

# Scalable Document Classification

*Jae-Moon Lee*<sup>(1,2)</sup>, *Rafael A. Calvo*<sup>(2)</sup>

(1) School of Information and Computer Engineering,  
Hansung University, Korea

(2) Web Engineering Group  
School of Electrical and Information Engineering  
University of Sydney (*rafa@ee.usyd.edu.au*)

## Abstract

*This paper describes the design and implementation of new Naïve Bayes and k-Nearest Neighbour methods that are highly scalable and efficient for document classification. Three methods for improving scalability are analysed: a change in the data representation and therefore in the algorithms' implementation, a partitioning mechanism that breaks down the problem into smaller parts, and a buffering mechanism to improve memory efficiency for large datasets. The classifiers were tested over two Reuters datasets: ModApte a popular but small benchmark, and RCV1 a new large collection of news stories, and compared to more standard implementations of these methods, both experimentally and analytically.*

## 1 Introduction

Information overload in which individuals are faced with an oversupply of content could become the road rage of the new millennium. In order for this content to become useful information and empower users, rather than frustrate or confuse them, we need novel ways of delivering only what is needed, at the right time and in the right format. News stories are the most important source of up-to-date information about the world around us, and represent some of the most often updated and highest quality content on the web. Therefore, it is most important to develop ways to process news efficiently. In this paper we have studied well known machine learning models, and modified them to improve the scalability in the automatic classification of large numbers of news stories and other types of documents. Language technologies have provided excellent tools for information retrieval by exploiting the content being indexed. In the beginning of the internet, search engines on the web only indexed information from unstructured text (i.e. Altavista); later they used the topology of the net to find which component of the information being indexed was more im-

portant (i.e. Google). These changes have considerably improved information retrieval. Document classification, meanwhile, has usually been done manually by thousands of human indexers, in private catalogues or in large scale ones such as Yahoo! and the Open Directory Project. Instead of human labour, automatic document classification tools [1, 8, 9] use statistical models, similar to those in information retrieval. These systems can be used to create hierarchical taxonomies of content such as those commonly found on the web, on e-learning systems and even on traditional library information systems. News story articles are another example; they are written by reporters from all over the world working for news agencies such as Associated Press and Reuters. These agencies collect the news, edit it and sell bundles of articles to the periodicals accessed by web users (e.g. news.yahoo.com, Sydney Morning Herald, etc). It is important for both the agencies and the periodicals to have an organized well-managed stream of news. News is normally classified according to taxonomies that are relevant to readers, (e.g. politics, Iraq or oil). This classification can be very difficult because it requires human expertise to spot relationships between the taxonomy and the documents. Even the experts do not agree on what should go where and inter-indexer consistency in classification has considerable variation [7].

Automatic classification techniques use machine learning algorithms that learn from these human classifications, so they can only do as well as the people did in the training data provided. In addition, different algorithms can learn different types of patterns in the data. In order to compare the classification performance of different algorithms, researchers have a set of standardised benchmarks with a particular dataset and a well defined task. The most popular classification benchmark during the late nineties was a Reuters collection called Reuters-21578 (based on Reuters-22173) with 12,902 documents, that had to be classified in about 100 different categories [1, 10]. This benchmark is still used to compare the performance of different algorithms but the challenges now lie in moving towards larger scale document classification [6]. In 2002, Reuters released a new research corpora with over 800,000 documents that we discuss in this paper.

There are several Machine Learning (ML) algorithms that have been successfully used in the past [1, 5, 8, 10]. They include Neural Networks, Naïve Bayes, Support Vector Machines (SVM) and k-Nearest neighbours (kNN). Each of these methods has their advantages and limitations on classification performance and scalability. The choice of algorithms will depend in the application, and the amount of data to be used. In web applications, efficiency is of particular importance, since the large number of users and data can make some algorithms impractical.

Section 2 gives a brief introduction to document classification and the notation used in this paper. After describing the vector model and the performance measures used in document classification, we describe the two methods used in this paper: in Section 2.1 we describe the Naïve Bayes classifier and in Section 2.2 the k Nearest Neighbour (kNN), both well studied methods in the document classification literature. Section 2.3 describes our benchmark data sets:

the small and well studied ModApte [12] and the new, large Reuters RCV1. Our contribution in developing scalable document classification methods is described in section 3. In section 3.1 we review the design and architecture of our object oriented classification framework and discuss good engineering practices. In section 3.2 we describe our improvements to the Naïve Bayes method and its implementation, and in section 3.3 we do the same for the kNN methods. These improvements refer to three changes that make the classification system more efficient: a data representation that makes the algorithm more efficient, changes for partitioning and potentially parallelizing the methods, and changes for improving data buffering. In section 4 we evaluate the changes on the standard Naïve Bayes and kNN implementations, first analitically (section 4.1) and then experimentally (section 4.2). Section 5 includes a summary of the outcomes plus an indication of future research.

## 2 Automatic Document Classification Methods

As mentioned earlier, several Machine Learning (ML) algorithms that have been successfully used in the past have been thoroughly described in the literature [1, 5, 8, 10]. Normally, the choice of algorithms will depend on the application: the level of accuracy and scalability it requires and the type and amount of data to be used. In web applications, efficiency is of particular importance, since the large number of users and data can make some algorithms impractical. Very large datasets are also becoming more common. Companies like PLS, Lexis-Nexis, DIALOG and Verity have long since created systems that integrate the results of multiple heterogeneous databases [3]. These collections have billions of documents (ie. news stories) from thousands of sources.

This paper focuses on improving the scalability instead of the accuracy of the algorithms. In order to improve efficiency we need to discuss how the algorithms work and how their accuracy is measured. Automatic classification methods start by reducing the number of terms in a document using linguistic techniques such as stop-words and stemming. They are then represented as vectors, often using the Term Frequency (TF) and Inverse Document Frequency (IDF) schemes that weigh a term by the number of occurrences of a term in a document and in the overall collection. Once the documents are represented as vectors, machine learning and statistical techniques can be used. In the following sections we describe two of these in more detail: Naïve Bayes and kNN.

It is also useful to discuss how accuracy is measured in these algorithms. Table 1 describes the possible outcomes of a binary classifier. The “assigned” YES/NO results refer to the classifier output and the “correct” YES/NO refers to the ODP assigned categories. A perfect classifier would have a value of 0 for  $b_j$  and  $c_j$ .

Using Table 1 we define the three performance measures common in document categorization literature:

	Human Expert	
Machine Classifier	YES	NO
YES	$a_j$	$b_j$
NO	$c_j$	$d_j$

Table 1: Contingency table for class  $j$

$$\begin{aligned}
 \text{Recall} = R &= \begin{cases} \frac{a}{a+c} & \text{if } a+c > 0 \\ 0 & \text{otherwise} \end{cases} \\
 \text{Precision} = P &= \begin{cases} \frac{a}{a+b} & \text{if } a+b > 0 \\ 0 & \text{otherwise} \end{cases} \\
 F_1 &= \frac{2PR}{P+R} \text{ if } P+R > 0, 0 \text{ otherwise}
 \end{aligned}$$

The first two measures contain information about whether classification errors are dominated by false positives or false negatives. The trade-off between recall and precision can often be controlled by setting a classifier’s parameters. Both measures should typically be used to describe the overall performance as neither is particularly informative by itself. The third measure  $F_1$  is an average of  $R$  and  $P$ .

When dealing with multiple classes there are two possible ways of averaging these measures, *macro-averaging* and *micro-averaging*. In macro-averaging, one contingency table per class is used, then performance measures are computed on each of them and averaged. In micro-averaging only one contingency table is used; an average of all the classes is computed for each cell and the performance measures are obtained therein. The macro-average weights equally all the classes, regardless of how many documents they contain. The micro-average weights equally all the documents, thus biasing toward performance of common classes.

## 2.1 Naïve Bayes Classification

Naïve Bayes is a well known statistical method and has been successfully applied to classification tasks [8, 4]. The different forms of Naïve Bayes are based on the Bayes theorem for computing the conditional probability that given a document represented by  $d$  it belongs to category  $c$ :

$$P(c|d) = \frac{P(d|c)P(c)}{P(d)} \tag{1}$$

The most probable category is given by:

$$\text{ArgMax}_{c_j \in \mathcal{C}} P(c_j|d_i) = \text{ArgMax}_{c_j \in \mathcal{C}} \frac{P(d_i|c_j)P(c_j)}{P(d_i)} = \text{ArgMax}_{c_j \in \mathcal{C}} P(d_i|c_j)P(c_j)$$

The estimation of  $P(d_i|c_j)$  is difficult since the number of possible vectors  $d_i$  is too high. This difficulty is overcome by using the naïve assumption that any two coordinates of the document vector are statistically independent. Using this assumption, the most probable category  $c_j$  can be estimated.

$P(c_j)$  is estimated from the number of documents in the training set that belongs to  $c_j$ . To estimate  $P(d_i|c_j)$ , we use the terms of  $d_i$ :  $t_{i1}t_{i2}\dots t_{ik}$  and  $\text{ArgMax}_{c_j \in \mathcal{C}} P(c_j|d_i)$  is then estimated as:

$$\text{ArgMax}_{c_j \in \mathcal{C}} P(c_j|d_i) = \text{ArgMax}_{c_j \in \mathcal{C}} P(t_{i1} \dots t_{ik}|c_j)P(c_j) \approx \text{ArgMax}_{c_j \in \mathcal{C}} \prod_{k=1}^n P(t_{ik}|c_j)P(c_j)$$

Taking  $T$  as the total number of distinct terms in the training set and using Laplace smoothing, the definition of probabilities is  $P(t_{ik}|c_j)P(c_j)$  and the conditional probability for the terms can be written as:

$$P(t_{ik}|c_j) = \frac{1 + TF((t_{ik}|c_j)P(c_j))}{|T| + \sum_{s=1}^{N(c_j)} TF(t_s, c_j)}$$

The optimum category can be chosen by:

$$C_{\text{best}} = \text{ArgMin}_{c_j \in \mathcal{C}} P(c_j) + \sum_{k=1}^n \log P(t_{ik}|c_j)$$

This last expression is the one used in our implementation of Naïve Bayes.

Previous studies [12] have shown that Naïve Bayes is accurate and fast compared to other algorithms. But these studies evaluated Naïve Bayes' performance on relatively small datasets. In this project we study its scalability on a large dataset and develop variations that make it more scalable. In order to improve computational efficiency it is useful to study in detail the computational cost of the above method.

Step	pseudo-code
Step 1	$C_{rank}[c_j] = \log(P_c[c_j])$ foreach $c_j \in \mathcal{C}$ ;
Step 2	foreach $c_j \in \mathcal{C}$ {
Step 2.1	foreach $\langle t_k, w_k \rangle \in d$ {
Step 2.1.1	$w = (\langle t_k, any \rangle \in TF_{c \rightarrow t}[c_j]) ? any + 1 : 1$ ;
Step 2.1.2	$C_{rank}[c_j] + = w_k \times \log(w/S_c[c_j])$ ;
Step 2.2	}
Step 3	}

Table 2: Step by step breakdown of Naïve Bayes testing algorithm. Inputs:  $\mathcal{C}[], P_c[], S_c[], TF_{c \rightarrow t}[], d$ . Outputs  $C_{rank}[]$

In most applications the training of a classifier does not happen very often, so the complexity of the method that classifies new documents is more important.

Table 2 shows the pseudo-code for the testing (classification) phase of Naïve Bayes. Here,  $C[]$  is the set of categories,  $P_c[]$  is the probability for each category and  $TF_{c \rightarrow t}[]$  is the term frequency in the above  $C_{best}$ , where the symbol  $x \rightarrow y$  means the list of pairs  $\langle y, weight \rangle$  for each  $x$ , and finally  $d$  is the testing document which consists of pairs  $\langle term, weight \rangle$ . The output of Naïve Bayes is assumed to be the ranked categories, such as  $C_{rank}[]$ , which contain the score for each category.

Step 1 initializes an array of size  $|C|$ , so the cost is  $c_1 * |C|$ . Steps 2-3 are repeated  $|C|$  times, and 2.1-2.2 are repeated  $|d|$  times. 2.1.1 checks to see if  $t_k$  is or not a member of  $TF_{c \rightarrow t}$ , so the cost is  $h(|TF_{c \rightarrow t}|)$ , where  $h(x)$  is the cost of probing a hash table. Since the cost 2.1.3 is a constant such as  $c_2$ , the total cost of the algorithm can be represented as

$$T_{NB} = (c_1 + (h(|TF_{c \rightarrow t}^{avg}|) + c_2) * |d|) * |C| = O(|d| * |C|) \quad (2)$$

where  $|TF_{c \rightarrow t}^{avg}|$  is the average size of  $|TF_{c \rightarrow t}|$  and  $h(x)$  is assumed to be  $O(1)$ . Equation (2) shows how the complexity of Naïve Bayes is proportional to  $|d|$  the number of distinct terms in the testing document and  $|C|$  the number of categories. This is in sharp contrast to other algorithms such as kNN discussed in section 2.2 that are proportional to the number of training documents.

## 2.2 k-Nearest Neighbour classification

The k-nearest neighbours (kNN) has been well studied in the the statistical literature but was first used in document classification by Yang [11]. For classifying a document  $d$ , kNN looks for the  $k$  most similar documents in the training set. If a large enough proportion of these documents belongs to a particular class  $c_i$ , it is said that  $d \in c_i$ . This simple method can be extended by weighting with distances (or similarities) [11]. Table 3 contains the pseudo-code for the kNN method we used in our experiments.

Table 3,  $C[]$  is the set of categories,  $TF_{d \rightarrow t}[]$  is the term frequency in the training document,  $d$  is the testing document which consists of pairs  $\langle term, weight \rangle$  and  $k$  is the number of neighbours. The output is assumed to be the ranked categories such as  $C_{rank}[]$ .

The kNN algorithm described in Table 3 starts by normalizing the terms' weights in the test document. The complexity of step 1 is then  $c_1 * |d|$ . The cost of steps 2 and 3 is a constant. Steps 4-5 are repeated  $|TF_{d \rightarrow t}|$  times. Step 4.1 computes the similarity between two documents. If the similarity between the two vectors is defined as the dot product and a hash technique is used, its cost becomes  $c_2 * |TF_{d \rightarrow t}^{avg}|$ , where  $|TF_{d \rightarrow t}^{avg}|$  is the average size of  $d_i$ . Steps 4.2 - 4.3 may be performed in several ways and the cost depends on the choice made but since it aims to find the  $k$ th maximum value in  $D_k$ , its cost is proportional to  $c_3 * k$ . Thus, the total cost of steps 4-5 becomes  $(c_2 + c_3 * k) * |TF_{d \rightarrow t}|$ . The cost of step 6 is  $c_4 * |C|$ . The cost of 7 is  $c_5 * |D_k| * |C|$ , that is,  $c_5 * k * |C|$ . The cost of 8 is  $c_6 * |C|$ . The overall cost for the algorithm is then:

Step	pseudo-code
Step 1	$d = \text{normalize}(d);$
Step 2	$D_k = \{\}$
Step 3	$(d_m, m_s) = (\phi, 0);$
Step 4	foreach $d_i \in TF_{d \rightarrow t}$ {
Step 4.1	$s_i = \text{compute-similarity}(d, d_i);$
Step 4.2	if $(m_s < s_i)$ {
Step 4.2.1	$D_k = \{D_k \cup \{ \langle d_i, s_i \rangle \} \} - \{ \langle d_m, m_s \rangle \}$
Step 4.2.2	$(d_m, m_s) = \text{find-minimal-similarity}(D_k)$
Step 4.3	}
Step 5	}
Step 6	$C_{rank}[c_j] = 0$ foreach $c_j \in C$
Step 7	foreach $d_j \in D_k$ {
Step 7.1	foreach $c_j \in C$ {
Step 7.1.1	$C_{rank}[c_j] ++$ if $d_j$ is classified to $c_j$
Step 7.2	}
Step 8	}
Step 9	$C_{rank} = \text{scaling}(C_{rank})$

Table 3: Step by step breakdown of kNN testing algorithm. Inputs:  $C[], TF_{d \rightarrow t}[], d, k$ . Outputs  $C_{rank}[]$ .

$$T_{kNN} = c_1 * |d| + (c_2 * |TF_{d \rightarrow t}^{avg}| + c_3 * k) * |TF_{d \rightarrow t}| + c_4 * |C| + c_5 * k * |C| + c_6 * |C|$$

Normally,  $|C| \ll |D|$ ,  $k$  is a small number (5-20) and  $h(x)$  is  $O(1)$ . Using these conditions we can simplify the above equation:

$$T_{kNN} = O(|TF_{d \rightarrow t}| * |TF_{d \rightarrow t}^{avg}|) = O(|D| * |TF_{d \rightarrow t}^{avg}|) \quad (3)$$

This result shows that the efficiency of kNN is linearly dependent on the number of training documents.

### 2.3 Benchmarking with the Reuters datasets

Reuters is one of the largest content producers in the world. Their news stories are read by millions every day. In order to improve the management of these news stories Reuters has made available for research purposes, a number of data sets. In this study we use two of them: the ModApte [12] set used for benchmarking new algorithms by many researchers in document classification and the newer RCV1 set, a collection not so well known but of much a bigger size, and therefore more appropriate for our project.

The ModApte version of Reuters-21578 consists of 12,902 documents from the Reuters financial news wire service, partitioned into a training set with 9,603 documents and 3,299 test documents, from which we removed documents not assigned to any categories. There are 90 categories and each document

may have belonged to one or more of them. The average number of categories per document is 1.2. Yang [12] provides a complete review of the performance achieved by different methods.

The Reuters RCV1 Corpus [7] consists of all English news stories (806,791) published by Reuters in the period between 20/8/1996 and 19/8/1997. The news is stored as files in XML format, using a News ML Document Type Definition (DTD). NewsML is an open standard being developed by the International Press and Telecommunications Council (IPTC). The news is written by approximately 2000 reporters and then classified by Reuters specialists in a number of ways. The classified news articles are then syndicated by websites such as news.yahoo.com and news.google.com or periodicals such as the Sydney Morning Herald that may or may not have a website. Due to seasonal variations, the number of stories per day is not a constant. In addition, on weekdays there is an average of 2,880 stories per day compared to 480 on weekends. Approximately 3.7Gb is required for the storage of the uncompressed XML files.

The NewsML schema contains metadata produced by human indexers about themes and categories. When two humans index a document differently they create inter-indexer variations. These can be measured using a correction rate  $C=(NC/NE)*100$ , where NE is the number of stories indexed by an editor and NC is the number of times an editor has been corrected by a second editor. Normally untrained editors will have a higher C than more expert editors, but even when all the editors are experienced, correction rates of 10% are common. In the RCV1 collection there are correction rates of up to 77%. Since ML algorithms learn from examples of classifications done by humans, correction rates are an important limiting factor to their performance. Performance measures in classification systems are really a measure of how much they correlate to the human classifiers, it is not possible for a student to be better than the teacher, or at least it is not possible for the teacher to know so. RCV1 data is stored in XML documents providing the metadata information normally required by news agencies and periodicals who need to deliver the stories to end users. NewsML defines a rich schema shown (simplified) in Figure 1, which we can see has entities for title, headline, text, copyright and several types of classification. For our experiments we have used all three available classifications (topic, country and industry) as a single task.

Web applications will increasingly exploit this type of metadata. As has been discussed by researchers studying the concept of the semantic web [2], the next revolution in the Internet will come when web applications have access to structured collections of information and a set of inference rules that can be used to perform automated reasoning. This project aims at producing such applications.

```

<?xml version="1.0" encoding="iso-8859-1" ?>
<newsitem itemid="23444" id="root" date="1996-08-29" xml:lang="en">
  <title>NETHERLANDS: PRESS DIGEST - Netherlands - Aug 29.</title>
  <headline>PRESS DIGEST - Netherlands - Aug 29.</headline>
  <dateline>AMSTERDAM 1996-08-29</dateline>
+ <text>
  <copyright>(c) Reuters Limited 1996</copyright>
- <metadata>
  - <codes class="bip:countries:1.0">
    + <code code="NETH">
      </code>
    </codes>
  - <codes class="bip:topics:1.0">
    + <code code="GCAT">
      </code>
    </codes>
    <dc element="dc.date.created" value="1996-08-29" />
  </metadata>
</newsitem>

```

Figure 1: An example of a NewsML document

## 3 Engineering for scalability

### 3.1 System Design and Architecture

Object Oriented Application Frameworks (OOAF) are software engineering artefacts that improve the reusability of design and implementation [5,6,10]. The classification framework used in this project [12] was designed to provide a consistent method to build document categorization systems. It allows the developer to focus on the quality of the more critical and complex modules by allowing reuse of common, simple base modules. In this project we study how the framework can be used in news story classification systems and extend the framework to be more scalable as required in web applications by adding an object buffering module.

We have implemented the faster Naïve Bayes and kNN classifiers using an Object Oriented Application Framework [2] for document classification [9]. The framework has been designed to increase reusability and offers a structured way to extend it. Figure 2 shows the overall structure of the framework. The *Knowledge* package includes the `KnowledgeSet` class that represents the set of documents, the set of categories, and the many-to-many mappings between them. The *Learner* package is made of the `Learner` class and subclasses. The abstract `Learner` class provides an interface to train on a set of pre-categorized documents. The result of asking a `Learner` to categorize a previously unseen document is a `Hypothesis` object, one of several that make up the *Hypothesis* package. These objects may be queried for reporting information such as, which categories were assigned, which was the single most appropriate category, what scores were assigned to each category, etc. Two of the most serious limitations when training ML algorithms, are the amount of data required to achieve satisfactory accuracy and the scalability issues that arise when training such al-

gorithms. Memory requirements are often the first obstacle for managing large corpora, particularly for a text categorization system, where the ML algorithms are being trained on very large vector spaces and large amounts of data. Most operating systems have dealt with these problems by adding support for virtual memory, and the applications often have an adaptive buffer management tool that allows the application to allocate specific amounts of memory. These general purpose techniques are not always enough or appropriate for special applications like ours.

## 3.2 A scalable Naïve Bayes implementation

### 3.2.1 An Efficient Naïve Bayes

Since Naïve Bayes' complexity is proportional to the number of categories and terms, not the number of documents, Naïve Bayes is faster and uses less memory than most other algorithms, including kNN. Most of the Naïve Bayes' computational load goes into searching for the terms of the test document into all the elements of  $TF_{c \rightarrow t}$  for all categories. This is normally done even when the term is not in the test document. The algorithm proposed here performs the search only on those terms that are in the particular document. The inputs and output are the same as those of Naïve Bayes except on  $TF_{t \rightarrow c}$ . It is the list of pairs  $\langle category, weight \rangle$  for each term in the training documents. The pseudo code for the proposed Naïve Bayes is shown in Table 4.

Step	pseudo-code
Step 1	$C_{rank}[c_j] = \log(P_c[c_j])$ foreach $c_j \in C$ ;
Step 2	$w_{total} = 0$ ;
Step 3	foreach $\langle t_k, w_k \rangle \in d$ {
Step 3.1	foreach $\langle c_j, w_j \rangle \in TF_{t \rightarrow c}[t_k]$ {
Step 3.1.1	$C_{rank}[c_j] + = w_k * \log(w_j + 1)$ ;
Step 3.2	}
Step 3.3	$w_{total} + = w_k$ ;
Step 4	}
Step 5	$C_{rank}[c_j] + = w_{total} * \log(1/S_c[c_j])$ foreach $c_j \in C$ ;

Table 4: Step by step Breakdown of fast Naïve Bayes testing algorithm. Inputs:  $C$ ,  $P_c$ ,  $S_c$ ,  $TF_{t \rightarrow c}$ ,  $d$ . Outputs:  $C_{rank}$

From Table 4, the cost of fast Naïve Bayes can be calculated as follows: The cost of Step 1 is  $c_1 * |C|$ . The cost of Step 2 is  $c_2$ . The step 3-4 are repeated as  $|d|$  times, and the step 3.1-3.2 are repeated as  $|TF_{t \rightarrow c}^{avg}|$  times, where  $|TF_{t \rightarrow c}^{avg}|$  is the average size of  $TF_{t \rightarrow c}$ . The step 3.1 contains the cost of finding the list with  $t_k$  in  $TF_{t \rightarrow c}$  such as  $TF_{t \rightarrow c}[t_k]$ , and its cost becomes  $h(|TF_{t \rightarrow c}^{avg}|)$ , where  $h(x)$  is the cost to find by using hashing. Thus, the cost of step 3-4 are  $(h(|TF_{t \rightarrow c}^{avg}|) + c_3 * |TF_{t \rightarrow c}^{avg}| + c_4) * |d|$ . The cost of step 5 is  $c_5 * |C|$ . Thus, the total cost can be summarized as follows:

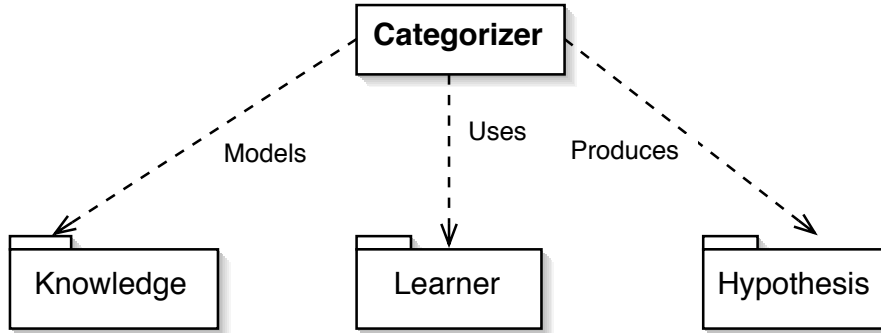


Figure 2: Packages in AI:Classifier

$$T_{FastNB} = (c_1 + c_5) * |C| + (h(|TF_{t->c}^{avg}|) + c_3 * |TF_{t->c}^{avg}| + c_4) * |d| = O(|TF_{t->c}^{avg}| * |d|) \quad (4)$$

### 3.2.2 Partitioning for Naïve Bayes

The "divide and conquer" approach is very commonly used to solve large scale computational problems, by dividing the task in a number of smaller sub-tasks that can be performed concurrently. Partitioning a data set is often used as a way of partitioning the problem and therefore reducing the time required to perform an operation. Partitioning is often used in parallel and distributed computing.

There are at least two ways for partitioning the data in a document classification task: document partitioning or category oriented partitioning. The document partitioning scheme shown in Figure 3 divides the data into N disjoint subsets of documents, represented as cylinders and shows how their categories might overlap. The category oriented partitioning has non overlapping categories.

The `PartitionedNaiveBayes` class, a subclass of `Learner` is similar to the one for kNN represented in Figure 7. It uses the `SingleNaiveBayes` class used in the rest of the paper.

### 3.2.3 Buffering Naïve Bayes

We have extended the text categorization framework [9] so that it can handle a much larger corpus. Figure 5 shows the object diagram of important classes in the run time of the framework. The circle in Figure 5 represents an object and the arrow represents a relationship meaning a category/document HAS-A feature vector (FV). The document object corresponds exactly to a document in the training or testing sets. The category object corresponds exactly to a

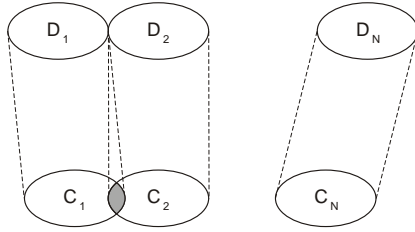


Figure 3: Document Oriented Partitioning

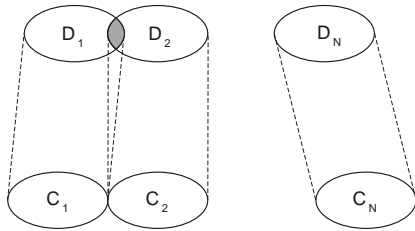


Figure 4: Category Oriented Partitioning

category. For the RCV1, there are over 800K documents, so the framework needs to manage that number of document objects and the same number of feature vectors. The object buffer can be configured to store as many objects as possible in RAM memory, requesting others from the hard disk as they are required. Thus, there is only a constant number of objects in the memory, and the system can be optimised for the available resources and independently on the amount of data. The buffered object module was implemented as described in 6. The module supports a unique persistent Object Identifier (OID) for all objects stored in it. The framework will then use this OID instead of the object or its reference. If the framework needs an object to be stored in the buffered object module, it makes a request to the buffered object manager using the OID.

The buffered object manager firstly searches for the requested object in the object pool. If the object manager finds it, then it returns the object's reference, otherwise it requests the object from the buffered file manager which tries to minimize I/O to the disk. The buffered file manager reads the requested object from the disk, then returns it to the buffered object manager. As the buffered object manager receives the object from the buffered file manager, it firstly stores the received object to the buffer pool then returns it to the requester in the framework.

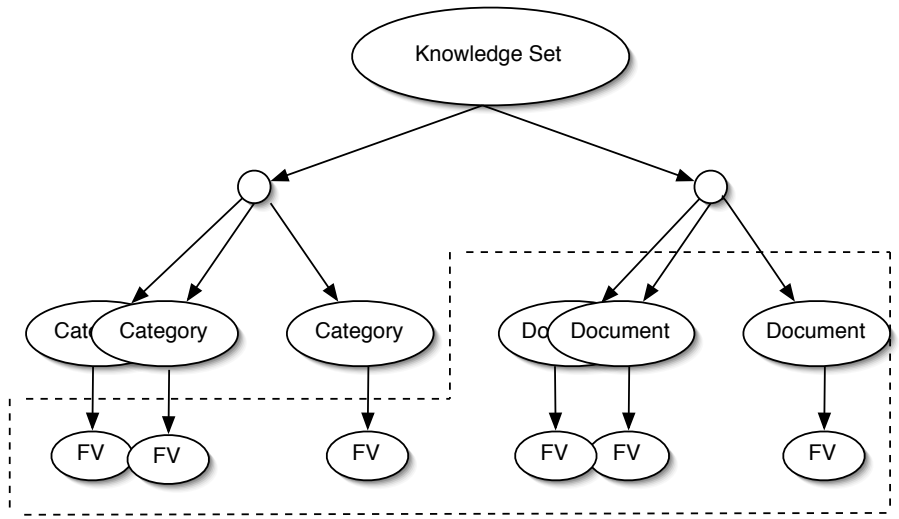


Figure 5: Run time object diagram

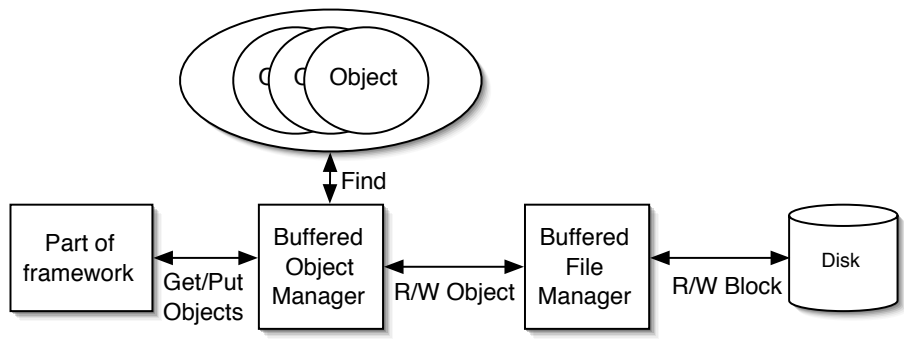


Figure 6: Architecture of the buffered object module

### 3.3 A scalable kNN implementation

#### 3.3.1 An Efficient kNN

kNN is one of the simplest methods in text classification and also very accurate. Regretably, due to its cost linear relationship with the number of training documents and their size, kNN can be very slow. Here we describe a new version of kNN that improves the classification time.

The way data is stored and accessed has an important impact on the performance of the algorithms. There are two methods to store the  $\langle term, weight \rangle$  pairs of all documents. The first one stores a list of  $\langle term, weight \rangle$  pairs for each document, the second one stores  $\langle document, weight \rangle$  pairs for each term. We use this data representation to rewrite the kNN method. Table 5 shows a breakdown of the computations performed in this "fast kNN".

Step	pseudo-code
Step 1	$d = \text{normalize}(d)$ ;
Step 2	for( $i=0$ ; $i <  TF_{t \rightarrow d} $ ; $i++$ ) $\text{score}[i] = 0$ ;
Step 3	foreach $\langle t_k, w_k \rangle \in d$ {
Step 3.1	$l_k = TF_{t \rightarrow d}[t_k]$ ;
Step 3.2	foreach $\langle d_j, w_j \rangle \in l_k$ {
Step 3.2.1	$\text{score}[d_j] += w_k * w_j$ ;
Step 3.3	}
Step 4	}
Step 5	$C_{rank}[c_j] = 0$ foreach $c_j \in C$
Step 6	$D_k = \text{choose-k-document-with-high-score}(\text{score})$ ;
Step 7	foreach $d_j \in D_k$ {
Step 7.1	foreach $c_j \in C$ {
Step 7.1.1	$C_{rank}[c_j] ++$ if $d_j$ is classified to $c_j$
Step 7.2	}
Step 8	}
Step 9	$C_{rank} = \text{scaling}(C_{rank})$

Table 5: Step by step breakdown of fast kNN testing algorithm. Inputs:  $C[], TF_{t \rightarrow d}[], d, k$ . Outputs  $C_{rank}[]$ .

In the fast kNN algorithm of Table 5, the inputs and output are the same as those of kNN except on  $TF_{t \rightarrow d}$ . The kNN uses  $TF_{d \rightarrow t}$ , but the fast kNN uses  $TF_{t \rightarrow d}$ .  $TF_{d \rightarrow t}$  stores  $\langle term, weight \rangle$  for each document, while  $TF_{t \rightarrow d}$  stores the  $\langle document, weight \rangle$  for each term. The cost of the fast kNN can be calculated as follows: The cost of step 1 is the same as that of kNN, and its cost is  $c_1 * |d|$ . Step 2 initializes the array, where its size is the number of training documents, so the cost is  $c_2 * |D|$ . At step 3-4, the loop is repeated  $|d|$  times (the number of testing documents), so the cost will be  $|d| * X$ , where  $X$  is the total cost for 3.1, 3.2 and 3.3. Step 3.1 looks up  $t_k$  in  $TF_{t \rightarrow d}$  so its cost is  $O(1)$  if a hash is used. The steps 3.2 and 3.3 are repeated  $|TF_{t \rightarrow d}^{avg}|$  times,

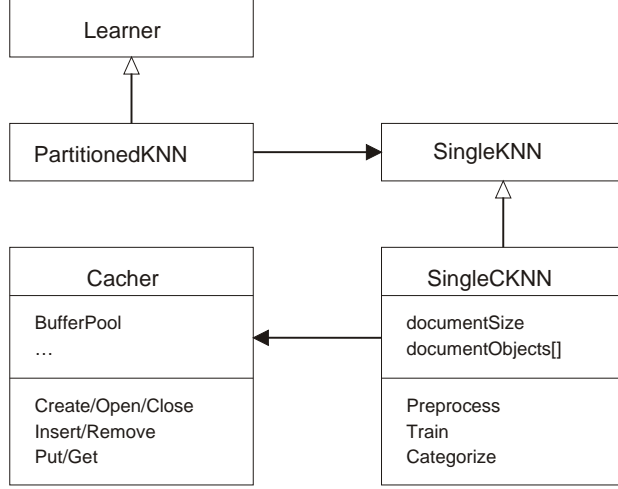


Figure 7: UML diagram for the kNN partitioning

where  $|TF_{t>d}^{avg}|$  is the average size of  $TF_{t>d}[t_k]$ . Thus, the cost of steps 3-4 is  $c_3 * |d| + c_4 * |TF_{t>d}^{avg}| * |d|$ . Step 5 is  $c_5 * |C|$ . At step 6, the function is required to select  $k$  documents with high score in  $|D|$  documents, so its cost is less than  $c_6 * k * |D|$ . The remaining costs are the same as those of kNN. Thus, the total cost becomes as follows:

$$T_{FastKNN} = (c_1 + c_3) * |d| + c_2 * |D| + c_4 * |TF_{t>d}^{avg}| * |d| + (c_5 + c_6 * k + c_7) * |C|$$

$|C| \ll |D|$  is always satisfied and  $k$  is usually a small value such between 5-20. Then,  $T_{FastKNN}$  can be simplified as follows:

$$T_{FastKNN} = O(|TF_{t>d}^{avg}| * |d|) \quad (5)$$

### 3.3.2 Partitioning for kNN

Using the partitioning schemes described earlier, the training set can be divided into  $n$  partitions. In kNN, when a new document arrives it is categorized  $n$  times, once for each partition. In each "pseudo-categorization" the closest  $k$  documents and their categories are selected. The results can be merged by performing kNN again on the  $k * n$  selected documents. This method can be applied with either of the two kNN methods described here.

Figure 7 shows a UML diagram for the implementation of the partitioned method within the document classification framework.

### 3.3.3 Buffering kNN

By partitioning the data, a SingleKNN class (and eventually the corresponding process) only has to classify a document on using a portion of the total training set. But unless the implementation uses a caching mechanism all data is processed in the main memory, or ends up being administered by the operating system. We have designed a caching mechanism that exploits our knowledge of the application by only having in memory in a time slice documents that are sufficiently similar.

We implemented a `bufferedkNN` class in the document classification framework and experimentally evaluated the idea.

## 4 Performance comparison and evaluation

In this section we analytically and experimentally compare the new Naïve Bayes and kNN methods with the original ones.

The analytical evaluation consists of the computational complexity analysis of each of the two methods discussed in section 2 and their modifications. In the experimental evaluation we measured the time improvements on ModApte, and on RCV1 whenever possible.

### 4.1 Analytical Evaluation

#### 4.1.1 Naïve Bayes

We will compare the simplified expressions for Naïve Bayes (equation 2) and for the modified FastNB (equation 4). The computational complexity as shown on those equations is  $O(|d| * |C|)$  and  $O(|d| * |TF_{t->c}^{avg}|)$ , respectively. Without loss of generality, the complexity of FastNB can be expressed in terms of the complexity of NB:

$$\begin{aligned} T_{FastNB} &= O(|d| * |TF_{t->c}^{avg}|) = O(|TF_{t->c}^{avg}|/|C| * |C| * |d|) \\ &= O(|TF_{t->c}^{avg}|/|C|) * T_{NB} \end{aligned}$$

where  $|TF_{t->c}^{avg}|$  is the average number of categories per term in the training documents and  $|C|$  is the number of categories. Thus,  $|TF_{t->c}^{avg}|$  is always less or equal than  $|C|$  which means that the proposed method always outperforms the conventional method at Naive Bayes.

#### 4.1.2 kNN

As in the previous section, we compare the simplified expressions for kNN (equation 3) and for the modified kNN (equation 5). For the same reason described earlier for Naive Bayes, we may compare the simplified equations (3) and (5) such as  $O(|TF_{d->t}| * |TF_{d->t}^{avg}|)$  and  $O(|TF_{t->d}^{avg}| * |d|)$  respectively. We represent  $T_{FastKNN}$  as the function of  $T_{KNN}$  and we note that the number of elements

of  $TF_{d \rightarrow t}$  in KNN is the same as those of elements of  $TF_{t \rightarrow d}$  in FastKNN, since  $TF_{t \rightarrow d}$  is the transposition of  $TF_{d \rightarrow t}$ . Then,  $|TF_{t \rightarrow d}| * |TF_{t \rightarrow d}^{avg}| = |TF_{d \rightarrow t}| * |TF_{d \rightarrow t}^{avg}|$  is always satisfied and  $|TF_{d \rightarrow t}|$  is the same to  $|D|$ . By using those,  $T_{FastKNN}$  can be represented as:

$$\begin{aligned} T_{FastKNN} &= O(|TF_{t \rightarrow d}^{avg}| * |d|) = O(|d| * |TF_{d \rightarrow t}| * |TF_{d \rightarrow t}^{avg}| / |TF_{t \rightarrow d}|) \\ &= O(|TF_{d \rightarrow t}^{avg}| / |TF_{t \rightarrow d}|) * T_{KNN} \end{aligned}$$

In the above equation,  $|TF_{d \rightarrow t}^{avg}|$  means the average number of terms per training document and  $|TF_{t \rightarrow d}|$  is the number of terms in all training documents, this means that  $|TF_{d \rightarrow t}^{avg}| \ll |TF_{t \rightarrow d}|$  is normally satisfied, which in turn, means that FastKNN normally outperforms KNN.

## 4.2 Experimental Evaluation

This section describes the evaluation of improvements achieved with the three improvements described earlier. The evaluations have been performed on the two Reuters sets described in section 2.3.

### 4.2.1 Reuters ModApte

Figure 8 compares the execution time for the traditional Naïve Bayes and our "fast" Naïve Bayes algorithm described in Section 3.2. Figure 9 shows the execution time for different sizes of data sets. The results shown in Figure 8 correspond to approximately 50% improvements in time efficiency.

Using the ModApte collection we first experimentally validated the linearity of kNN's cost, as shown in section 2.2. Figure 9 shows reductions of between 75% and 82% in execution time. The same Figure shows that the execution time of the two algorithms grows linearly, but the proposed method always outperforms the conventional kNN. These improvements would increase for larger data sets.

### 4.2.2 Reuters RCV1

The data was selected from the original RCV1 data set using a simple strategy where we randomly chose 80% of dates for the training set, and 20% for the test set. With this approach the total number of news items in each set was not an exact percentage since there were different numbers of stories on each day. This sampling strategy has the disadvantage of producing test sets that might have somewhat different statistics, for example a particular topic could be discussed only on one date or only on dates that are not included in the training set. In table 6, the notation XTr-YTe means that the files of X days are used as the training documents and files of Y days are used as testing documents. 10Tr, 50Tr and 100Tr means that we used 10, 50 and 100 days for training.

It is interesting to look at the accuracy of the kNN and NB methods on the RCV1 data set (D700K). Table 7 shows the micro and macro performance

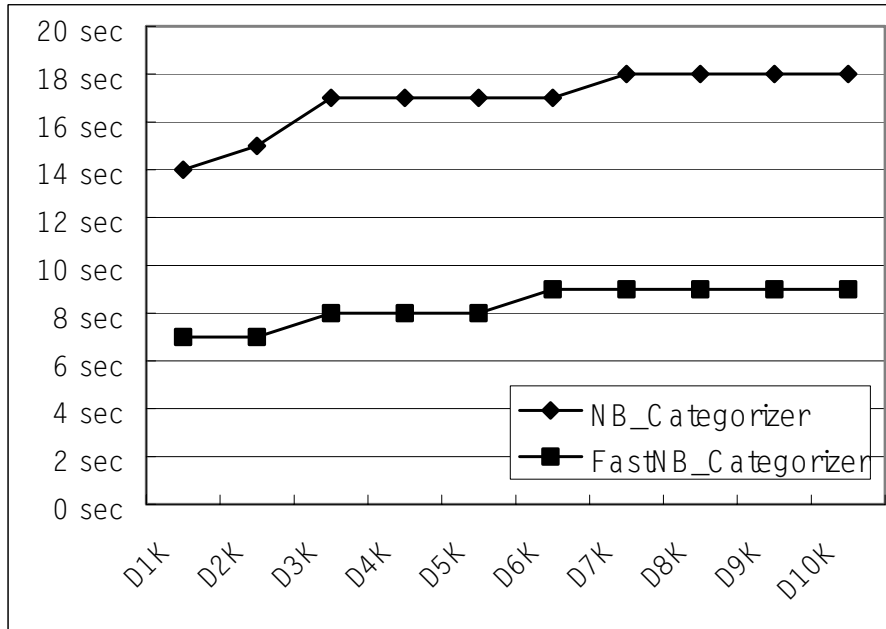


Figure 8: Execution time of NB and FastNB vs. number of training documents on ModApte set

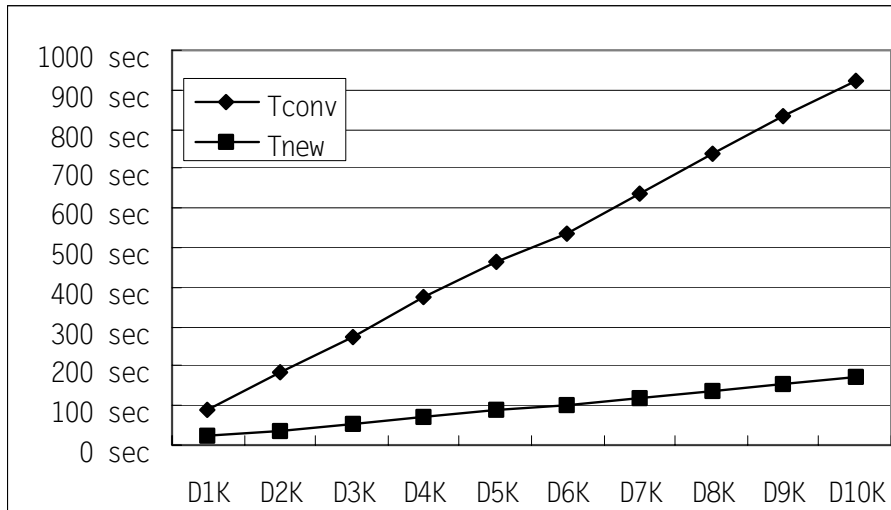


Figure 9: Execution time of kNN vs number of training documents on ModApte set

Data	Files	Bytes	Files	Bytes
10Tr-2Te	23,298	70Mb	5,872	18Mb
50Tr-10Te	107,543	331Mb	14,936	46Mb
100Tr-20Te	228,089	701Mb	43,008	133Mb

Table 6: Amount of training and testing data for the selected subsets of RCV1

measures when using 700,000 training documents. We can see that Naïve Bayes achieves considerably better accuracy than kNN.

Execution time comparisons on RCV1 were not possible because the original algorithms did not scale well enough to run on the hardware used.

Method	maR	maP	maF1	miR	miP	miF1
Naïve Bayes	0.327	0.860	0.430	0.540	0.950	0.690
kNN	0.456	0.657	0.487	0.260	0.773	0.389

Table 7: Accuracy measures for Naïve Bayes and kNN on D700K

## 5 Conclusions

We have shown the feasibility of large scale automatic document classification of news stories. We have focused on scalability issues and developed three improvements to the traditional Naïve Bayes and kNN algorithms. By changing the data representation we can improve the algorithms considerably and we have analytically shown how the new algorithms consistently outperform the original one. We have added data partitioning and memory buffering, and experimentally shown how these improvements make the methods faster and feasible for large data sets.

The classification performance shown with these experiments is extremely promising and would allow real web applications to use this type of functionality. We obtained 0.9 miP, meaning that 9 out of 10 stories were correctly classified. The miR and therefore the miF1 were somewhat lower because some classes contained few examples and were harder to classify. Naïve Bayes classifiers produced lower quality classification but seemed to be better suited for applications where classification needed to be performed in real time, unlike kNN which produced better classification but placed too much load on classification time since there was no training. Since we wanted to assess the possibility of using automatic classification techniques for real web applications, we chose a sampling strategy that produced lower classification performance, but was more scalable and therefore realistic. We extended our classification framework in order to manage large volumes of documents. The object buffer was successful in increasing the number of documents that we were able to manage by an order

of magnitude and reduced the computational time to approximately one fifth.

Using faster XML parsing for the pre-processing stage proved to be important in managing large amounts of newsML documents. This shows that very efficient parsers are required for content syndicators who manage large quantities of news stories. This project will continue by further studying the scalability issues of other algorithms and integrating the classifier into real web applications. This will be done following the integration of the framework to the Postgres database, as described in [13], and using it in web application frameworks as described in [12].

In order to optimise memory usage, the buffered object manager uses a constant amount of memory. Our current implementation follows the Least Recently Used (LRU) strategy. In this strategy, common in paging in operating systems, an object (or page) is selected to be taken out of the buffer (paged out) if it has been used (read or written) less recently than any other page. The same rule is also used in other cache systems to select which cache entry to flush. In future work we plan to add new buffering strategies such as: Most Recently Used (MRU) and others. We expect that the selection of these strategies will depend on each machine learning algorithm. For example, Naïve Bayes might work better with a LRU and kNN with a MRU strategy. With a scalable framework we are able to show the feasibility of automatically classifying large amounts of news stories. Since news stories in the RCV1 dataset are in newsML format the extensions to the framework need to include more efficient XML parsing. We first tried the Perl and C implementations of the SAX package. Adapting one of the libraries implemented in C and integrating it into the framework produced the best results.

## Acknowledgements

The authors are thankful to Ken Williams for his contribution to the project, and to the anonymous reviewers who helped with the paper. The authors acknowledge the support of the Australian Research Council through an ARC Linkage International grant, the University of Sydney through a Sesqui Grant, the Australian Academy of Science, Korean Science and Engineering Foundation and Hansung University.

## References

- [1] Rafael A. Calvo and H. A. Ceccatto. Intelligent document classification. *Intelligent Data Analysis*, 4(5), 2000.
- [2] Mohamed Fayad and Douglas C. Schmidt, editors. *Building Application Frameworks*. John Wiley & Sons, 1999.
- [3] Steve Lawrence and C. Lee Giles. Searching the web: General and scientific information access. *IEEE Communications*, 37(1): 116-122, 1999.

- [4] David D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. In Claire Nédellec and Céline Rouveirol, editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398, pages 4–15, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.
- [5] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.
- [6] J. Mayfield, P. McNamee, C. Costello, C. Piatko, and A. Banerjee. Jhu/apl at trec 2001: Experiments in filtering and in arabic, video, and web retrieval. In M. Voorhees and D. K. Harman, editors, *Proceedings of the Tenth Text Retrieval Conference (TREC 2001)*. NIST Special Publication, 2002.
- [7] Tony G. Rose, Mark Stevenson, and Miles Whitehead. The reuters corpus volume 1 - from yesterday's news to tomorrow's language resources. In *3rd International Conference on Language Resources and Evaluation*, page 7, May 2002.
- [8] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys (CSUR)*, 34(1):1–47, 2002.
- [9] Ken Williams and Rafael A. Calvo. A framework for text categorization. In The University of Sydney, editor, *7th Australasian Document Computing Symposium*, Sydney, Australia, 2002.
- [10] Ken Williams, Rafael A. Calvo, and David Bell. Automatic categorization of questions for a mathematics education service. In *Proceedings of the 11th International Conference on Artificial Intelligence in Education*, August 2003.
- [11] Y. Yang and C.G. Chute. An example based mapping method for text categorization and retrieval. *ACM Transactions in Information Systems*, 12(3):252–277, 1994.
- [12] Yiming Yang and X. Liu. A re-examination of text categorization methods. In *22nd Annual International SIGIR*, pages 42–49, Berkley, August 1999.