectors are combined by a training algorithm to produce this single category vector, which can be thought of as a prototypical document of the category. In our experiments, the weights in the category vector are determined through the Okapi BM25 measure discussed above and the training phase is performed by a Support Vector Machine.

## 2.3 Support Vector Machines

Support Vector Machines (SVM) is a machine-learning technique which has proved very effective when applied to text categorisation [9]. SVM is suited to text categorisation problems because of the high dimensional feature space, few irrelevant features, and sparse document vectors that the text categorisation problem exhibits [9]. There are several public domains implementations available for research purposes<sup>2</sup>.

For our work, we use the *SVM<sup>light</sup>* package [9]. *SVM<sup>light</sup>* is an efficient, robust implementation which is popular in text categorisation research [12, 33]. In unreported experiments we have found that *SVM<sup>light</sup>* is more effective than the Rocchio [17] and Winnow [5] training algorithms in the tasks we describe in Section 4, and we therefore use *SVM<sup>light</sup>* as a baseline for our experiments. However, accuracy is not the main focus of this research: our area of interest in this paper is scalability for training, and the tradeoff between accuracy and the heuristics used to develop a fast, scalable solution.

SVM is used to train a categoriser using positive and negative examples, represented as feature vectors; as discussed earlier, our feature vectors are Okapi BM25 weights for each distinct document term. The SVM algorithm divides the vector space into two regions using a hyperplane of the form  $\vec{w} \cdot \vec{x} + b = 0$ , where  $\vec{w}$  is a tangent to the hyperplane,  $b$  is a threshold value, and  $x$  is any point in vector space. If any given point  $\vec{x}$  in the vector space is assigned a value  $y$ , such that  $y = 1$  if the point is on the positive side of

<sup>2</sup>See, for example, <http://www.kernel-machines.org/>

the hyperplane, and  $y = -1$  if the point is on the negative side of the hyperplane, then the following holds true for all points:

$$y(\vec{w} \cdot \vec{x} + b) \geq 0$$

In this paper, we confine our discussion of SVMs to linear kernels, as these have been found to work well with text categorisation [9].

An SVM is designed to misclassify as few training examples as possible, without overfitting to the training data. Ideally, the objective is to construct a hyperplane with all positive and negative examples  $x_i$  separated by the hyperplane by at least a unit margin. However, not every classification problem is linearly separable and, therefore, it is necessary to introduce *slack variables* to handle the case where an example document is misclassified. These slack variables must be kept to a minimum in order to minimise the number of misclassified examples; the techniques for constructing a solvable minimisation problem from these constraints is beyond this paper and is described in detail elsewhere [2].

The solution of the SVM optimisation problem is a vector  $\vec{\alpha}$  of *multipliers*, one for each training example, and a threshold,  $b$ . These  $\alpha$  multipliers have a value between 0 and some limit  $C$ . Those documents whose multipliers have values greater than 0 are known as *support vectors*. To categorise the vector of an unlabelled document  $x$  using  $N$  support vectors  $s$ , the following is computed:

$$y = \left( \sum_{i=1}^N y_i \alpha_i \vec{s}_i \cdot \vec{x} \right) - b$$

If  $y > 0$ , the document is labelled with the category.

## 2.4 Inverted Indexing for Categorisation

Inverted indexes are used to support querying in all practical text retrieval systems [32]. An inverted index consists of two components: first, a *lexicon* or vocabulary of all distinct terms that occur in the collection; and, second, a set of *postings* for each term that lists the locations of the term in the collection. In practice, the lexicon is largely held in main-memory, while the much larger postings are stored on disk.

To support ranked and Boolean querying, a postings list contains tuples of the form  $\langle d, f_{d,t} \rangle$ , where  $d$  is a document identifier, and  $f_{d,t}$  is the frequency of the term in that document [35]. Consider an example postings list for the term “rodan”:

$$\langle 2, 4 \rangle \langle 3, 1 \rangle \langle 10, 2 \rangle \langle 14, 6 \rangle$$

This indicates that the term “rodan” occurs 4 times in document 2, once in document 3, twice in document 10, and 6 times in document 14. In practice, indexes are stored compactly using integer compression techniques [22, 30, 32].

We have previously proposed the adaptation of inverted indexes to permit fast categorisation [24] using the Rocchio weight learning technique [17]. Other approaches to implementing fast categorisation store category vectors that are derived during the training process [9, 14, 31]. In contrast, we have inverted the category vectors and stored postings lists for each distinct collection term, with the aim of improving categorisation speed and reducing main-memory requirements.

We have proposed storing postings lists that consist of tuples  $\langle c, w_c \rangle$ , where  $c$  is a category identifier and  $w_c$  is the learned weight of the term in the category  $c$ . Consider the following example of a postings list for the term “farmer”<sup>3</sup>:

$$\langle \text{grain}, 0.34 \rangle \langle \text{oil}, -0.05 \rangle \langle \text{coffee}, 0.56 \rangle \langle \text{acq}, 0.00 \rangle$$

This postings list indicates that “farmer” has a positive association with the categories *grain* and *coffee*, that there is no or contradictory information about the category *acq*, and that the term is uncharacteristic of the category *oil*.

To reduce the size of inverted lists, we have proposed quantisation to convert weights to integers. We omit categories with weights less than or equal to a threshold (usually the threshold is zero) from the postings. In addition, we compress both category number differences and weights using integer compression

<sup>3</sup>These category examples are from the Reuters-21578 collection [11]

schemes [6]. The result of this approach is a compact index that permits fast, accurate categorisation: when floating point values are quantised to 256 equal-sized bins and represented as an 8-bit integer, the index for the 312 Mb collection is just over 5 Mb in size, the categorisation accuracy is the same as the floating-point approach over three accuracy measures, and categorisation speed around 20 times faster than the floating point-based index [24].

However, despite these excellent results, the index construction process would require that the entire collection be held in main-memory during training with an SVM. In the next section, we propose techniques for constructing large inverted indexes for SVM categorisation.

### 3 Optimised Indexing for Categorisation

In this section we propose an efficient, scalable inverted index construction approach for categorisation using SVM. Section 3.1 outlines an efficient technique for the construction of an inverted index for linear categorisation. Section 3.2 describes novel heuristics for scalable index construction for large collections.

#### 3.1 Index Construction

Index-based query evaluation requires that an index is pre-computed. A commonly used technique for index construction is sort-based inversion [32]. Sort-based inversion has three phases:

1. **Parsing:** For each term of each document parsed, a triple of the form  $\langle t, d, f_{d,t} \rangle$  is stored, where  $t$  is the term,  $d$  is the document, and  $f_{d,t}$  is the frequency of  $t$  in  $d$
2. **Sorting:** Blocks of up to  $k$  triples (where  $k$  is the maximum number of triples which will fit in main-memory) are sorted, keyed first on term identifier and then on document identifier, and the block is written to disk. In practice, the partial lists written to disk have the same format as the final postings lists described earlier but contain information for a range of documents in the collection.
3. **Merging:** At least two blocks from the previous step are merged into a block, again keyed on term identifier and usually then on document identifier. When only one block remains, the result is a complete postings list for each term, and the location on disk of each postings list is then associated with the term in the lexicon.

This approach to index construction is used in all open source text retrieval systems that can index large collections; there are theoretically better-performing construction approaches [7, 32] but we are not aware of these being implemented in practice.

We propose a variant of sort-based construction for building inverted indexes for categorisation. Our approach consists of three phases performed in two passes through the collection, which we discuss in detail throughout this section:

1. **Lexicon accumulation:** The training set is parsed, and a term and category lexicon are accumulated. In addition, the disk location and category labels of all documents in the training set are stored
2. **Category training:** Each category in the lexicon is trained and the category vectors are stored in main-memory until a threshold is reached. When main-memory is exhausted, the category vectors are written out as temporary blocks containing postings lists in sorted term order
3. **Merging:** At least two blocks from the previous step are merged into a block that is again in sorted term order. When only one block remains, the result is a complete postings list for each term, and the location on disk of each postings list is then associated with the term in the lexicon.

In query-based retrieval, postings store information about documents, and documents are typically ordinally numbered by their occurrence in the collection. In contrast, our categorisation index stores postings that contain information about categories and it is unusual for collection documents to be ordered by category. Indeed, where documents can be labelled with more than one category label, a total ordering by

category may not be possible. Moreover, it is not possible to predict the number or distribution of categories based on collection size or document statistics. We therefore use a *lexicon accumulation* stage to gather information about categories and terms in the collection.

The lexicon accumulation stage requires a single pass through the collection and gathers category statistics prior to training. During the pass, a *document map* is created for the collection that stores tuples of the form  $\langle c, d_o \rangle$  where  $c$  is a category identifier and  $d_o$  is a document offset. After accumulation, this list is ordered by  $c$  and then  $d_o$ , so that document offsets are grouped together within categories and are in disk position order. This information is used to provide fast access to the documents in each category during the category training stage.

The lexicon accumulation stage also determines the distinct set of terms in the collection and their frequency of occurrence. These are pre-computed before training, so that term weights can be calculated for use in the category training stage. The term frequencies are then used to calculate an inverse document frequency for each term according to the Okapi BM25 formula described in Section 2.1:

$$idf_t = \frac{\ln((N + 0.5)/f_t)}{\ln(N + 1)}$$

where  $N$  is the total number of documents in the collection and  $f_t$  is the number of documents in which the term occurs. In addition, during the lexicon accumulation, an in-memory *pool* of negative training examples is constructed; this process is discussed in Section 3.2.

At the conclusion of the lexicon accumulation stage, statistics have been gathered concerning the number and location of category documents, the terms and their weights in the collection, and a negative document pool formed. In the category training stage, our scheme trains each category by iterating through the lexicon of categories accumulated in the first stage. In this work, we use an SVM in the training process; however, any linear categoriser can be used and our preliminary experiments were with the Rocchio weight learning scheme [17, 24].

To train a category vector we retrieve and process all positive category example documents identified in the first phase and selected negative documents from the negative document pool. The positive documents are retrieved in offset order using the document map and, in general, this necessitates random access to the collection. The aggregate time required for these passes through the collection — one for each category — is therefore typically more than the time required for the single parsing step in conventional sort-based construction. For each term in each positive document, a document-term weight  $w_{d,t}$  is calculated using the Okapi BM25 formula described in Section 2.1. The resulting list of  $\langle t, w_{d,t} \rangle$  tuples are sorted by term identifier and stored as a vector in main-memory. Negative vectors are selected from documents not in the category being trained; the process used in selection is detailed in the next section.

With the positive and negative vectors stored in memory, the training algorithm is used to learn a category vector. For the SVM algorithm, a category vector is derived by factorising out the vector  $\vec{x}$  to give:

$$y = \vec{x} \cdot \left( \sum_{i=1}^N y_i \alpha_i \vec{s}_i \right) - b$$

The vector resulting from the bracketed expression is a combination of support vectors for all training documents, which needs to be calculated only once during training to later categorise all unlabelled documents for that category; however, this combination of support vectors is only possible with a linear SVM. Interestingly, the categorisation computation — now reduced to a dot product and a threshold — resembles a cosine similarity measure [20] used in query-based retrieval and, as we show later, permits fast categorisation of unlabelled document vectors.

As in a conventional sort-based construction approach, category vectors are written to disk when a main-memory threshold is reached. The category vectors are stored as partial inverted lists in term identifier order. At this stage, weights are written for all terms in all categories, and category identifiers can therefore be omitted from the partial postings lists to improve compressibility. As in our previous work [24], weights are stored as quantised 8-bit integer values and compressed using Elias gamma coding [6]. As we discuss later, we found that 8-bit quantisation works well for our SVM approach.

Unlike our previous work with the Rocchio weight learning method — where all weights are in the range 0 to 1 — the SVM approach has weights that are not limited to a pre-defined range. Therefore,

we keep track of a minimum and maximum weight value observed in training, and use these during the merging phase to decode the floating point values from the quantised weights.

The merging phase combines the temporary block files into a final index structure. Each block file is processed sequentially, in term order, and quantised weights are restored to floating point values and then quantised into the final range of values for the combined postings list. For some training sets, where large vocabularies and large numbers of categories are present, system limitations can prevent merging of all block files concurrently. In these situations a recursive algorithm can be used, where the maximum number of block files are merged into a single block file, and process repeated until only one block file remains; this is the same process as is often used in sort-based construction [28].

The process of quantising weights into temporary blocks, then restoring weights to approximate floating point values, and then quantising again into a final global range involves two lossy steps. However, the lossy approach has no effect on accuracy. For example, in preliminary experiments with the small Reuters-21578 collection, raw floating point values gave an  $F_1$  value of 80.5% during testing and using 8-bit quantisation gave an  $F_1$  of 80.4% (precision and recall were also unaffected, and we discuss collections and accuracy measures further in Section 4). It is therefore unnecessary to store minimum and maximum weights per term or per category.

## 3.2 Scalable Training

Our approach to index construction has the significant limitation that the collection — the complete set of positive and negative examples — must be held in main-memory during training. The SVM and Winnow categorisers assume that the training set fits into main-memory, and so would require modification to support a disk-based approach; for example, the *SVM<sup>light</sup>* package that we use in our experiments assumes a main-memory model. We consider practical alternatives in this section.

A multi-category text categoriser effectively consists of one categoriser per category. These single-category categorisers are *binary classifiers*, that is, the decisions they make can have only two possible outcomes: to classify an unknown example as being in the given category or not. In text categorisation, the binary classifier discriminates between examples that are on a single topic and examples that are concerned with any other topic; the other topics can be thought of as a general model of the training set.

For a large training set of documents (in which typically no category has a significant number of examples) the set of positive examples is much smaller than the set of negative examples. One approach to reduce the amount of data used in training is to use only selected features, that is, to ignore selected words [10, 34]. Another approach is to ignore documents, and this is the approach we consider here. We hypothesise that the negative document example set does not need to consist of all documents in the training set that are not members of the set of positive topic examples.

When learning a category model, a training algorithm should be able to generalise better about what constitutes a document of that category if the number of positive examples is large. A categoriser which does not have enough positive examples may not have a sufficiently broad vocabulary of the target category, and may overfit to the particular examples provided. In unreported experiments using Rocchio, Winnow, and SVM categorisers we have found that categories with relatively large numbers of positive example documents generally tend to be categorised more accurately; these observations are consistent with other work [5, 8, 33]. Therefore, in general, a categoriser should be trained using as many positive examples as possible, subject to main-memory limitations.

In text categorisation, a negative example is any document in the training set that is not a positive example for the category being trained. Therefore, for most applications, the number of negative training examples is large relative to the number of positive examples. Given a training set of  $N$  categories, a category which has close to the mean number of documents per category will have approximately  $N - 1$  times as many negative examples as positive examples. When  $N$  is large, the negative set is approximately the size of training collection.

We propose reducing the set of negative examples to a smaller representative set to permit training on large collections using limited main-memory. Our aim is to select a *negative pool* that is at most as large as the available main-memory, thus ensuring that negative training data is sourced only once from disk. To test whether such an approach allows both scalable and accurate categorisation, we consider two issues: first, how to select a representative subset of negative examples from the larger set; and, second, how many

negative examples should be used to train each category. We use this approach in parallel with our index construction scheme to permit scalable training.

If only selected negative examples are used in training, these examples should be representative of the complete set of negative examples. Moreover, the negative examples should not overlap with the set of positive examples, and the set should have a similar distribution in vocabulary and topics as the complete set. We therefore propose a two-step approach to selecting negative examples: first, we propose forming a pool of negative examples by randomly selecting examples from the collection; and, second, for each category we propose selecting random examples from the negative pool for use in training, with the condition that these examples are not members of the positive example pool for the category being trained.

Given a negative pool of  $k$  documents, and a requirement for no more than  $n$  negative examples per category, we therefore propose the following algorithm:

1. From the document map generated during the lexicon accumulation stage, randomly select  $k$  documents and store these in the set  $\mathcal{K}$  – the *negative pool*.
2. Parse the documents in  $\mathcal{K}$ , and generate a document vector for each, stored in main-memory.
3. For each category  $c_i$ :
  - (a) Remove document vectors of category  $c_i$  from  $\mathcal{K}$
  - (b) Create an empty set  $\mathcal{N}$  — the set of *negative examples*
  - (c) While  $|\mathcal{N}| < n$  and  $|\mathcal{K}| > 0$ :
    - i. Randomly remove a document vector  $d$  from set  $\mathcal{K}$  temporarily and add it to set  $\mathcal{N}$
  - (d) Parse the positive documents of set  $c_i$  from the training set into a set  $\mathcal{P}$  of *positive example vectors*, held in main-memory
  - (e) Train a model for category  $c_i$  using  $\mathcal{P}$  and  $\mathcal{N}$
  - (f) Restore the document vectors of category  $c_i$  to the negative pool  $\mathcal{K}$
  - (g) Empty sets  $\mathcal{P}$  and  $\mathcal{N}$

In the next section, we show how the choice of  $k$  and  $n$  affects the accuracy and speed of categorisation.

## 4 Results

We present our results in this section. We begin by discussing the text collection and accuracy measures we use. We then present overall results that show our index construction process works for a large collection, and the effect of choosing different negative pool sizes and negative example set sizes. We also briefly discuss the effects of thresholding, and confirm our previous result that shows fast categorisation is possible with an indexed approach.

### 4.1 Test Collection and Environment

We use the *Reuters-2000* corpus of newswire documents [18] in our experiments<sup>4</sup>. Each document in the corpus is tagged with several codes that signify which regions, industries, and topics pertain to the document, and we use the topic codes as categories. The collection has 126 possible topic codes but only 103 are used. The topic codes applied to the documents in the collection are hierarchical, divided roughly into four genres, MCAT, ECAT, CCAT, and GCAT, which are then divided into a hierarchy of subtopics. To remove high-level topic codes, we followed a technique frequently used with this collection [15], and removed any topic which occurred in more than 5% of the documents in the training set, and any topic not found in the training set. This left a set of 85 topics that we use in our experiments.

To generate a training set large enough to test the performance of our indexing techniques, we use the first 75% of the collection as the training set. The remaining 25% of the collection is used as the test set to measure the accuracy of our categoriser. The training set contains documents from 20 August 1996 to

---

<sup>4</sup>See: <http://about.reuters.com/researchandstandards/corpus/>

Pool Size	Negative Examples	Recall	Precision	$F_1$	Training Time (min)
100,000	100,000	0.5134	0.8003	0.6063	825
30,000	30,000	0.6221	0.6944	0.6408	223
20,000	20,000	0.6526	0.6602	0.6410	155
20,000	10,000	0.6998	0.6054	0.6321	108
20,000	5,000	0.7394	0.5379	0.6032	73
20,000	2,000	0.7778	0.4645	0.5597	57
20,000	1,000	0.8033	0.4161	0.5235	33
10,000	10,000	0.6947	0.6015	0.6290	88
10,000	5,000	0.7387	0.5432	0.6086	56
10,000	2,000	0.7780	0.4620	0.5584	36
10,000	1,000	0.8011	0.4159	0.5216	27
5,000	5,000	0.7388	0.5439	0.6086	56
5,000	2,000	0.7807	0.4653	0.5601	37
5,000	1,000	0.8044	0.4112	0.5203	29
2,000	2,000	0.7798	0.4650	0.5590	37
2,000	1,000	0.8014	0.4199	0.5268	28
1,000	1,000	0.8008	0.4103	0.5174	27

Table 1: A comparison of different negative pool sizes and negative example set sizes, showing accuracy and the time required to train for each combination.

25 May 1997, and the test set from 25 May 1997 until 19 August 1997. In all, the training set contains 609,073 documents in 930 Mb of text, with 452,901 unique terms, and the test set 197,718 documents in 310 Mb.

The experiments were run on an Intel Pentium-based quad-CPU machine with 2 Gb of main-memory, running the Linux operating system. For all experiments, the machine was under light-load, that is, no other significant processes were running.

## 4.2 Accuracy Measures

The accuracy of our results are reported using the well-known measures of mean macroaveraged recall, mean macroaveraged precision, and mean macroaveraged  $F_1$  [26, 32]. Recall and precision quantify the ability of a retrieval technique to return all of the relevant answers and only the relevant answers respectively. In text categorisation, recall is the ratio of the number of relevant documents that are assigned to a category to the number of documents that are members of that category. Precision is the ratio of the number of relevant documents that are assigned to a category to the number of documents assigned to that category. The  $F_\beta$  measure [26] combines recall and precision as a weighted average. With  $\beta = 1$ , this becomes the  $F_1$  measure, which equally weights recall  $r$  and precision  $p$  as:

$$F_1 = \frac{2rp}{r + p}$$

We use  $F_1$  as our primary evaluation metric for accuracy. We calculate recall, precision, and  $F_1$  for each category, and then average these across the categories to compute the mean macroaveraged recall, precision, and  $F_1$  values. The mean macroaveraged value gives equal weight to categories with few or many training documents, and so provides a stringent measure of the effectiveness of each technique.

## 4.3 Overall Results

Our overall results are shown in Table 1. The first row with a pool size of 100,000 documents — or around one-sixth of the training set — is presented as a baseline that shows the effects of using a significant fraction of all examples from the collection. Due to resource limitations, we were unable to produce a baseline using

the entire collection, however we were able to train a categoriser for the first five categories occurring in the training set (M12, GJOB, C24, E41, and E11) using all documents. Over the five categories trained on the entire training set, the categoriser scores a macroaveraged  $F_1$  of 75%; on the same five categories, trained on 100,000 negative examples, the macroaveraged  $F_1$  was 74%. Based on this result we chose to use a pool size of 100,000 as our baseline. At this setting, training is slow, with a throughput of only 0.8 Mb per minute; this is primarily because of the large number of examples used in training each category.

Our results show that using large numbers of negative examples is both slow and no more accurate than using less. With a pool and example set size of 30,000, the  $F_1$  measure under testing using the test collection is around 3.5% better than with 100,000 examples, while with only 10,000 examples it is 2% higher. Because of thresholding, recall is around 11% lower for 100,000 examples than for 30,000 examples, while precision is almost 11% higher for 100,000 than 30,000. We believe this effect occurs because as the number of negative examples is reduced, the negative examples remaining in the vector space are distributed more sparsely, causing the hyperplane to move in the negative direction. This effect can be compensated for by manually adjusting the threshold, or by varying the threshold using an heuristic approach; in our experiments, we optimised our results, using the training set, for pool sizes of between 10,000 and 30,000 by manually biasing the threshold  $b$  by  $+0.8$ . We plan to investigate thresholding further in our future work.

The initial negative pool size has almost no effect on accuracy. For example, with a pool size of 20,000 and selecting 5,000 examples, the  $F_1$  is around 60%. For pool sizes of 10,000 and 5,000 (but still selecting 5,000 examples), the  $F_1$  remains the same, that is, diversity of the example set does not affect the overall accuracy. We therefore conclude that the pool size should be maximised and all examples in the pool used in training, but that this should be considered in terms of training times.

Training times are highly dependent on the number of negative examples that are used. For example, with 30,000 negative examples per category, the training time is reduced by over 70% compared to using 100,000 examples, and gives an indexing throughput of around 4 Mb per minute; our implementation is a research prototype, and speed can be significantly improved in a production implementation. For smaller set sizes, the time savings are even more dramatic: for 10,000 examples, the indexing process proceeds at almost 11 Mb per minute, and at almost 17 Mb per minute for 5,000 examples. These results are unimpressive when compared to constructing indexes for query-based retrieval; however, index construction for query-based retrieval is much less disk intensive, since only one pass through the collection is required and no random accesses are needed to retrieve documents and, additionally, SVM learning is much more computationally intensive than conventional inversion.

Index size and categorisation speed are largely unaffected by the size of the negative pool. As we reported in our previous work, categorisation using an index is fast: using the indexes constructed for Reuters-2000 in these experiments, categorisation of the test collection takes between 430 and 475 seconds, depending on the index used; this is a throughput of around 39 to 45 Mb per minute. Again, we believe this figure can be significantly improved in a production implementation.

## 5 Conclusion

Automatic text categorisation is a fast and accurate technique for assigning documents to categories. However, it is often limited in its usefulness by constraints in the number of examples available for learning, the difficulties of working with noisy data, algorithmic limitations that impact on efficiency, and other factors. In this paper, we have addressed the efficiency of categorisation by proposing a scalable, fast, and accurate approach to training.

We have previously proposed using an inverted index structure to store weights for categories, and shown that this permits fast categorisation of unknown documents. In this paper, we have considered how this index can be constructed using two techniques: first, a variation of the sort-based construction technique used in text retrieval; and, second, a sampling approach that avoids the need to keep all examples in main-memory simultaneously. Using our novel techniques, an index can be constructed in a fixed amount of main-memory that is not dependent on the collection size or number of categories. We have shown that our approach permits training of a categoriser that is almost as accurate as using significant main-memory and large numbers of examples, while being both fast and scalable.

We plan to investigate several avenues as future work. We will consider whether other index construction techniques that are known to be theoretically more efficient are actually efficient in practice for categorisation. We also plan to consider other thresholding and validation techniques when a negative pool is used that may permit better tradeoffs in training between recall and precision. Last, we plan to generate categories for a very large web collection, and perform both training and testing experiments in that environment.

## Acknowledgments

This work is supported by the Australian Research Council and the Search Engine Group at RMIT University.

## References

- [1] R. Bekkerman, R. El-Yaniv, N. Tishby, and Y. Winter. On feature distributional clustering for text categorization. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 146–153, New Orleans, LA, 2001. ACM Press, NY.
- [2] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.
- [3] S. Chakrabarti, S. Roy, and M. Soundalgekar. Fast and accurate text classification via multiple linear discriminant projections. In *Proc. of 28th International Conference on Very Large Data Bases*, pages 658–669, Hong Kong, China, August 2002. Morgan Kaufmann.
- [4] N. Craswell, D. Hawking, and S. Robertson. Effective site finding using link anchor information. In W.B. Croft, D.J. Harper, D.H. Kraft, and J. Zobel, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 250–257, New Orleans, LA, 2001. ACM Press, NY.
- [5] I. Dagan, Y. Karov, and D. Roth. Mistake-driven learning in text categorization. In C. Cardie and R. Weischedel, editors, *Proc. Conference on Empirical Methods in NLP*, pages 55–63, Providence, RI, 1997.
- [6] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.
- [7] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*. (To appear).
- [8] T. Joachims. A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization. In D. H. Fisher, editor, *Proc. International Conference on Machine Learning*, pages 143–151, Nashville, TN, July 1997. Morgan Kaufmann.
- [9] T. Joachims. Text categorisation with support vector machines: Learning with many relevant features. In C. Nédellec and C. Rouveirol, editors, *Proc. European Conference on Machine Learning*, pages 137–142, Chemnitz, Germany, 1998. Springer.
- [10] D. Koller and M. Sahami. Hierarchically classifying documents using very few words. In D.H. Fisher, editor, *Proc. of the 14th International Conference on Machine Learning (ICML97)*, pages 170–178, Nashville, 1997. Morgan Kaufmann.
- [11] D. Lewis, R. Schapire, J. Callan, and R. Papka. Training algorithms for linear text classifiers. In H. Frei, D. Harman, P. Schäuble, and R. Wilkinson, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 296–306, Zurich, Switzerland, August 1996.
- [12] D. D. Lewis. Applying support vector machines to the TREC-2001 batch filtering and routing tasks. In E. M. Voorhees and D. K. Harman, editors, *Proc. Text REtrieval Conference*, Gaithersburg, MD, 2001. NIST publication number SN003-003-03750-8.
- [13] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.
- [14] A. K. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. Available at: <http://www.cs.cmu.edu/~mccallum/bow>, 1996.
- [15] S. Robertson and I. Soboroff. The TREC 2001 filtering track report. In E. M. Voorhees and D. K. Harman, editors, *Proc. Text REtrieval Conference*, pages 26–37, Gaithersburg, MD, 2001. NIST publication number SN003-003-03750-8.

- [16] S. E. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track. In E. M. Voorhees and D. K. Harman, editors, *Proc. Text REtrieval Conference*, pages 253–264, Gaithersburg, MD, November 1998. NIST publication number SN003-003-03614-5.
- [17] J.J. Rocchio. Relevance feedback in information retrieval. In G. Salton, editor, *The SMART Retrieval System: Experiments in Automatic Document Processing*, pages 313–323. Prentice-Hall, 1971.
- [18] T.G. Rose, M. Stevenson, and M. Whitehead. The Reuters corpus volume 1 - from yesterday's news to tomorrow's language resources. In *Proc. Third International Conference on Language Resources and Evaluation*, Las Palmas de Gran Canaria, may 2002.
- [19] D. Heckerman S. T. Dumais, J. Platt and M. Sahami. Inductive learning algorithms and representations for text categorization. In K. Makki and L. Bouganim, editors, *Proc. International Conference on Information and Knowledge Management*, pages 148–155, Bethesda, MD, November 1998.
- [20] G. Salton. *Automatic Text Processing*. Addison Wesley, Massachusetts, 1989.
- [21] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.
- [22] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In K. Järvelin, M. Beaulieu, R. Baeza-Yates, and S. H. Myaeng, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 222–229, Tampere, Finland, 2002.
- [23] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.
- [24] V. R. Shanks, H. E. Williams, and A. Cannane. Indexing for fast categorisation. In *Proc. Australasian Computer Science Conference*, pages 119–127, Adelaide, Australia, 2003. Australian Computer Society.
- [25] A. Singhal, G. Salton, M. Mitra, and C. Buckley. Document length normalisation. *Information Processing & Management*, 32(5):619–633, 1996.
- [26] C.J. van Rijsbergen. *Information Retrieval*. Butterworths, second edition, 1979.
- [27] W. Wibowo and H.E. Williams. Strategies for minimising errors in hierarchical web categorisation. In C. Nicholas, D. Grossman, K. Kalpakis, S. Qureshi, H. van Dissel, and L. Seligman, editors, *Proc. International Conference on Information and Knowledge Management*, pages 525–531, McLean, VA, 2002.
- [28] H.E. Williams. *Indexing and Retrieval for Genomic Databases*. PhD thesis, RMIT University, 1998.
- [29] H.E. Williams and J. Zobel. Searchable words on the web. *International Journal of Digital Libraries*. (to appear).
- [30] H.E. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.
- [31] I. H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. J. Cunningham. Weka: Practical machine learning tools and techniques with Java implementations. In N. Kasabov and K. Ko, editors, *Proc. ICONIP/ANZIIS/ANNES'99 International Workshop: Emerging Knowledge Engineering and Connectionist-Based Information Systems*, pages 192–196, Dunedin, NZ, 1999.
- [32] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA, second edition, 1999.
- [33] Y. Yang and X. Liu. A re-examination of text categorization methods. In M. A. Hearst, F. Gey, and R. Tong, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 42–49, Berkeley, CA, August 1999.
- [34] Y. Yang and J.O. Pedersen. A comparative study on feature selection in text categorization. In D.H. Fisher, editor, *Proc. of ICML-97, 14th International Conference on Machine Learning*, pages 412–420, Nashville, TX, 1997. Morgan Kaufmann.
- [35] J. Zobel and A. Moffat. Exploring the similarity space. *ACM SIGIR Forum*, 32(1):18–34, Spring 1998.