CLUSTERING UNSTRUCTURED TEXT DOCUMENTS USING NAÏVE BAYESIAN CONCEPT AND SHAPE PATTERN MATCHING

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

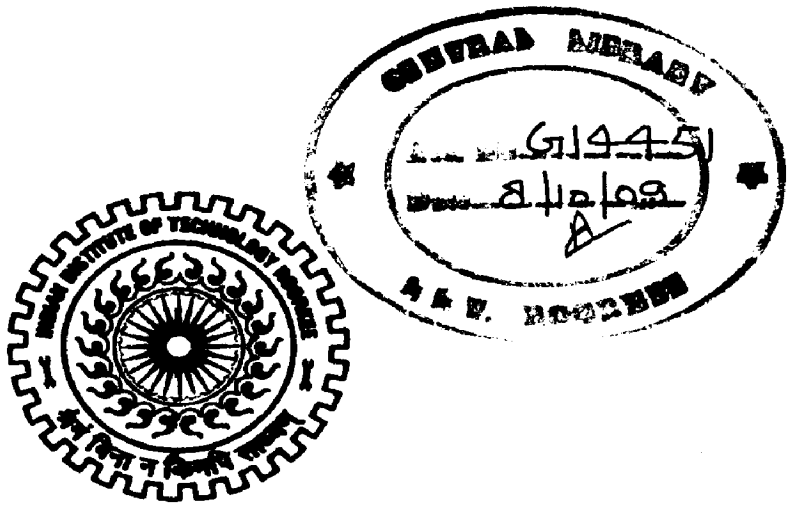
MASTER OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

By

RISHIRAJ SAHA ROY



DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)

JUNE, 2009

ACKNOWLEDGEMENTS

First of all, I would like to thank **Dr. Durga Toshniwal**, my supervisor, for getting me interested in the area of data mining, and her consistent encouragement of my ideas and work. I am particularly grateful for her enthusiasm and constant support. This dissertation would not have been a reality without her insightful advice. Working under her guidance will always remain a cherished experience in my memory.

I am also thankful to the Indian Institute of Technology Roorkee for giving me this opportunity. I extend my sincere thanks to **Dr. S. N. Sinha**, Professor and Head of the Department of Electronics and Computer Engineering. I am also grateful to the research scholars and the staff of our department, for their kind cooperation. I thank all my friends who have helped me directly or indirectly in completing this dissertation.

Most importantly, I would like to extend my deepest appreciation to my family for their love, encouragement and moral support. Finally I thank God for being kind to me and driving me through this journey.

Rishiraj Saha Roy

Rishiraj Saha Roy

M.Tech. (I.T.)

ABSTRACT

Text document databases are growing rapidly due to the increasing amount of information available in electronic form, such as research publications, news articles, books, and e-mails. Most text databases are semi-structured in that they are neither fully unstructured nor completely structured. Clustering is performed to organize this text data in an unsupervised fashion. It also acts as a preprocessing step for further mining operations like indexing and classification. Time series data mining involves applying mining techniques to time sequences. Much work has been done in this field in the past few decades. But the idea of applying time series data mining techniques on text data mapped to sequences has not yet been explored. We intend to address this problem in this work.

In this dissertation, an algorithm for clustering unstructured text documents using naïve Bayesian concept and shape pattern matching has been proposed. The first step involves data preprocessing. This includes stop word removal, word stemming, and dimensionality reduction using locality preserving indexing scheme. In the proposed work, we use the Vector Space Model to represent our dataset as a term weight matrix. In any natural language, semantically linked terms tend to occur together in documents. Based on this observation, the co-occurrences of pairs of terms in the term weight matrix are observed. This information is then used to build an initial term cluster matrix where each term may belong to one or more clusters. The naïve Bayesian concept and cluster conditional independence is used to uniquely assign each term to a single term-cluster. The text documents are assigned to clusters using the simple statistical measure of arithmetic mean. This completes the first level of clustering in our proposed algorithm. Mapping text documents to vectors based on a list of terms converts them to sequences. Shape pattern-based similarity is a well-established technique in time series data mining. In this work, we apply shape pattern matching to group documents within the broad clusters obtained earlier, thus performing a second level of clustering.

The proposed algorithm has been validated using benchmark datasets available on the internet. This includes two special datasets ADA (a marketing application) and SYLVA (an ecology application). Our results show that our proposed two-level text clustering scheme has a significantly better running time as compared to traditional algorithms.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
INTRODUCTION

In the last few years, there has been an explosion in the amount of data available in electronic form. This includes transactional data in bank databases, electronic news articles, satellite images of the earth, streaming video data from surveillance cameras, and all the data available on the World Wide Web. There is an imminent need for turning such data into useful information. The knowledge gained from this process can be used in a wide range of applications ranging from market analysis, fraud detection, and scientific discovery. Data mining or Knowledge Discovery from Data (KDD) refers to extracting or "mining" knowledge from large amounts of data [1]. This kind of knowledge is usually shown in the form of definitions, rules, patterns, etc. Data mining today has indeed found a diverse field of applications like helping in recent discoveries in biomedical science, predicting weather and climatic changes, analyzing security scenarios in public places, and organizing thousands of documents in digital libraries.

Alternatively, data mining is often viewed as a step in knowledge discovery. Knowledge discovery as a process is shown in Figure 1.1 and consists of the following steps [1]:

1. **Data cleaning** (for removing noise and inconsistent data)
2. **Data integration** (for combining data from multiple sources)
3. **Data selection** (for retrieving data relevant to our task from the entire database)
4. **Data transformation** (for transforming or consolidating the data into forms appropriate for mining)
5. **Data mining** (the process where intelligent methods are applied in order to extract patterns hidden in the data)
6. **Pattern evaluation** (for identifying the truly interesting patterns representing knowledge based on some interestingness measures)
7. **Knowledge presentation** (where visualization and knowledge representation techniques are used to present the mined knowledge to the user)

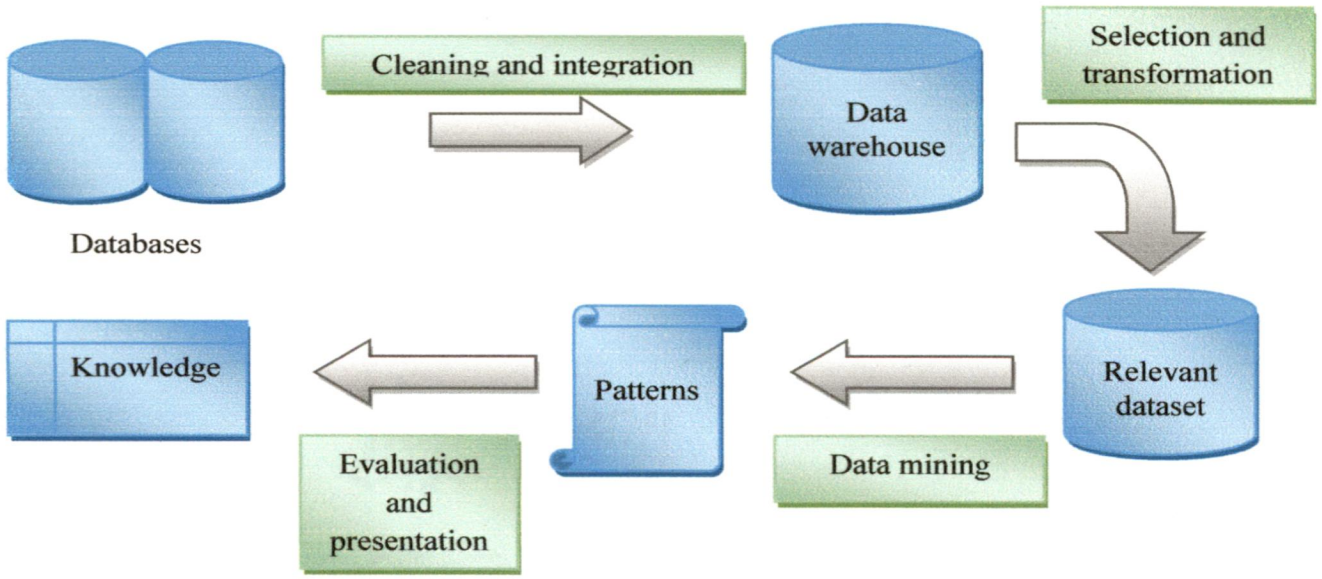


Figure 1.1 Data mining as a step in the process of knowledge discovery

Complex forms of data (like hypertext and multimedia, semi-structured and unstructured text data, spatial and temporal data) have grown explosively due to the rapid progress in advanced database system technologies. As a result, sophisticated data mining applications which are able to mine interesting patterns within these complex data forms are the need of the hour. In this work, we focus on mining unstructured text data, explained in Section 1.1.

Data mining commonly involves three classes of task, out of which we focus on the first in this work:

1. **Clustering** - Groups similar data items together in an unsupervised manner as the group labels are not known. For example, a clustering algorithm may form twelve clusters from an input of five hundred news articles such that data within one cluster are similar to one another and different from those of other clusters. Common algorithms include *k*-means and Expectation Maximization (EM).
2. **Classification** - Arranges the data into predefined groups. For example, an email program classifies an email as legitimate or spam. Common algorithms include the *k*-Nearest Neighbor classifier (kNN) and the naïve Bayesian classifier.
3. **Association rule mining** - Searches for relationships between variables. For example, a computer store might gather data of what each customer buys and find that seventy-five percent of the customers buying desktop computers also buy printers.

Common algorithms include the Apriori algorithm and the Frequent Pattern (FP)-growth.

1.1 Text Mining

Nowadays, a substantial portion of the available information is stored in text document databases, which consist of large collections of documents from various sources, such as news articles, research papers, books, digital libraries, and e-mail messages [1]. Text databases are rapidly growing due to the increasing amount of information available in electronic form, such as electronic publications, various kinds of electronic documents, e-mail, and the World Wide Web (which can also be viewed as a huge, interconnected, dynamic text database). Nowadays most of the information in government, industry, business, and other institutions are stored electronically, in the form of text databases. Data stored in most text databases are semistructured data in that they are neither completely unstructured nor completely structured. For example, a document may contain a few structured fields, such as title, authors, and publication date, but it also contains some largely unstructured components, such as the abstract and the contents. There have been a great deal of studies on the modelling and implementation of semi-structured and unstructured data in recent database research. Moreover, information retrieval techniques, such as text indexing methods, have been developed to handle unstructured documents.

Traditional information retrieval techniques become inadequate for the increasingly vast amounts of text data. Typically, only a small fraction of the many available documents will be relevant to a given individual user. Without knowing what could be in the documents, it is difficult to formulate effective queries for analyzing and extracting useful information from the data. Users need tools to compare different documents, rank the importance and relevance of the documents, or find patterns and trends across multiple documents. Thus, text mining has become an increasingly popular and essential part of data mining. There are several aspects of mining text databases. The key ones include classification, clustering, and association rule mining. We focus on text clustering in this work.

1.2 Time Series Data Mining

Time series data consists of sequences of values or events obtained over repeated measurements of time. With the growing deployment of large numbers of sensors, telemetry devices, and on-line data collection tools, the amount of time series data is increasing rapidly. The values are typically measured at equal time intervals. Databases storing time series data are called time series databases. A time series database is also a sequence database [1]. But a sequence database is any database that consists of sequences of ordered events, with or without concrete notions of time.

The quest for finding correlation relationships within the data and the need for analysis of huge numbers of time series to find similar or regular patterns, trends, bursts (sudden sharp changes), and outliers, with fast or real-time on-line response leads us to perform various data mining operations on time series databases. Performing these functions on time series databases is referred to as time series data mining. There are several aspects of mining time series databases. The key ones include similarity search, trend analysis, mining periodic patterns, classification, clustering, and association rule mining. Techniques involving shape pattern-based similarity have been highly successful in the field of time series data mining (refer to Section 2.3). Since we are using the sequence representation of text documents in our work, and time series data are in essence sequences, exploring similar applications of such techniques in the field of text mining was a worthwhile effort. This required a basic understanding of time series data mining.

1.3 Problem Statement

The problem statement for this dissertation is stated as follows:

"Clustering unstructured text documents using naïve Bayesian concept and shape pattern based similarity.

This problem can be broken down into the following smaller sub-problems:

- *To discover term-clusters from the total term set using naïve Bayesian theory*

- *To find document clusters from the entire unstructured text document set on the basis of these term-clusters*
- *To detect sub-clusters within the document clusters by making use of the shape pattern matching technique*

Assumption: The terms are arranged in an inherent sequence which remains fixed throughout the model."

1.4 Organization of Dissertation

This report is organized as follows. It comprises of a total of six chapters including this chapter. This is preceded by the candidate's declaration, the certificate, acknowledgements, the abstract, the table of contents, and the list of figures and tables. The six chapters are followed by the references used for this work, the list of publications, and two appendices which contain the source code listing and a list of common stopwords in English.

In **Chapter 1**, we give an introduction to data mining. We then briefly discuss text mining and time series data mining. In the end, we give our problem statement and the organization of this dissertation.

In **Chapter 2**, we discuss about the literature review performed before doing this work. We also give the research gaps thus found and the motivation for our work.

In **Chapter 3**, we focus on text preprocessing, a sequence of operations which is necessary before any mining task can be performed efficiently on a set of text data.

In **Chapter 4**, we describe the proposed design for the complete clustering scheme in detail.

In **Chapter 5**, we give our experimental results and the relevant discussion.

In **Chapter 6**, we conclude this report by giving the conclusions drawn from the obtained results and the suggestions for future work.

Chapter 2
LITERATURE REVIEW

In this chapter, we discuss about the literature review performed as groundwork for this dissertation.

2.1 Text Clustering

Clustering is performed to organize text documents in an unsupervised manner. When text documents are represented in the form of vectors (refer to Section 3.4), common clustering methods that employ the concepts of distances, hierarchies, and densities among data objects can be applied. But the vector space almost always has a very large number of dimensions, due to the great number of terms present. A projection of the documents into a lower dimensional subspace brings the semantic structure of the document to light. After the operations of dimension reduction have been performed, traditional clustering algorithms can be applied to obtain meaningful results efficiently. This curse of dimensionality poses a tough challenge for clustering and other text mining operations. Now we describe a few recent text clustering approaches.

Feature selection is an important method for improving the efficiency and accuracy of text clustering algorithms by removing redundant and irrelevant terms from the corpus. A supervised feature selection method, named CHIR [2], has been proposed which is based on the $\chi 2$ statistic and new statistical data that can measure the positive term-cluster dependency. A new text clustering algorithm named TCFS has been proposed, which stands for Text Clustering with Feature Selection. TCFS can incorporate CHIR to identify relevant features (i.e. terms) iteratively, and the clustering becomes a learning process. TCFS and the *k*-means clustering algorithm [3] have been compared in combination with different feature selection methods for various real data sets. Experimental results showed that TCFS with CHIR had a better clustering accuracy in terms of the *F*-measure.

An approach to text clustering has been proposed in [4] which combines the advantages of the *k*-means algorithm [3] and the Self-Organizing Map (SOM) [5] techniques. The experimental results indicate that the improved algorithm has a higher accuracy and a better stability, compared with the original algorithm.

A text-clustering algorithm of Frequent Term Set-based Clustering (FTSC), which uses frequent term sets for texts clustering, has been proposed [6]. This algorithm can reduce the dimensionality of the text data (refer to Section 3.3) efficiently. Thus it can improve the accuracy rate and running speed of the clustering algorithm. The results of clustering text by the FTSC algorithm cannot reflect the overlap of texts' classes. Based on the FTSC algorithm, its improved form, the Frequent Term Set-based Hierarchical Clustering algorithm (FTSHC) has also been proposed. This algorithm can determine the overlap of texts' classes by the overlap of frequent term-sets, and provide an understandable description of the discovered clusters by the frequent term-sets. The experiment results proved that the FTSC and the FTSHC algorithms are more efficient than the *k*-means algorithm [3] in clustering performance.

2.2 Naïve Bayesian Classifiers

Now we will discuss about the application of the naïve Bayesian concept in classification. We will explain it in detail as it is one of the fundamental concepts used in our algorithm. Bayesian classifiers are statistical classifiers [7, 8]. They can predict a class membership probability, i.e. the probability that a given tuple belongs to a particular class. Bayesian classification is based on Bayes' theorem, described below. Studies comparing classification algorithms have found a simple Bayesian classifier known as the naïve Bayesian classifier to be comparable in performance with popular classifiers. Bayesian classifiers have also exhibited high accuracy and speed when applied to large databases.

Naïve Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called class conditional independence. It is made to simplify the computations involved and, in this sense, is considered "naïve" [1]. Let $\boldsymbol{X}$ be a data tuple. In Bayesian terms, $\boldsymbol{X}$ is considered "evidence." As usual, it is described by

measurements made on a set of n attributes. Let H be some hypothesis, such as that the data tuple $\boldsymbol{X}$ belongs to a specified class C. For classification problems, we want to determine $P(H|\boldsymbol{X})$, the probability that the hypothesis H holds given the "evidence" or observed data tuple $\boldsymbol{X}$. In other words, we are looking for the probability that tuple $\boldsymbol{X}$ belongs to class C, given that we know the attribute description of $\boldsymbol{X}$.

$P(H|\boldsymbol{X})$ is the posterior probability, or *a posteriori* probability, of H conditioned on $\boldsymbol{X}$. In contrast, $P(H)$ is the prior probability, or a priori probability, of H. The posterior probability, $P(H|\boldsymbol{X})$, is based on more information than the prior probability, $P(H)$, which is independent of $\boldsymbol{X}$. Similarly, $P(\boldsymbol{X}|H)$ is the posterior probability of $\boldsymbol{X}$ conditioned on H. $P(\boldsymbol{X})$ is the prior probability of $\boldsymbol{X}$. $P(H)$, $P(\boldsymbol{X}|H)$, and $P(\boldsymbol{X})$ may be estimated from the given data, as we shall see below. Bayes' theorem is useful in that it provides a way of calculating the posterior probability, $P(H|\boldsymbol{X})$, from $P(H)$, $P(\boldsymbol{X}|H)$, and $P(\boldsymbol{X})$.
Bayes' theorem is given as:

$$P(H|\boldsymbol{X}) = (P(\boldsymbol{X}|H) \times P(H)) / P(\boldsymbol{X}) \qquad (2.1)$$

The naïve Bayesian classifier, or simple Bayesian classifier, works as follows:
1. Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n attributes, respectively, $A_1, A_2, \ldots, A_n$.

2. We suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, $\boldsymbol{X}$, the classifier will predict that $\boldsymbol{X}$ belongs to the class having the highest posterior probability, conditioned on $\boldsymbol{X}$, i.e. the naïve Bayesian classifier predicts that the tuple $\boldsymbol{X}$ belongs to the class C_i if and only if

$$P(C_i|\boldsymbol{X}) > P(C_j|\boldsymbol{X}) \text{ for } 1 \leq j \leq m, j \neq i.$$

Thus we maximize $P(C_i|\boldsymbol{X})$. The class C_i for which $P(C_i|\boldsymbol{X})$ is maximized is called the maximum posteriori hypothesis. By Bayes' theorem (Equation (2.1)),

$$P(C_i|\boldsymbol{X}) = (P(\boldsymbol{X}|C_i) \times P(C_i)) / P(\boldsymbol{X}) \qquad (2.2)$$

3. As $P(X)$ is constant for all classes, only $(P(X|Ci) \times P(Ci))$ needs be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1) = P(C_2) = \ldots = P(C_m)$, and we would therefore maximize $P(X|C_i)$. Otherwise, we maximize $(P(X|C_i) \times P(C_i))$. We note that the class prior probabilities may be estimated by $P(C_i) = |C_{i,D}| / |D|$, where $|D|$ is the total number of training tuples, and $|C_{i,D}|$ is the number of training tuples of class C_i in D.

4. Given datasets with many attributes, it would be extremely computationally expensive to compute $P(X|C_i)$. In order to reduce computation in evaluating $P(X|C_i)$, the naïve assumption of class conditional independence is made. This presumes that the values of the attributes are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

$$P(\boldsymbol{X}|C_i) = \prod_{k=1}^{n} P(x_k|C_i) \tag{2.3}$$

$$P(X|C_i) = P(x_1|C_i) \times P(x_2|C_i) \times \ldots \times P(x_n|C_i) \tag{2.4}$$

We can easily estimate the probabilities $P(x_1|C_i)$, $P(x_2|C_i)$, ... , $P(x_n|C_i)$ from the training tuples. Recall that here x_k refers to the value of attribute A_k for tuple $\boldsymbol{X}$. For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute $P(X|C_i)$, we consider the following:

(a) If A_k is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D having the value x_k for A_k, divided by $|C_{i,D}|$.

(b) If A_k is continuous-valued, then we need to perform the following calculations. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean μ and standard deviation s, defined by

$$g(x, \mu, \sigma) = \frac{1}{(\sqrt{2\pi}\sigma)} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.5}$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{Ci}, \sigma_{Ci}) \tag{2.6}$$

We need to compute μ_{Ci} and σ_{Ci}, which are the arithmetic mean and standard deviation, respectively, of the values of attribute A_k for training tuples of class C_i. We then plug these two quantities into Equation (2.5), together with x_k, in order to estimate $P(x_k|C_i)$.

5. In order to predict the class label of $\boldsymbol{X}$, $(P(\boldsymbol{X}|Ci) \times P(Ci))$ is evaluated for each class C_i. The classifier predicts that the class label of tuple $\boldsymbol{X}$ is the class C_i if and only if

$$(P(\boldsymbol{X}|C_i) \times P(C_i)) > (P(\boldsymbol{X}|C_j) \times P(Cj)) \qquad \text{for } 1 \leq j \leq m, j \neq i. \qquad (2.7)$$

In other words, the predicted class label is the class C_i for which $(P(\boldsymbol{X}|C_i) \times P(C_i))$ is the maximum.

There is another modification to be introduced. In the product of Equation (2.3), if any of the $P(x_k|C_i)$ is zero, it makes the whole product zero. But without the zero probability, we may have ended up with a high probability, suggesting that $\boldsymbol{X}$ may have belonged to class C_i. A zero probability cancels the effects of all of the other posteriori probabilities (on C_i) involved in the product. This problem is avoided as follows. We can assume that our training database, D, is so large that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the case of probability values of zero. This technique for probability estimation is known as the Laplacian correction.

2.3 Shape Patterns in Time Series Data Mining

Time series data mining (TSDM) techniques permit exploring large amounts of time series data in search of consistent patterns and / or interesting relationships between variables. TSDM is becoming increasingly important as a knowledge management tool where it is expected to reveal knowledge structures that can guide decision-making in conditions of limited certainty. The necessity of extraction of meaningful information from huge time series databases (TSDB), which can be useful for decision making, caused the development of the methods of time series data

mining. In this section, we look at a few applications of the concept of shape pattern-based similarity in the field of TSDM.

Human decision-making in problems related with the analysis of time series databases is usually based on perceptions like "end of the day", "high temperature", "quickly increasing", "possible", etc. Though many effective algorithms of TSDM have been developed, the integration of TSDM algorithms with human decision making procedures is still an open problem. An architecture of a perception-based decision making system in a time series database domain has been proposed in [9] which integrates perception-based TSDM, computing with words and perceptions, and expert knowledge. The new tasks which should be solved by the perception-based TSDM methods to enable their integration in such systems have also been discussed. These tasks include the precisiation of perceptions, shape pattern identification, and pattern retranslation.

Clipping is the process of transforming a real valued series into a sequence of bits representing whether each data is above or below the average [10]. It has been demonstrated how time series stored as bits can be very efficiently compressed and manipulated and that, under some assumptions, the discriminatory power with clipped series is asymptotically equivalent to that achieved with the raw data. Unlike other transformations, clipped series can be compared directly to the raw data series. It has been shown that this means we can form a tight lower bounding metric for Euclidean and dynamic time warping distance and hence efficiently query by content. Clipped data can be used in conjunction with a host of algorithms and statistical tests that naturally follow from the binary nature of the data. Shape pattern-based similarity is one of the basic concepts used in this work.

An algorithm has been proposed which applies a linguistic variable concept tree to describe the slope feather of time series, and has been named Shape Dynamic Time Warping [11]. For reducing the computational time and the local shape variance disturbance, the piecewise linear representation has been used to preprocess the warping path. Moreover, the linguist concept tree was developed based on the theory of cloud models which integrates randomness and the probability of uncertainty.

2.4 Research Gaps Found

- Traditional text clustering algorithms attempt to find clusters among the documents directly, based on term weight vectors. So they have to deal with vectors of a very high dimensionality. Very few attempts were made to first cluster the terms on the basis of semantic correlation and then cluster the documents based on these term-clusters.

- The naïve Bayesian theory had been applied only to classifiers.

- Shape pattern-based similarity, a highly successful technique in time series data mining, had not yet been explored in the mining of text data, even though representing text documents as sequences has long been in practice.

Chapter 3

TEXT PREPROCESSING

In this chapter, we will explain how text data from raw files are prepared before any mining operation is performed on it. The process consists of a sequence of operations which include removal of stopwords, word stemming, and dimensionality reduction. Selecting a feature subset to represent the text and clustering on it is an effective method to minimize the problem posed by the curse of dimensionality [12]. In dimensionality reduction, we will mainly focus on Locality Preserving Indexing (LPI) [13], which gives the best results when the ultimate goal is clustering. Text preprocessing paving the way for the main mining operations is shown as a block diagram in Figure 3.1. In Section 3.4, we explain the Vector Space Model [14] which is used to mathematically represent the text documents. In Section 3.5, we explain the sparse matrix representation [15] which is used extensively in text mining for making memory and disk space utilization efficient.

3.1 Stopword Removal

For representing documents, the first step in most retrieval systems is to identify keywords by morphological analysis [16], a preprocessing step often called tokenization. To avoid indexing useless words called stopwords, a text retrieval system often associates a stop list with a set of documents [1]. A stop list is a set of words that are deemed "irrelevant." For example, *a*, *the*, *of*, *for*, *with*, and so on are stopwords, even though they may appear frequently. Stop lists may vary per document set. For example, *cricket* could be considered an important keyword while clustering a set of random newspaper articles. However, it may be considered as a stopword in a set of articles about a cricket tournament. A list of the most common stop words in English [17] are given in Appendix B.

3.2 Word Stemming

A group of different words may share the same word stem. A text retrieval system needs to identify groups of words where the words in a group are small syntactic variants of one another and collect only the common word stem per group.

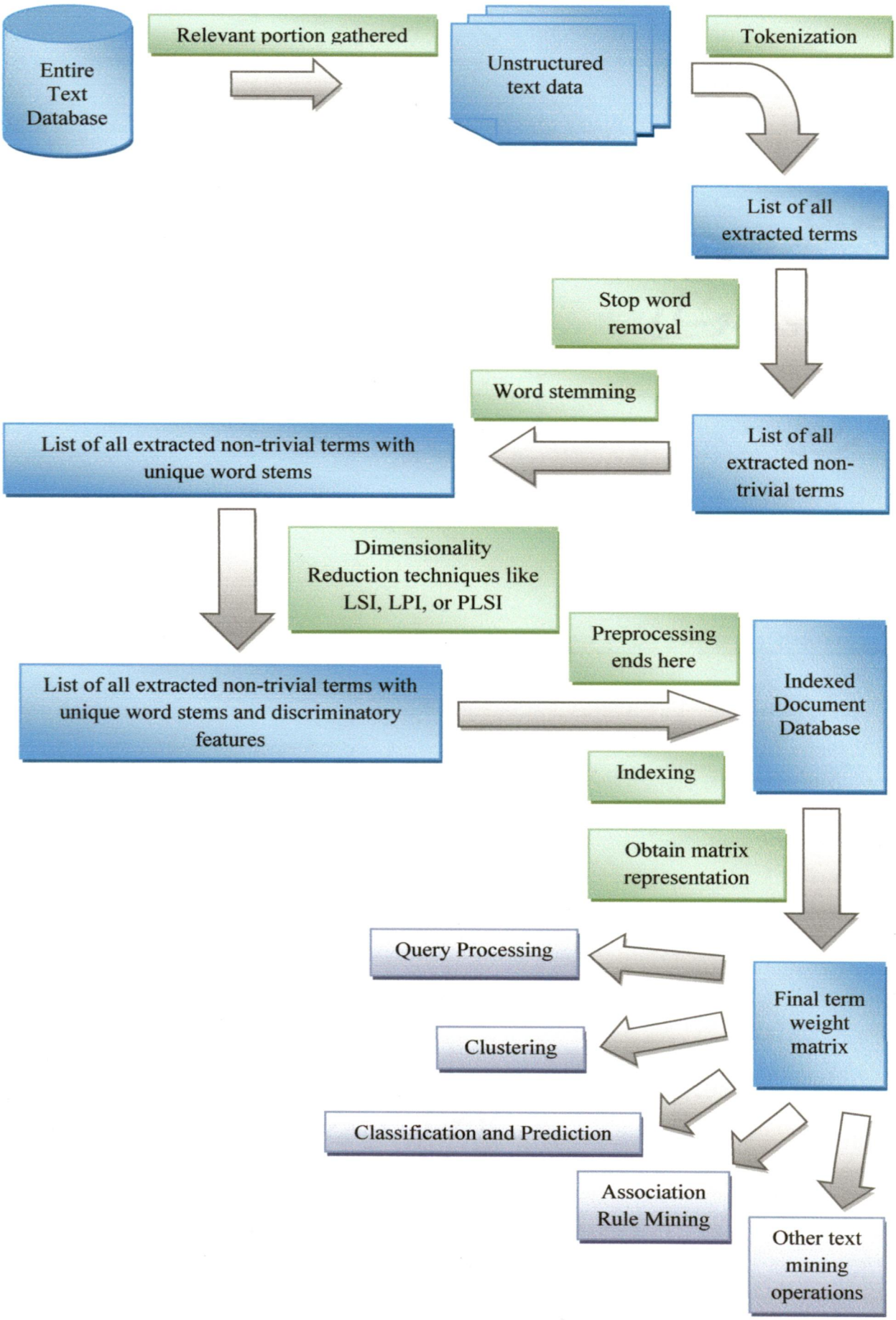


Fig. 3.1 Preprocessing paving the way for text mining operations

For example, the group of words *bowl, bowled*, and *bowling*, share a common word stem, *bowling*, and can be viewed as different occurrences of the same word. The word stemming algorithm identifies words with a common stem and replaces all words sharing a common word stem with the word stem itself. The most famous of these is the Porter's Stemming Algorithm [18], a part of which is presented next.

The Porter's Stemming Algorithm is based upon a set of conditions of the stem, suffix, prefix, and associated actions given the condition. The measure, *m*, of a stem is a function of sequences of vowels and *y* (*a, e, i, o, u,* and *y*) followed by a consonant. If *V* is a sequence of vowels and *C* is a sequence of consonants, then *m* is:

$$[C](VC)\{m\}[V]$$

where the initial *C* and final *V* are optional and *(VC){m}* denotes *VC* repeated *m* times.

Table 3.1 Examples of measures and corresponding terms

SL. NO.	MEASURE *m*	EXAMPLES
1	0	FREE, TREE, BY, WHY
2	1	TROUBLE, TREES, FREES, IVY, WHOSE
3	2	PROLOGUE, COMPUTE, PRIVATE

Some stem conditions are as follows:

1. ***<X>*** : Stem ends with letter *X*
2. ****v**** : Stem contains one vowel
3. ****d*** : Stem ends in double consonant
4. ****o*** : Stem ends with consonant-vowel-consonant sequence where the final consonant is not *w, x* or *y*

Suffix conditions take the form : Current suffix = Pattern

Actions take the form : Old suffix → New suffix

Table 3.2 Examples of rules from Porter's Stemming Algorithm

RULE	CONDITION	SUFFIX	REPLACEMENT	EXAMPLE
1a	NULL	ssess	ss	stresses → stress
1b	*v*	ing	NULL	bringing → bring
1c	*v*	y	i	happy → happi
2a	m>0	icate	ic	duplicate → duplic
2b	m>0	aliti	al	formality → formal
3	NULL	at	ate	inflat → inflate
4	m>1	able	NULL	adjustable → adjust
5	m>1 and *d and *<*L*>	NULL	single letter	controll → control

Rules are divided into steps to define the order of applying the rules. The following is an example. Given the word "duplicatable", the following are the steps in the stemming process:

Step 1: duplicatable → duplicat (By Rule 4)
Step 2: duplicat → duplicate (By Rule 3)
Step 3: duplicate → duplic (By Rule 2)

We note that only one rule from each step can be applied. Steps have to be chosen in descending order.

3.3 Dimensionality Reduction

Due to the presence of the huge number of terms in the initial data set, the mining operations are faced with the curse of dimensionality. As a result, specialized dimensionality reduction techniques have been developed for text data. When these are applied, the documents are projected onto a lower dimensional subspace in which the semantic structure of the document space becomes clear. In the low-dimensional semantic space, clustering algorithms can be applied effectively. So these dimensionality reduction techniques yield a reduced set of terms which helps us proceed with our mining task in a more efficient manner. The most popular of these

techniques are locality preserving indexing (LPI) [13], latent semantic indexing (LSI) [19], and probabilistic latent semantic analysis (PLSA) [20]. LPI aims to discover the local geometrical structure of the document space. Since the neighboring documents (data points in high dimensional space) probably relate to the same topic, LPI can have a high discriminating power. Therefore, for document clustering and document classification, we might expect LPI to have a better performance than LSI and PLSA. This has been confirmed empirically [13]. So we now explain LPI briefly.

We use $x_1, \ldots, x_n \in \mathbf{R}^m$ to represent the n documents with m terms. They can be represented as a term-document matrix $X = [x_1, x_2, \ldots, x_n]$. The basic idea of LPI is to preserve the locality information (i.e. if two documents are near each other in the original document space, LPI tries to keep these two documents close together in the reduced dimensionality space). Since the neighboring documents (data points in high-dimensional space) probably relate to the same topic, LPI is able to map the documents related to the same semantics as close to each other as possible. Given $x_1, \ldots, x_n \in \mathbf{R}^m$, LPI constructs a similarity matrix $S \in \mathbf{R}^{n \times n}$. The transformation vectors of LPI can be obtained by solving the following minimization problem:

$$a_{opt} = \arg\ min_a \sum_{i,j} (a^T x_i - a^T x_j)^2 S_{ij} = \arg min_a\ a^T XLX^T a \qquad (3.1)$$

with the constraint,

$$a^T XDX^T a = 1 \qquad (3.2)$$

where $L = D - S$ is the *Graph Laplacian* and $D_{ii} = \Sigma_j S_{ij}$. D_{ii} measures the local density around x_i. LPI constructs the similarity matrix S as

$$\begin{aligned} S_{ij} &= (x_i^T x_j / \| x_i^T x_j \|) && \text{if } x_i \text{ is among the } p \text{ nearest neighbors of } x_j, \\ & && \text{or if } x_j \text{ is among the } p \text{ nearest neighbors of } x_i \\ &= 0 && \text{otherwise} \end{aligned} \qquad (3.3)$$

Here p is a user input. Thus, the objective function in LPI incurs a heavy penalty if neighboring points x_i and x_j are mapped far apart. Therefore, minimizing it

is an attempt to ensure that if x_i and x_j are "close" then y_i (= $a^T x_i$) and y_j (= $a^T x_j$) are close as well. Finally, the basis functions of LPI are the eigenvectors associated with the smallest eigenvalues of the following generalized eigen-problem:

$$XLX^T a = \lambda XDX^T a \tag{3.4}$$

3.4 Vector Space Model

Proposals of many models have been made for dealing with text mining problems. One of them is the Vector Space Model [14, 21], the use of which has been made in this work. We briefly explain it in this section. Let there be m documents and n terms in all. Then each document can be modelled as a vector v in an n-dimensional space. This is why this model is called the Vector Space Model. The term frequency of a term t in a document d is the number of occurrences of t in d. Let it be denoted by $TF(d, t)$. There are ways to normalize this term frequency. For example, in our work, we use the Cornell-SMART (System for the Mechanical Analysis and Retrieval of Text) system that uses the following formula to compute the normalized term frequency [1]:

$$\begin{aligned} TF(d, t) &= 0 && \text{if } freq(d, t) = 0 \\ &= 1 + \log_{10}(1 + \log_{10}(freq(d, t))) && \text{otherwise} \end{aligned} \tag{3.5}$$

There is another important measure called the Inverse Document Frequency (IDF) that represents the scaling factor, or the importance, of a term t. If a term t occurs frequently in many documents, its importance will be scaled down due to its reduced discriminative power. For example, the term 'football' is likely to be less relevant if it occurs in a set of news articles about a football tournament. So we need to scale down its importance accordingly. According to the same Cornell-SMART system, $IDF(t)$ is defined by the following formula:

$$IDF(t) = \log_{10}((1 + |d|) / |d_t|) \tag{3.6}$$

where $|d|$ is the total number of documents, and $|d_t|$ is the number of documents containing the term t. Here $|d_t|$ cannot be zero as then we would not have included the

term t in our term-list. In a complete vector-space model, the TF and the IDF measures are combined together, which forms the TF-IDF measure used throughout this work:

$$TF\text{-}IDF(d, t) = TF(d, t) \times IDF(t) \tag{3.7}$$

Table 3.3 shows a sample TF-IDF matrix where the i^{th} row represents a document vector for document d_i, the j^{th} column represents the TF-IDF values for term t_j, and each entry registers $TF\text{-}IDF(d_i, t_j)$.

Table 3.3 Sample TF-IDF matrix

Term → / ***Document*** ↓	t_1	t_2	…	…	t_{j-1}	t_j	t_{j+1}	…	…	t_n
d_1	0	0	…	…	3	0	7	…	…	0
d_2	0	5	…	…	0	5	0	…	…	6
…	…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…	…
d_{i-1}	0	0	…	…	2	4	0	…	…	8
d_i	5	8	…	…	1	1	0	…	…	0
d_{i+1}	0	7	…	…	5	0	0	…	…	2
…	…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…	…
d_m	3	4	…	…	0	0	8	…	…	5

3.5 Sparse Matrix Representation

A sparse matrix is a matrix populated primarily with zeros. When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard matrix structures and algorithms are slow and consume large amounts of memory when applied to large sparse matrices. Sparse

data is by nature easily compressed, and this compression almost always results in significantly less memory usage. Indeed, some very large sparse matrices are impossible to manipulate with the standard algorithms.

The naïve data structure for a matrix is a two-dimensional array. Each entry in the array represents an element *a(i, j)* of the matrix and can be accessed by the two indices *i* and *j*. For an $m \times n$ matrix we need at least enough memory to store ($m \times n$ storage locations) entries to represent the matrix. Many, if not most, entries of a sparse matrix are zeroes. An example of a sparse matrix is given in Table 3.4. The matrix has ten rows and ten columns, but has only eighteen non-zero values out of the total possible one hundred. The basic idea when storing sparse matrices is to store only the non-zero entries as opposed to storing all of them. Depending on the number and distribution of the non-zero entries, different data structures can be used and yield huge savings in memory when compared to a naïve approach.

Table 3.4 Example of a sparse matrix

0	1	0	0	0	0	0	6	0	0
0	0	0	4	0	5	0	0	1	0
0	0	0	9	0	0	2	0	0	0
0	2	0	0	0	0	9	0	0	9
0	0	1	0	0	7	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	4	0	0	0	0	0	5	0	0
0	0	0	4	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0
0	0	0	0	7	0	0	0	0	3

In our work, we use the following representation for efficient memory usage. The matrix is stored as non-zero-only values. The rows appear as jagged two-dimensional arrays. Each row is stored in a single-dimensional array (that grows as necessary), and the column indexes are stored accordingly. For example, the matrix in Table 3.4 would be internally stored as (assuming row and column indices start from 1):

[1, 2], [1, 8]

[2, 4], [2, 6], [2, 8]

[3, 4], [3, 7]

[4, 2], [4, 7], [4, 10]

[5, 3], [5, 6]

[7, 2], [7, 8]

[8, 4], [8, 10]

[10, 5],[10, 10]

With this we come to the end of this chapter. In the next chapter, we discuss the proposed work in detail.

Chapter 4

PROPOSED WORK

In this chapter, we will describe the complete scheme of our proposed text clustering algorithm in detail.

4.1 Overall Proposed Scheme

In this section, we give a broad overview of our proposed work. The associated block diagram is given in Figure 4.1. As mentioned in Chapter 3, the first step in all text mining operations involves data preprocessing. Through a sequence of steps which include stopword removal, word stemming, and dimensionality reduction, we arrive at the final TF-IDF matrix which will act as the preprocessed dataset on which we will run our algorithm. Step 1 involves deriving the co-occurrences of terms from the TF-IDF matrix to build the co-occurrence matrix (details in Section 4.2). In the second step, we build term-clusters based on term co-occurrence and the naïve Bayesian concept (details in Section 4.3). Next, in Step 3, we compute the arithmetic means of TF-IDF values corresponding to every term-cluster for each document and assign the document to the cluster with the highest mean, thus forming document clusters (details in Section 4.4). Finally, in Step 4, we apply shape pattern-based similarity to group documents within each document cluster to form document sub-clusters (details in Section 4.5).

4.2 The Co-occurrence Matrix

We now have the TF-IDF matrix with us. Each row of the matrix corresponds to a document d and each column corresponds to a term t. We also assume that henceforth the sparse matrix representation is used wherever applicable. We will not explicitly explain sparse matrix operations repeatedly but instead focus on the concept of the clustering procedure. As we progress through the sequence of operations, we will explain the entire concept with a simple example. We will start from the small

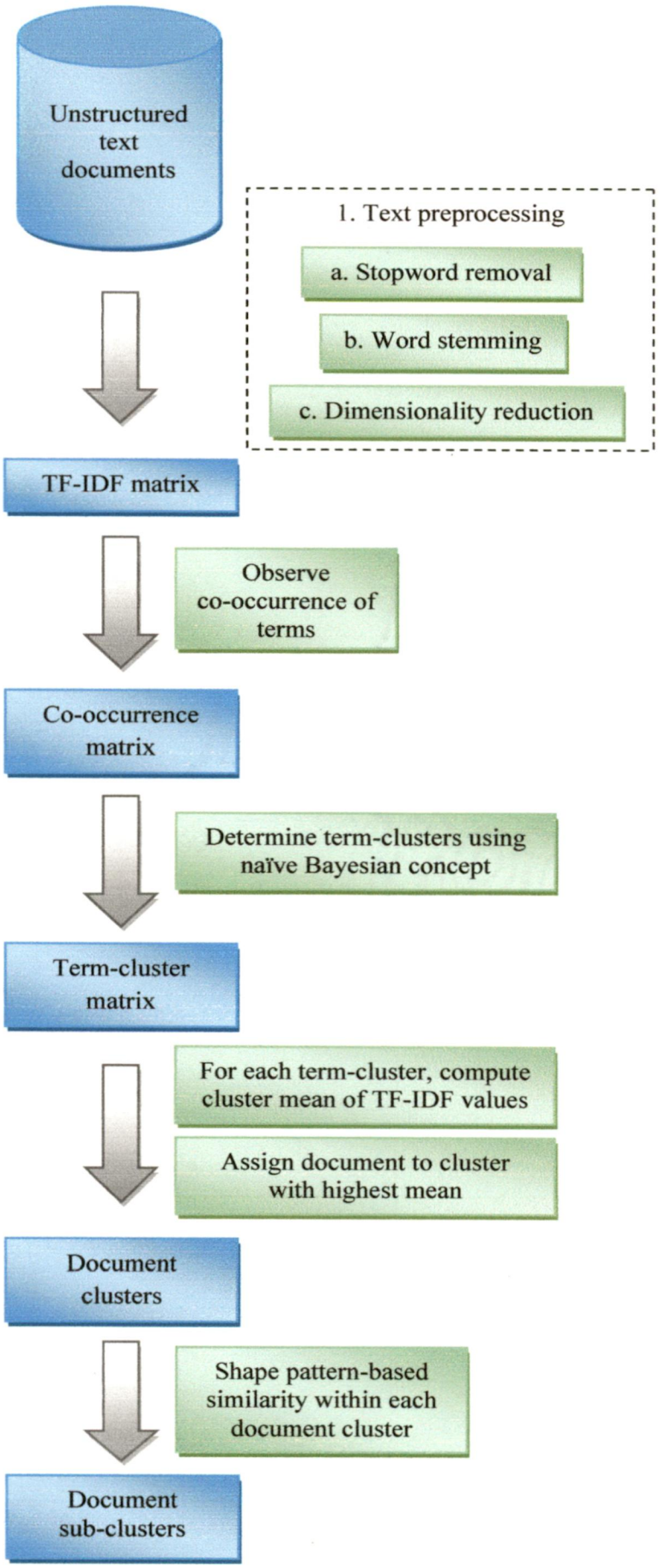


Fig. 4.1 Complete block diagram of proposed clustering scheme

TF-IDF matrix given in Table 4.1. It has twenty rows and ten columns indicating that it corresponds to a set of twenty documents and ten features.

Table 4.1 TF-IDF matrix for example dataset

	t_1	t_2	t_3	...	...	...	...	t_8	t_9	t_{10}
d_1	9	0	0	...	...	...	...	8	8	0
d_2	0	8	0	...	...	...	...	0	0	0
d_3	8	0	0	...	...	...	...	9	8	0
d_4	0	9	0	...	...	...	...	0	0	0
d_5	0	0	7	...	...	...	...	0	0	9
d_6	7	0	0	...	...	...	...	8	9	0
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
d_{15}	0	0	8	...	...	...	...	0	0	9
d_{16}	9	0	0	...	...	...	...	7	9	0
d_{17}	0	9	0	...	...	...	...	0	0	0
d_{18}	7	0	0	...	...	...	...	8	9	0
d_{19}	0	9	0	...	...	...	...	0	0	0
d_{20}	0	3	0	...	...	...	...	0	1	0

The TF-IDF matrix is generally stored as a data file on disk. As a result, before we can begin to use the matrix, we must load it (whole or the relevant portion) onto the main memory. We must also convert the character representation into a number format so that it can be used in future computations. The TF-IDF matrix is initially present in a CSV (Comma Separated Value) format. Non-printing characters like ← (CR) and ¶ (LF) are also present within it. Detection of these characters becomes useful when converting the file from a character-only text file to a matrix in a numeric format. This format is used as a standard nowadays and most commercial

text mining packages, including WEKA [22], accept files formatted this way to load their data. We use WEKA later on to compare our experimental results (Section 5.4). Once the conversion process is complete, we can build the co-occurrence matrix from it. We present the algorithm to build the co-occurrence matrix from the TF-IDF matrix in Figure 4.2.

```
TFIDF-TO-COOCC(TFIDFMat, NumDocs, NumFeats)
1  for every pair of features 1 to NumFeats
2        do for every document 1 to NumDocs
3              min ← minimum of TF-IDF values for feature pair in current
               document
4        increment co-occurrence measure between feature pair by min
5        insert co-occ-measure in proper position in co-occurrence matrix
6        do not repeat for feature pair j and i if co-occurrence between i and j
         already computed
7  return co-occurrence matrix
```

Fig. 4.2 Algorithm to obtain co-occurrence matrix

In this algorithm, we study the co-occurrences between terms. When two terms co-occur in a document, we take the minimum of the number of their occurrences as a co-occurrence measure. For example, if term 4 and term 5 occur 5 and 15 times in a document respectively, the strength of their co-occurrence is appropriately represented only when we take 5 as the number as **co**-occurrences. The sum or the difference would not reflect the strength of their co-occurrence. It gives misleading interpretations. For example, let the same two terms occur in a document 2 and 100 times respectively. If we add them, a value of 102 would not express the weak correlation between term 4 and term 5, as 102 is a misguidingly high number. Similarly, the difference is 98, which is also very high, and does not express their weak mutual relation. But we need not adopt any advanced formula; simply taking their minimum, which is 2, reflects the weak degree of correlation. This is also computationally very inexpensive. It also works when the terms are closely related, i.e., their co-occurrence is high. For example, if the same two terms occurred in a document 90 and 95 times respectively, taking the minimum of 90 reflects that their degree of co-occurrence is very high. Here it is understood that when we speak about number of term occurrences, we are referring to corresponding TF-IDF values and not simple term frequencies. This will not be repeated explicitly henceforth and will be

assumed throughout the remainder of this text. The co-occurrence matrix stores the total number of such co-occurrences between all pairs of terms across all documents. The co-occurrence measure between any two terms is a symmetric measure, i.e.

$$Co\text{-}occurrence(term\ i,\ term\ j) = Co\text{-}occurrence(term\ j,\ term\ i) \quad (4.1)$$

As a result, if *co-occurrence(term i, term j)* has been computed previously, we need neither compute nor store *co-occurrence(term j, term i)*. We proceed to build this matrix in a row-major fashion, so the resulting co-occurrence matrix is an upper triangular matrix. We can now formally state the method used to build the co-occurrence matrix *CoOccMat* mathematically:

$$\begin{aligned} CoOccMat(i,j) &= \sum_{k=1}^{m}\{minimum(TFIDFMat(k,i), TFIDFMat(k,j))\} && \text{if } i < j \\ &= -1 && \text{if } i = j \\ &= 0 && \text{if } i > j \end{aligned} \quad (4.2)$$

where m is the total number of documents

and *TFIDFMat* is the input TF-IDF matrix

For the trivial case of a term co-occurring with itself, we insert a value of -1 in the corresponding location. The co-occurrence matrix obtained from the TF-IDF matrix shown in Table 4.1 is given in Table 4.2.

Table 4.2 Co-occurrence matrix for example dataset

	t_1	t_2	t_3	...	...	...	...	t_8	t_9	t_{10}
t_1	-1	0	0	0	0	0	15	65	70	0
t_2	0	-1	0	43	38	56	0	0	1	0
t_3	0	0	-1	9	0	0	22	0	0	22
...	...	0	0	-1	43	42	9	0	1	9
...	...	...	0	0	-1	37	0	0	1	0
...	...	...	...	0	0	-1	0	0	1	0
...	...	...	...	...	0	0	-1	15	15	22
t_8	...	...	...	...	...	0	0	-1	68	0
t_9	...	...	...	...	...	...	0	0	-1	0
t_{10}	...	...	...	...	...	...	...	0	0	-1

We will illustrate how we obtain the values at location (1, 9). Since 1 < 9, the first condition of Equation 4.2 will hold.

$$
\begin{aligned}
CoOccMat(1,9) = {} & minimum(TFIDFMat(1,1), TFIDFMat(1,9)) + \\
& minimum(TFIDFMat(2,1), TFIDFMat(2,9)) + \\
& minimum(TFIDFMat(3,1), TFIDFMat(3,9)) + \\
& minimum(TFIDFMat(4,1), TFIDFMat(4,9)) + \\
& minimum(TFIDFMat(5,1), TFIDFMat(5,9)) + \\
& minimum(TFIDFMat(6,1), TFIDFMat(6,9)) + \\
& minimum(TFIDFMat(7,1), TFIDFMat(7,9)) + \\
& minimum(TFIDFMat(8,1), TFIDFMat(8,9)) + \\
& minimum(TFIDFMat(9,1), TFIDFMat(9,9)) + \\
& minimum(TFIDFMat(10,1), TFIDFMat(10,9)) + \\
& minimum(TFIDFMat(11,1), TFIDFMat(11,9)) + \\
& minimum(TFIDFMat(12,1), TFIDFMat(12,9)) + \\
& minimum(TFIDFMat(13,1), TFIDFMat(13,9)) + \\
& minimum(TFIDFMat(14,1), TFIDFMat(14,9)) + \\
& minimum(TFIDFMat(15,1), TFIDFMat(15,9)) + \\
& minimum(TFIDFMat(16,1), TFIDFMat(16,9)) + \\
& minimum(TFIDFMat(17,1), TFIDFMat(17,9)) + \\
& minimum(TFIDFMat(18,1), TFIDFMat(18,9)) + \\
& minimum(TFIDFMat(19,1), TFIDFMat(19,9)) + \\
& minimum(TFIDFMat(20,1), TFIDFMat(20,9)) \\
= {} & 8+0+8+0+0+7+0+8+0+8+7+0+8+0+0+9+ \\
& 0+7+0+0 \\
= {} & 70, \text{ which can be verified from Table 4.2.}
\end{aligned}
$$

We are now ready to proceed to build the term cluster matrix using the co-occurrence matrix. The algorithm for this procedure is given in Figure 4.3.

4.3 The Term Cluster Matrix

In this algorithm, we try to form clusters within our term-set. Terms which are linked semantically will be grouped under one cluster. We assume that terms which

have a high degree of co-occurrence are likely to be linked semantically. For example, the terms *movies*, *films*, *actors*, and *director* are all linked semantically.

```
COOCC-TO-TERMCLUS(CoOccMat, NumFeats)
1   for every term 1 to NumFeats
2           do identify which terms it co-occurs with
3           put each such term in cluster of current term
4   terms which do not co-occur with any other term are put in their own
    clusters
5   for every term 1 to NumFeats
6           do identify which clusters it belongs to
7           for every term in such cluster
8                   do calculate co-occurrence probability with itself
9                   compute products of all such probabilities (application of
                            naïve Bayesian concept)
10          select highest probability
11          assign term finally to cluster with highest probability
12  terms which do not co-occur with any other term remain in their own
    clusters
13  return term cluster matrix
```

Fig. 4.3 Algorithm to obtain term-cluster matrix

Indeed, in documents related to *cinema*, we do find these terms co-occurring to a large extent. But again, there exist terms which can be grouped into more than cluster as they co-occur often with more than one group of terms. For example, the term *playback* will be often found in a document set about films (playback singing), sharing occurrence with cinematic terms or in a document set concerned primarily about *music*, co-occurring with music terms like *singer*, *music*, and *guitar*. In our work, we assume that one term may belong to only cluster. We uniquely assign a term to a single cluster. This is done by the application of conditional probability and the naïve Bayesian concept. We calculate the conditional probabilities of a term belonging to each of the possible clusters and assign it to the cluster with the highest probability. Now we will explain the algorithm which has been used.

From the co-occurrence matrix obtained, we come to know which terms co-occur. Initially, each term is treated as a cluster centre and all terms co-occurring with this term are put into the cluster corresponding to this term. Terms which do not co-occur with any other term are the singular terms in their respective clusters. For example, if term 4 co-occurs with term 1, term 5, and term 7, then the fourth term cluster will contain terms 1, 4, 5, and 7. Again if term 6 co-occurs with terms 4, 9, and

10, then the sixth term cluster will contain terms 4, 6, 9, and 10. We note that the co-occurrence of term 1 with term 4, and the co-occurrence of term 4 with term 6, does not imply the co-occurrence of term 1 with term 6, i.e., the co-occurrence relation is not transitive in nature. Had it been so, terms 1, 4, 5, 6, 7, 9, and 10 all would have been grouped under one large cluster. For example, the term *actors* and *playback* may co-occur frequently, as may *playback* and *singing*; but *actors* and *singing* may never co-occur. The initial term cluster matrix *IntlTermClusMat* is built from the co-occurrence matrix *CoOccMat* according to the following equation:

$$\begin{aligned} IntlTermClusMat(i, j) &= 1 \quad \text{if } CoOccMat(i, j) \neq 0 \text{ or } CoOccMat(j, i) \neq 0 \\ &= 0 \quad \text{otherwise} \end{aligned} \tag{4.3}$$

In the newly obtained matrix, the rows correspond to initial clusters and the columns to the member terms. Following our example, since initially each term is a cluster centre, there are ten rows. The initial term cluster matrix obtained from the co-occurrence matrix in Table 4.2 is given in Table 4.3. Since term 4 co-occurs with terms 2, 3, 5, 6, 7, 9, and 10, all corresponding entries in the fourth cluster (fourth row) are marked as 1. We will show how the useless terms are filtered out before determination of the final clusters.

Table 4.3 Initial term cluster matrix for example dataset

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
C_1	1	0	0	0	0	0	1	1	1	0
C_2	0	1	0	1	1	1	0	0	1	0
C_3	0	0	1	1	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
C_8	1	0	0	0	0	0	1	1	1	0
C_9	1	1	0	1	1	1	1	1	1	0
C_{10}	0	0	1	1	0	0	1	0	0	1

So now we have the initial clusters as

$C_1 = \{t_1, t_7, t_8, t_9\}$

$C_2 = \{t_2, t_4, t_5, t_6\}$

$C_3 = \{t_3, t_4, t_7, t_{10}\}$

$C_4 = \{...\}$

$C_5 = \{...\}$

$C_6 = \{...\}$

$C_7 = \{...\}$

$C_8 = \{t_1, t_7, t_8, t_9\}$

$C_9 = \{t_1, t_2, t_4, t_5, t_6, t_7, t_8, t_9\}$

$C_{10} = \{t_3, t_4, t_7, t_{10}\}$

We now have to remove the clashes and assign a term uniquely to a cluster. We use the naïve Bayesian concept now. It is based on the assumption that a term's probability of belonging to a particular cluster is independent of its probabilities of its belonging to the other clusters. This effectively translates to the fact that a term's probability of co-occurrence with one term is independent of its probability of co-occurrence with another term. This assumption can be called cluster conditional independence. Just like the corresponding Bayesian classifiers, it is naïve in this regard. The basis of this concept is the Bayes' theorem and conditional probability (refer to Section 2.2). We will use notations similar to that section for ease in understanding. Let X represent one of m terms and $C_1, C_2, \ldots, C_m$ the term-clusters. Then, $P(C_i|X)$ represents the posterior probability of term X belonging to cluster C_i, given that we know X. It is also called the *a posteriori* probability of C_i conditioned on X. In contrast, $P(C_i)$ is the prior probability, or *apriori* probability, of C_i. This is the probability of the cluster C_i being chosen at random from the m clusters. The posterior probability, $P(C_i|X)$, is based on more information (i.e. knowledge of term number) than the prior probability, $P(C_i)$, which is independent of X. Similarly, $P(X|C_i)$ is the posterior probability of X conditioned on C_i. It is the probability that given the cluster chosen is C_i, the term chosen is X. $P(X)$ is the prior probability of X, i.e. the probability of the term X being chosen at random from the list of all terms. By the Bayes Theorem (Equation (2.1)), we have

$$P(C_i|X) = (P(X|C_i) \times P(C_i)) / P(X) \qquad (4.4)$$

Given a term X, our clustering scheme will predict that X belongs to the term-cluster having the highest posterior probability, conditioned on X. So it predicts that term X belongs to the cluster C_i if and only if

$$P(C_i|X) > P(C_j|X) \text{ for } 1 \leq j \leq m, j \neq i.$$

Thus we maximize $P(C_i|X)$. The cluster C_i for which $P(C_i|X)$ is maximized is called the maximum posteriori hypothesis. $P(C_i|X)$ is given by Equation (4.4). As $P(X)$ is constant for all clusters, only $(P(X|Ci) \times P(Ci))$ needs be maximized. Since the cluster prior probabilities are not known, it is assumed that the clusters are equally likely, that is, $P(C_1) = P(C_2) = \ldots = P(C_m)$, and we would therefore maximize $P(X|C_i)$.

By the naïve assumption of cluster conditional independence, we can estimate $P(X|C_i)$ in the following way:

$$P(X|C_i) = \prod_{k=1}^{n} P(\text{co} - \text{occurrence of } X \text{ and } X_k) \quad (4.5)$$

where n is the number of terms in C_i

$$P(X|C_i) = P(\text{co-occurrence of } X \text{ and } X_1) \times P(\text{co-occurrence of } X \text{ and } X_2) \times \ldots \times P(\text{co-occurrence of } X \text{ and } X_n)$$

where $X_1, X_2, \ldots, X, \ldots X_n$ are the terms belonging to C_i (4.6)

The probability of co-occurrence of terms X_i and X_j is defined by

$$P(\text{co} - \text{occurrence of } X_i \text{ and } X_j) = \frac{\text{No. of co-occurrences of } X_i \text{ and } X_j}{\text{No. of co-occurrences of } X_i \text{ and all other terms}} \quad (4.7)$$

The probability of co-occurrence of a term with itself (trivial case) is assumed to be one. There is another important modification to be introduced. In the product of Equation (4.5), if any of the co-occurrence probabilities is zero, it makes the whole product zero. But a term need not co-occur with every other term in its cluster. But without any modification to our existing calculations, non-co-occurrence with even a single term in a term-cluster would nullify the whole product. Without the zero

probability, we may have ended up with a high probability, suggesting that X may have belonged to class C_i. A zero probability cancels the effects of all of the other (*posteriori*) probabilities (on C_i) involved in the product. There is a simple trick to avoid this problem. We can assume that our training database is so large that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the case of probability values of zero. This technique for probability estimation is known as the Laplacian correction (refer to Section 2.2). As a result, we treat the number of co-occurrences of a term with another term (with which it does not co-occur) as 1, and similarly the denominator also gets increased by 1 in the probability calculations. Now we present the algorithm (Figure 4.4) for calculating the co-occurrence probability between two terms.

```
CALC-COOCC-PROBAB(term1, term2, CoOccMat, NumFeats)
1   numerator ← number of co-occurrences between term1 and term2
    obtained from CoOccMat
    MinVal is the minimum of the two terms
    MaxVal is the maximum of the two terms
    numerator ← CoOccMat[MinVal][MaxVal]
2   denominator ← number of co-occurrences between term1 and all other
    terms
    for every element in term1-th row and term1-th column in CoOccMat
            do increment denominator by corresponding value
8   if numerator ← 0          ► Laplacian correction
9           then numerator ← numerator + 1
10          denominator ← denominator + 1
11  if denominator ≠ 0
12          then probability ← numerator / denominator
13  else
14          probability ← 0
15  return probability
```

Fig. 4.4 Algorithm to calculate co-occurrence probability

Now we come back to our example. From the data presented in Table 4.3 and the ensuing initial cluster information, we find that term 4 is initially a member of eight clusters – C_2, C_3, C_4, C_5, C_6, C_7, C_9, and C_{10}. We will show the probability computations in detail for one of these clusters and give the cluster-belonging probabilities for the other ones.

To find the probability of term 4 belonging to cluster 2, we have to find the products of the probabilities of term 4 co-occurring with each term of cluster 2. Cluster 2 contains four terms – term 2, term 4, term 5, term 6, and term 9.

Probability(co-occurrence of term 4 and term 2)
= Number of co-occurrences of term 4 and term 2 / (Number of co-occurrences of term 4 with all other terms)
= 43 / (43 + 9 + 43 + 42 + 9 + 1 + 9)
= 43 / 156
= 0.2756
Probability(co-occurrence of term 4 and term 4) = 1.0000 (Trivial case)
Probability(co-occurrence of term 4 and term 5) = 43 / 156 = 0.2756
Probability(co-occurrence of term 4 and term 6) = 42 / 156 = 0.2756
Probability(co-occurrence of term 4 and term 9) = 1 / 156 = 0.0064

Thus, Probability(term 4 belonging to cluster 2)
$= P(X_4|C_2)$
$= 0.2756 \times 1.0000 \times 0.2756 \times 0.2756 \times 0.0064$
$= 1.3113 \times 10^{-4}$

Similarly,
$P(X_4|C_3) = 2.1336 \times 10^{-4}$
$P(X_4|C_4) = 2.5179 \times 10^{-8}$
$P(X_4|C_5) = 1.3113 \times 10^{-4}$
$P(X_4|C_6) = 1.3113 \times 10^{-4}$
$P(X_4|C_7) = 4.9938 \times 10^{-11}$
$P(X_4|C_9) = 3.0691 \times 10^{-10}$
$P(X_4|C_{10}) = 1.9202 \times 10^{-4}$

Clearly, $P(X_4|C_3)$ is the highest and so $P(C_i|X_4)$ is maximized for $i = 3$. Hence, term 4 is finally assigned to the third cluster. At the end of this process, the terms which did not co-occur with any other term still remain in their own clusters. After this round of final assignment, we obtain a term cluster matrix in which every term belongs to a single cluster only. So there is exactly a single one entry in a single

column. As we traverse a row columnwise (corresponding to an equivalent cluster), the terms which have 1s in their corresponding locations belong to the cluster under consideration. By this time, we can understand the effectiveness of the use of the sparse matrix representation, as the various data matrices encountered have been shown to be getting sparser. The matrices keep getting sparser with an increase in the dataset size and the number of terms. Coming back to our discussion on the term clusters, finally we convert the term cluster matrix into a memory-efficient bag-of-words representation. This means that instead of a row containing 0s in locations of terms not belonging to the cluster and 1s in locations of terms belonging to the cluster, the row directly contains the identifiers of the terms belonging to the cluster. This matrix is the final term cluster matrix and is used for document cluster and sub-cluster determination.

Table 4.4 Term cluster matrix for example dataset

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
C_1	1	0	0	0	0	0	0	1	1	0
C_2	0	1	0	0	1	1	0	0	0	0
C_3	0	0	1	1	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
C_8	0	0	0	0	0	0	0	0	0	0
C_9	0	0	0	0	0	0	0	0	0	0
C_{10}	0	0	0	0	0	0	0	0	0	0

Table 4.5 Final term cluster matrix for example dataset

	t_1	t_2	t_3	t_4
C_1	1	8	9	0
C_2	2	5	6	0
C_3	3	4	7	10

Table 4.5 shows the memory-efficient bag-of-words form of the term cluster matrix shown in Table 4.4. As shown earlier, term 4 was assigned to cluster 3. Here we finally have terms 1, 8, and 9 in cluster 1, terms 2, 5, and 6 in cluster 2, and terms 3, 4, 7, and 10 in cluster 3. Now we move on to the techniques used for document cluster and sub-cluster determination by the use of these matrices.

4.4 Document Cluster Determination

Now that we have determined our desired term-clusters, the next task is to use them to obtain the document clusters. We do this by computing the arithmetic mean of the TF-IDF values corresponding to the terms of every cluster, sequentially. The document will be assigned to the cluster yielding the highest mean. The main implication of this is that the number of document clusters is equal to the number of term-clusters. It does not vary with the number of documents, provided the number of terms remains fixed. This is very helpful as the number of documents D is generally much larger than the number of terms N ($D >> N$). As a result, the number of term-clusters is also much lower than D. This helps us divide a large document set into a manageable number of clusters. Mathematically, the cluster number of document number i is given by

$$Cluster(i) = max_p[(\sum_{k=1}^{n} TFIDFMat(i, FinalTermClusMat(p,k)))/n] \quad (4.8)$$

where *TFIDFMat* is the TF-IDF matrix
FinalTermClusMat is the final term-cluster matrix
and, the maximization is performed over all p, i.e. all term-clusters;
n is the number of terms in each cluster; so n may vary from cluster to cluster

The algorithm for this procedure is given in Figure 4.5. The document clustering results are stored in a document cluster matrix which has three columns and a number of rows equal to the number of documents. The first column stores the document identifier, the second column stores the document cluster identifier, and third column is allocated to store the document sub-cluster identifier. After this first

level of clustering is performed, this matrix is returned but the third column, as expected, is still empty. It is filled in only after the sub-clustering procedure is also completed.

```
CLUS-BY-MEAN(TFIDFMat, FinalTermClusMat, NumDocs, NumFeats)
1   for every document from 1 to NumDocs
2           do for every term cluster
3                   do compute arithmetic mean of values in document vector of
                            current document in TFIDFMat corresponding to
                            terms in current term cluster
4           assign document to term cluster with highest mean
5   return final document cluster matrix with cluster information but without
    sub-cluster information
```

Fig. 4.5 Algorithm for document clustering

The document cluster matrix (with the first two columns filled) for our example dataset is given in Table 4.6. We will show the cluster determination for one example, document number 20. The term-clusters are terms {1, 8, 9}, {2, 5, 6}, and {3, 4, 7, 10}. The corresponding cluster-wise means are as follows:

Mean for cluster 1 = (0 + 0 + 1) / 3 = 0.33

Mean for cluster 2 = (3 + 8 + 5) / 3 = 5.33

Mean for cluster 3 = (0 + 8 + 0 + 0) / 4 = 2.00

Since the mean for cluster 2 is the highest, the document belongs to the second cluster. Now that document clustering is complete, we proceed to finding sub-clusters in the next section.

4.5 Document Sub-cluster Determination

The document clusters provide us with a broad grouping of the documents. Often we require a finer level of clustering which is provided by our sub-clustering procedure. Here the representation of text documents as sequences in the form document vectors is of fundamental importance. Here we apply the concept of shape pattern-based similarity. We assume a logical graph consisting of the points in the

Table 4.6 Document cluster matrix for example dataset (after Level I)

Document id	Cluster id	Sub-cluster id
1	1	0
2	2	0
3	1	0
4	2	0
5	3	0
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
16	1	0
17	2	0
18	1	0
19	2	0
20	2	0

TF-IDF matrix corresponding to the cluster of the document. The TF-IDF values (equivalently term weights) (y-axis) are observed against the terms (x-axis). Here we use the word 'observed' and not 'plotted' because though we are conceptually dealing with shapes and graphs, explicit plotting and a manual study of the graphs are not necessary. The shape of this plot gives the inherent pattern associated with this document. Computations on the document vectors help us in performing the equivalent operations. The graphical representations, as provided in the figures later, help us in an easy illustration of the concept. The algorithm for the sub-clustering procedure is presented in Figure 4.6.

```
SUB-CLUS-BY-SHAPE(TFIDFMat, FinalTermClusMat, DCM, NumDocs, NumFeats)
1   declare ShapeList to store list of unique patterns
2   initially ShapeList contains only end-marker
3   for every document from 1 to NumDocs
4           declare and initialize string to store associated shape pattern
5           do fetch TF-IDF values in document vector corresponding to terms
                    of term cluster
6           for every pair of consecutive terms in term cluster
7                   do observe difference between corresponding TF-IDF values
8                   if TF-IDF value corresponding to second term higher
9                           then add U to current pattern as graph moves Up
                            ► Here graph refers to plot of TF-IDF values versus
                                    corresponding terms
10                  else if TF-IDF value corresponding to second term lower
11                          add D to current pattern as graph moves Down
12                  else (TF-IDF values equal)
13                          add L to current shape pattern as graph remains
                                    Level
14          compare shape pattern with every pattern in ShapeList sequentially
15          if match is found
16                  then associate document with current shape identifier
17          else
18                  add new shape to ShapeList
19                  push end marker by one position
20                  associate document with new shape identifier
21  Sort in ascending order of shape indices within clusters
22  Assign first document to first sub-cluster
23  for i ← 1 to NumDocs
24          do if shapes match and clusters match for consecutive documents
25                  then assign documents to previous sub-cluster
26          else if shapes do not match or clusters do not match
27                  then create new sub-cluster and assign document to it
28  return final document cluster matrix with sub-cluster information
```

Fig. 4.6 Algorithm for document sub-clustering

The sub-clustering procedure is also fully unsupervised and based on the notion of the relative importance of the various terms in the term-cluster in the document under consideration. This is reflected by the changes that the TF-IDF values go through corresponding to the terms in the term-cluster of the document.

Let there be k terms in the term-cluster of the document under consideration. This corresponds to k points on the x-axis. Corresponding to the k points in a term-cluster, there are $(k - 1)$ transition points of importance in the graph. The differences in the TF-IDF values over consecutive points are of interest to us and help in

determining the shape pattern present in the plot. These differences help us in determining the gradient of the graph as it moves across these transition points. A *(k - 1)*-character array for every document is maintained which stores the alphabets *'U', 'D', or 'L'* according as the graph moves up, down, or remains level (three possibilities) across a transition point, in sequence, i.e. this array stores the description of the shape pattern present in the document's graph. As a result, there will be a total of 3^{k-1} possible shapes inherent in the document vectors, a number which may become quite large for a large k. But even for large real datasets, only a much reduced set of shape patterns appear (the number of patterns discovered are only of significance within a sub-category, and not across them; as explained later). This has been shown experimentally in Chapter 5. Whenever we come across a new document, the shape array for this document is compared to the arrays of the existing shapes, which are maintained separately in a text file. If the pattern matches with an existing one, the index number for this shape (shape identifier) is assigned to the document. If it is a new shape, the next unique serial number is assigned to the shape and the document, and the pattern is added to the list of existing shapes. This numbering is done on a global basis, i.e. two different shapes always have different serial numbers, even if they appear in different sub-categories only. This simplifies the indexing procedure without increasing any time or space requirements. The set of all the indices of the obtained shape patterns forms the shape alphabet. Shape identifier 0 (null) is reserved for documents with clusters where the number of terms is one, i.e. a case when no pattern can be formed. Documents within a particular cluster with the same shape pattern (or equivalently sharing the same shape identifier) form sub-clusters. This completes the clustering procedure within the clusters based on shape patterns. Let us take an example. Say, a document belongs to cluster with five terms in it. So k is 5. So we have $k - 1$, i.e. 4 transition points. Let the corresponding TF-IDF values be {9, 16, 16, 21, 6}. Then the associated graph can be said to move up, remain level, again move up, and finally move down. As a result, the associated shape pattern will be {*U, L, U, D*} (Figure 4.7). We note that only the shape of the pattern (and not the magnitude of a rise or a fall) is sufficient to reflect the importance of the respective terms within the document, which is the basis for our sub-clustering procedure. The shapes inherent in the documents of our example dataset are enlisted in Table 4.7 and shown pictorially in Figure 4.8.

The sub-cluster identifiers are copied back into the third column of the document cluster matrix. This matrix, for our example dataset, with the sub-cluster information filled in, has been given in Table 4.8. A vector from this matrix may be represented as

{document_id, cluster_id, sub-cluster_id}

Our test document from the previous section, document 20, has three terms in its term cluster, is associated with the second shape identifier ($\{U, D\}$), and belongs to the sixth sub-cluster, which is the first sub-cluster in the second cluster. These details are evident from Tables 4.8 and 4.9 (sorted by cluster number, sub-cluster number, document number). Three clusters and ten sub-clusters (across all clusters)

Fig. 4.7 Shape pattern $\{U, L, U, D\}$

Table 4.7 Shape patterns in example dataset

Shape identifier	Shape pattern description
0	*nil*
1	$\{D, L\}$
2	$\{U, D\}$
3	$\{L, L\}$
4	$\{U, U\}$
5	$\{D, U\}$
6	$\{D, D\}$
7	$\{U, U, D\}$
8	$\{U, D, L\}$

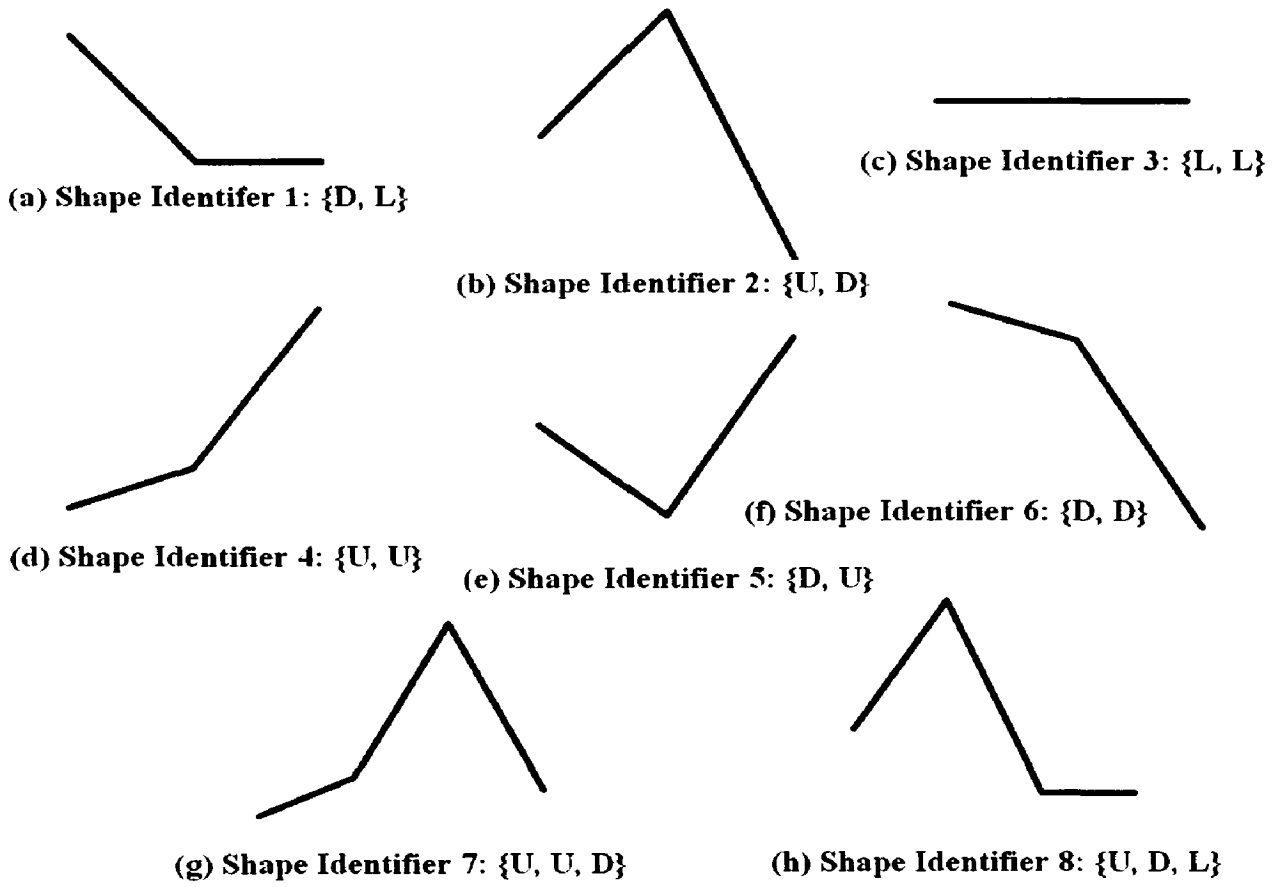


Fig. 4.8 Pictorial representation of shapes in example dataset

Table 4.8 Document cluster matrix for example dataset (after Level II)

Document id	Cluster id	Sub-cluster id
18	1	1
1	1	2
6	1	2
10	1	2
11	1	2
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...

...	...	...
20	2	6
12	2	7
17	2	7
2	2	8
15	3	9
5	3	10
9	3	10

were found in our example dataset. Thus the average number of sub-clusters per cluster came out to be 3.33. The average number of documents per cluster and sub-cluster were 6.67 and 2.00 respectively. This summary of our example dataset concludes this chapter. The detailed results for the much larger datasets used for validation purposes have been provided in the next chapter.

Chapter 5

RESULTS AND DISCUSSION

In this chapter, we will elaborate on our experimental results and the associated discussion. We will describe the datasets used for validation and the implementation details in the first two sections. Our results are accompanied by comparisons with standard algorithms available in the popular data mining software suite WEKA [22]. We conclude this chapter by giving the time complexity of our algorithm.

5.1 Datasets used for Validation

We have used a variety of benchmark datasets [23] available on the internet to validate our algorithm. The details of these datasets are given below, in increasing order of complexity.

Case 1: The TF-IDF matrix corresponds to a set of five thousand documents and fifty terms. The term set consists of groups of co-occurring terms, with no co-occurrence between terms of different groups.

Case 2: The TF-IDF matrix corresponds to a set of five thousand documents and fifty terms. The term set consists of groups of co-occurring terms, but with co-occurrence between terms of different groups.

Case 3: The TF-IDF matrix corresponds to a set of five thousand documents and two hundred terms. The term set consists of groups of co-occurring terms, with co-occurrence between terms of different groups.

Case 4: We deal with two special cases in the last two datasets. The first one is named ADA [23, 24]. ADA has marketing applications. The task of ADA is to discover high revenue people from census data, presented in the form of a two-class classification problem. The raw data from the

census bureau is known as the Adult database in the UCI machine-learning repository. The fourteen original attributes (features) represented age, workclass, education, marital status, occupation, and native country. They included continuous, binary and categorical attributes. They were finally aggregated to form a data matrix corresponding to forty six thousand and thirty three text documents, with forty eight terms, each term representing an attribute. We have used the first five thousand rows and all the forty eight columns for our work. Since the dataset is known to have only two classes (clusters), it gives us the opportunity to verify our algorithm in cases where large datasets have only a few underlying clusters.

Case 5: The last dataset is named SYLVA [23, 24], an ecology application. The task of SYLVA is to classify forest cover types. The forest cover type for 30 × 30 metre cells was obtained from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. The problem dealt with the study of Ponderosa pine versus everything else. The input matrix consisted of one lakh, forty five thousand, two hundred and fifty two rows (documents) (out of which we have used the first five thousand to maintain uniformity among the datasets) and two hundred and sixteen input variables (terms) (all have been considered). Half of these features are known to be distractors. As a result, it is known that there is only one major cluster within the entire dataset although it is not immediately apparent from the huge matrix with lots of stray variables having non-zero values. This also proved to be an interesting test case.

5.2 Implementation Details

This work has been fully programmed in Java, using the NetBeans IDE platform [25], which is open source and freely downloadable from the internet. The project was implemented on a system running Windows XP Professional Version with Service Pack 2, with a system memory of 1 GB and the processor used being Intel Core 2 Duo 2.13 GHz.

Six java classes were used – *Main, ReadFromFile, TFIDFToCoOcc, CoOccToTermClus, ClusByMean,* and *SubClusByShape*. But since we have already provided the various algorithms used in our overall scheme (Chapter 4) and attached the source code listing (Appendix A), we will not elaborate on details like the methods present in each of these classes, their inputs and outputs, etc. in this section. We now proceed to the results obtained and the comparisons with standard algorithms.

5.3 Experimental Results

For each of the five datasets listed in Section 5.1, we give the number of documents (ND), terms (NT), clusters (NC) and sub-clusters (NSC), the average number of sub-clusters per cluster (ANSCPC), and the average number of documents per cluster (ANDPC) and sub-cluster (ANDPSC). This summary is given in Table 5.1 (legend is given at the bottom of the table). Due to space constraints, we had to use the abbreviated forms in the column headers. We gradually vary the size of the dataset (number of documents) from one thousand through five thousand (with an increment of one thousand after each phase) and observe the change in our metrics, keeping the number of terms constant. We also record how our metrics vary with the number of terms when we vary the number of terms from forty through two hundred (with an increment of forty after each phase) for dataset 3, keeping the number of documents fixed at five thousand. For all the five cases (case 3 has two parts as shown in Table 5.1), we plot graphs for the results (Figures 5.1 through 5.6) and then explain our findings.

P. T. O.

Table 5.1 Summary of cluster information for all datasets

Case	ND	NT	NC	NSC	ANSCPC	ANDPC	ANDPSC
1	1000	50	16	51	3.189	62.500	19.608
1	2000	50	16	77	4.813	125.000	25.974
1	3000	50	16	80	5.000	187.500	37.500
1	4000	50	16	89	5.563	250.000	44.944
1	5000	50	16	90	5.625	312.500	55.556
Avg.	**3000**	**50**	**16.0**	**77.4**	**4.838**	**187.500**	**36.716**
2	1000	50	16	57	3.563	62.500	17.544
2	2000	50	16	79	4.938	125.000	25.316
2	3000	50	16	83	5.188	187.500	36.145
2	4000	50	16	93	5.813	250.000	43.011
2	5000	50	16	95	5.938	312.500	52.632
Avg.	**3000**	**50**	**16.0**	**81.4**	**5.088**	**187.500**	**34.930**
3	1000	200	57	78	1.368	17.544	12.821
3	2000	200	57	90	1.579	35.088	22.222
3	3000	200	57	107	1.877	52.632	28.037
3	4000	200	57	127	2.228	70.175	31.496
3	5000	200	57	141	2.474	87.719	35.461
Avg.	**3000**	**200**	**57.0**	**108.6**	**1.905**	**52.632**	**26.007**
3	5000	40	25	55	2.200	200.000	90.909
3	5000	80	58	80	1.379	86.207	62.500
3	5000	120	58	117	2.017	86.207	42.735
3	5000	160	58	123	2.121	86.207	40.650
3	5000	200	57	141	2.474	87.719	35.461
Avg.	**5000**	**120**	**51.2**	**103.2**	**2.038**	**109.268**	**54.451**
4	1000	48	2	19	9.500	500.000	52.632
4	2000	48	3	15	5.000	666.667	133.333
4	3000	48	1	9	9.000	3000.000	333.333
4	4000	48	2	13	6.500	2000.000	307.692
4	5000	48	2	13	6.500	2500.000	384.615
Avg.	**3000**	**48**	**2.0**	**13.8**	**7.300**	**1733.333**	**242.321**
5	1000	216	1	998	998.000	1000.000	1.002
5	2000	216	1	1991	1991.000	2000.000	1.005
5	3000	216	1	2768	2768.000	3000.000	1.084
5	4000	216	1	3810	3819.000	4000.000	1.050
5	5000	216	1	4837	4837.000	5000.000	1.034
Avg.	**3000**	**216**	**1.0**	**2880.8**	**2882.600**	**3000.000**	**1.035**

ND = Number of documents, **NT** = Number of terms, **NC** = Number of clusters, **NSC** = Number of sub-clusters, **ANSCPC** = Average number of sub-clusters per cluster, **ANDPC** = Average number of documents per cluster, **ANDPSC** = Average number of documents per sub-cluster, **Avg.** = Average

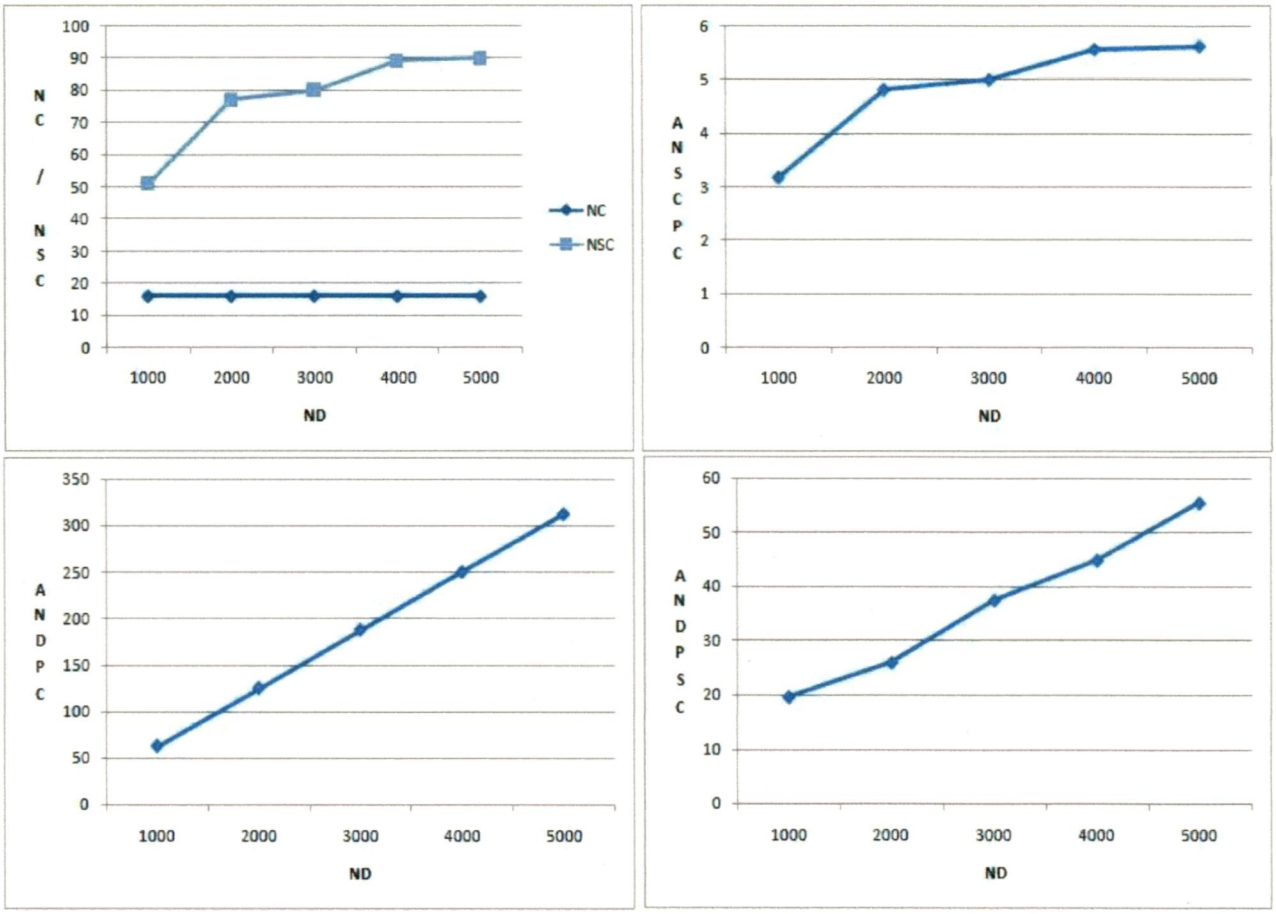


Fig. 5.1 Graphs for Case 1 data

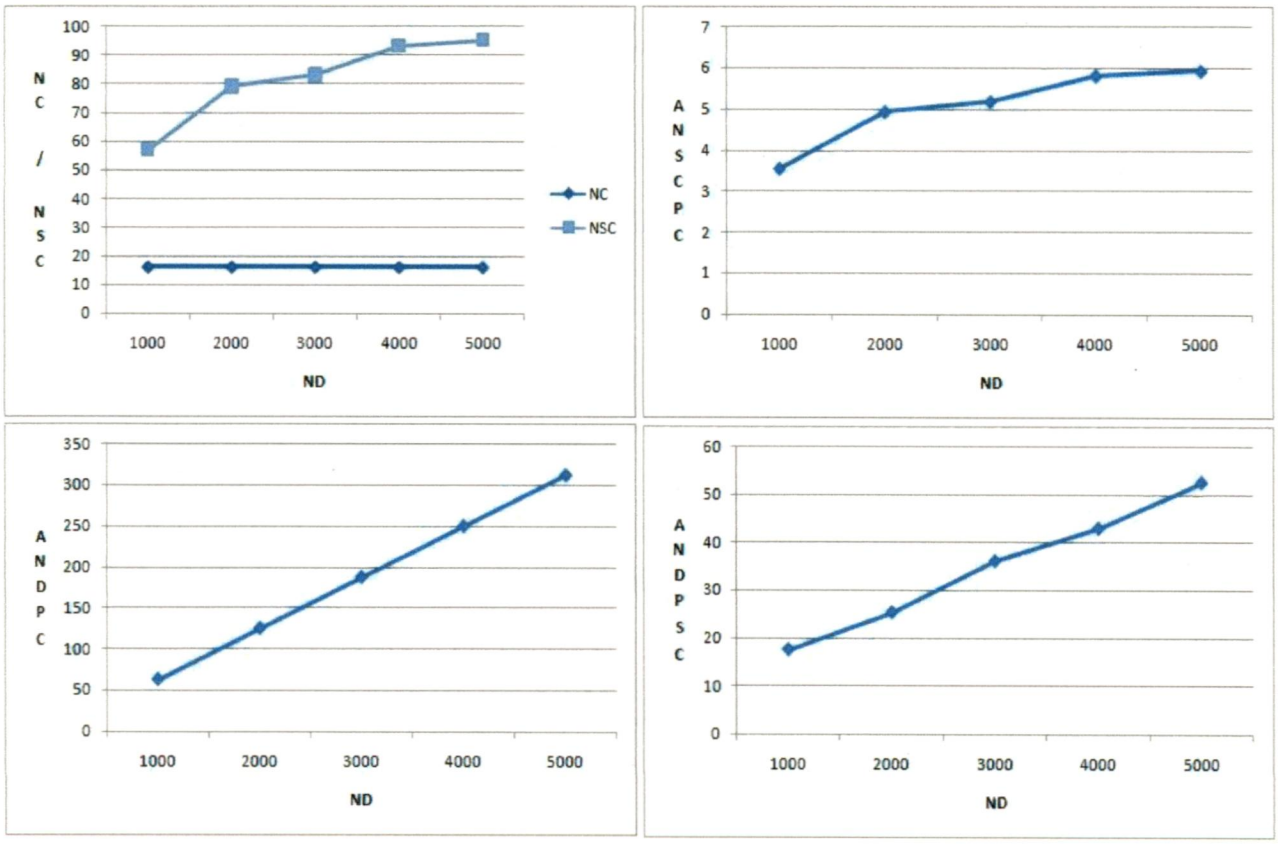


Fig. 5.2 Graphs for Case 2 data

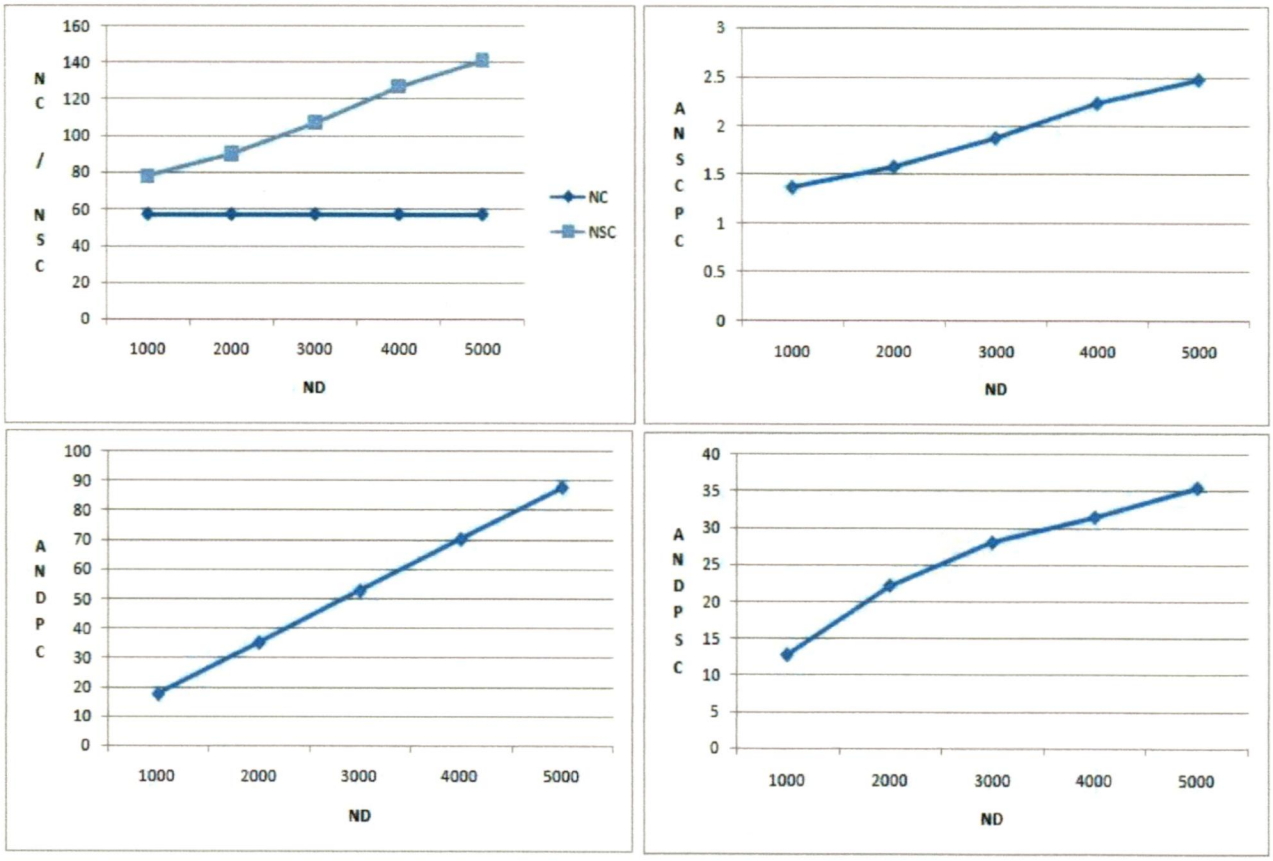


Fig. 5.3 Graphs for Case 3 data (Part I)

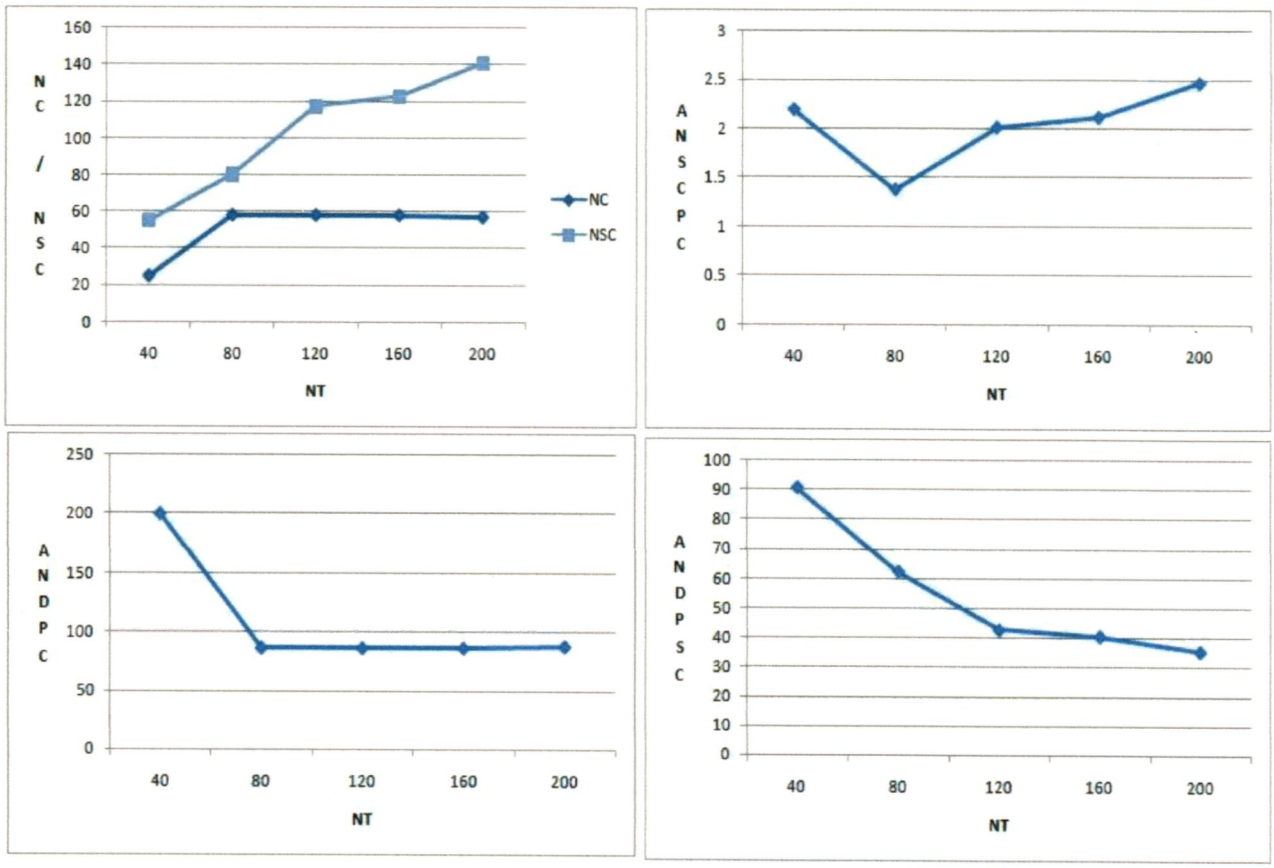


Fig. 5.4 Graphs for Case 3 data (Part II)

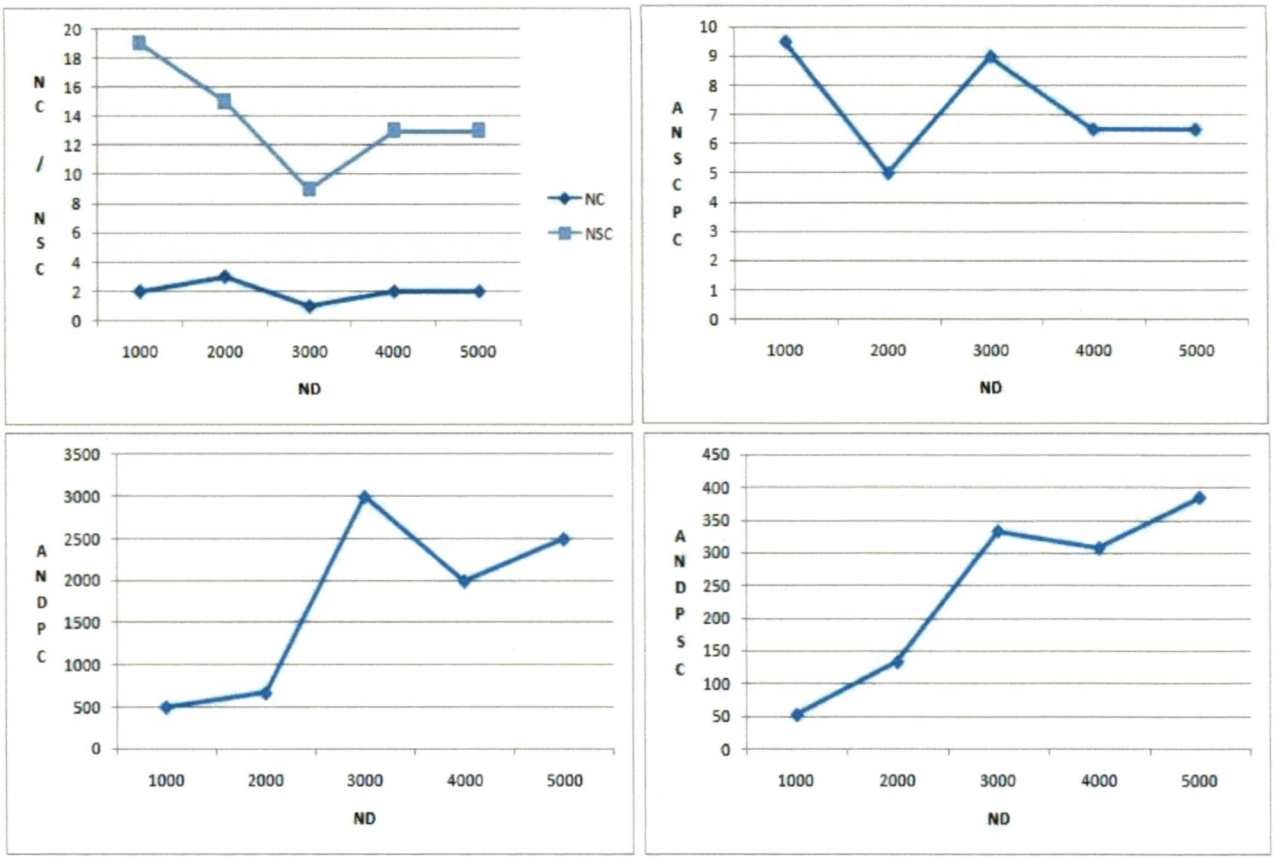


Fig. 5.5 Graphs for Case 4 data (ADA)

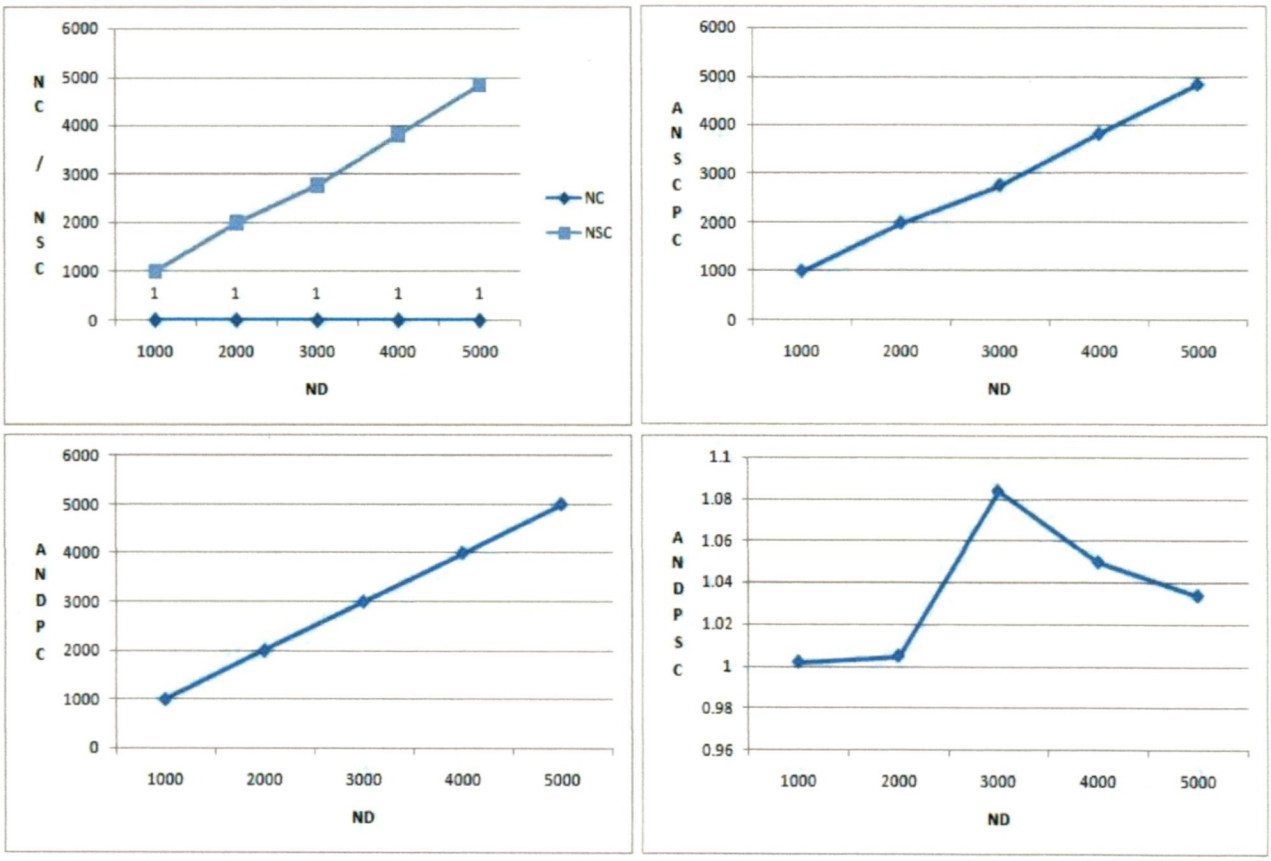


Fig. 5.6 Graphs for Case 5 data (SYLVA)

Now that we have presented the graphs, we will explain the findings. To begin with, we observe that for each of the datasets 1, 2, and 3 (Part I), the number of clusters does not vary with the number of documents (resulting in the steady increase of ANDPC). This is due to the fact that the document clustering is a two stage process: the first being clustering of the terms, and the second being the assignment of the document to the term-cluster with the highest corresponding TF-IDF mean. So, if the number of terms is kept constant, the number of document clusters will not vary with the number of the documents. This has the great advantage of managing the large corpus with a reasonable number of clusters (since number of terms << number of documents). It is also a logical conclusion of the fact that for a reasonably large document database, unless the dictionary is expanded, the number of document categories will not change. The number of clusters detected in Cases 1 and 2 strongly prove that our naïve Bayesian assumption works well. The difference between these two datasets was that there were terms overlapping with more than one well-formed cluster, strongly with one and weakly with the others. There were also stray distracting terms which did not form a cluster of their own but tried to destabilize the structure of well-formed clusters. Otherwise, the well-formed term-clusters were the same in both these datasets. Our scheme has been successful in nullifying the effect of the stray terms (also evident in Cases 4 and 5 analyzed later) and also in uniquely assigning overlapping terms to the cluster with whose terms which it had the strongest co-occurrence.

To demonstrate the effect of a change in the number of terms, we have varied the number of terms from forty through two hundred keeping the number of documents fixed at five thousand for the dataset of Case 3 (referring to this as Part II in Figure 5.4 and to the normal operation as Part I in Figure 5.3). The results then display a change in the number of clusters initially, but later become almost constant. But simply the number of clusters does not reveal the full picture here. We observe that the numbers of clusters are 25, 58, 58, 58, and 57 when the numbers of features are 40, 60, 80, 120, and 160 respectively. But initially the 58 clusters all contained only one or two terms each. We had mostly single-term clusters of trivial real-world use. As the number of terms grew, the clusters became meaningful, and began to contain reasonable numbers of terms like three to six. For space constraints, we are

not able to provide the number of terms in each term-cluster formed or the number of documents in each document cluster; otherwise this behavior would have been apparent. With the increase in the number of terms, the number of associated shape patterns within term-clusters also increase, increasing steadily the sub-cluster count. But since the number of documents is kept constant, the average number of documents per sub-cluster decreases monotonically, though at a very slow rate. This is because the rate of increase in the number of sub-clusters (due to the appearance of new shape patterns) is less than the rate at which new terms are added. But this step is done only as a demonstration, as increasing the number of terms while keeping the number of documents constant does not have much significance in real life, whereas the reverse is the case in most text clustering applications like organizing documents for a news agency or for a research conference.

Adding new documents incrementally (keeping the number of terms constant) results in the appearance of new shape patterns within the existing clusters. As a result, we observe the trend of an increasing NSC, ANSCPC, and ANDPSC with an increase in the number of documents for each of the datasets 1 through 3 (Part I).

Coming to the special datasets, we observe that although there were minor deviations, the average number of clusters detected for Case 4 data was two. This confirms our prior knowledge about the dataset. Case 5 data (SYLVA) was found to have only one cluster, again, as known earlier. This confirms that our algorithm is capable of detecting true clusters from large datasets even when a large number of the terms are distractors (having stray non-zero values) and the actual number of clusters is as low as one or two. For both cases, as the number of clusters is low, ANDPC is very high. For Case 5, since the number of terms in the special clusters is much higher than normal, the associated number of shape patterns that it may give rise to is also very high ($3^{Number\ of\ terms\ in\ term\ cluster}$, refer to Section 4.5). As a result, we have a very high NSC and very low ANDPSC. But the notion of sub-clusters does not have much significance for these two cases.

P. T. O.

5.4 Comparison of Running Times

All traditional text clustering algorithms (*k*-means, EM, farthest-first, and density-based) require the number of desired clusters as user input. But our clustering scheme does not require any user input or domain knowledge. It determines the inherent clusters present within the documents based on semantically linked terms. There is also no sub-clustering feature available in standard algorithms. As a result, we have adopted running time to be the main performance metric between our scheme (level I) and the standard algorithms (available in WEKA [22]). A screenshot of the WEKA clustering tool is shown in Figure 5.7. Both of the systems have been run on the same Java platform (with the number of clusters detected by our system as the input to the standard algorithms). The time required by a program running on a Java platform is computed easily by the NetBeans CPU profiler. We provide such a snapshot in Figure 5.8. These results are tabulated in Table 5.2 (legend at the bottom).

From Table 5.2 (especially the shaded regions) and the associated bar charts (Figures 5.9 through 5.12), we can easily see that our algorithm's average running time is significantly better than the standard algorithms for the same number of clusters detected. This is because all the standard algorithms tend to find clusters on a global basis, treating the entire document vector as a unit entity. As a result, they have to constantly deal with vectors of a very high dimensionality. Our algorithm tries to find local entities (term-clusters) within the term-set first and then clusters the documents on the basis of these local entities. Thus we look at local entities preserving the global structure of the document vector. The running times of the standard algorithms depend greatly on the number of desired clusters. As a result, when the number of clusters is known to be extremely low, they provide results in a very quick time. This explains their really low running times in Cases 4 and 5 (consequently, these values will not be visible in the corresponding bar charts and hence are not shown). But since our scheme does not assume any prior knowledge about the number of clusters, it has to proceed in its usual algorithm for all data, explaining the general trend of rising running time with the increase in the number of documents and terms. In general, the comparison was fair as both were run on the same Java platform. For space constraints, it has not been possible to include the number of documents in each cluster separately, or which documents were put into which cluster, for each algorithm.

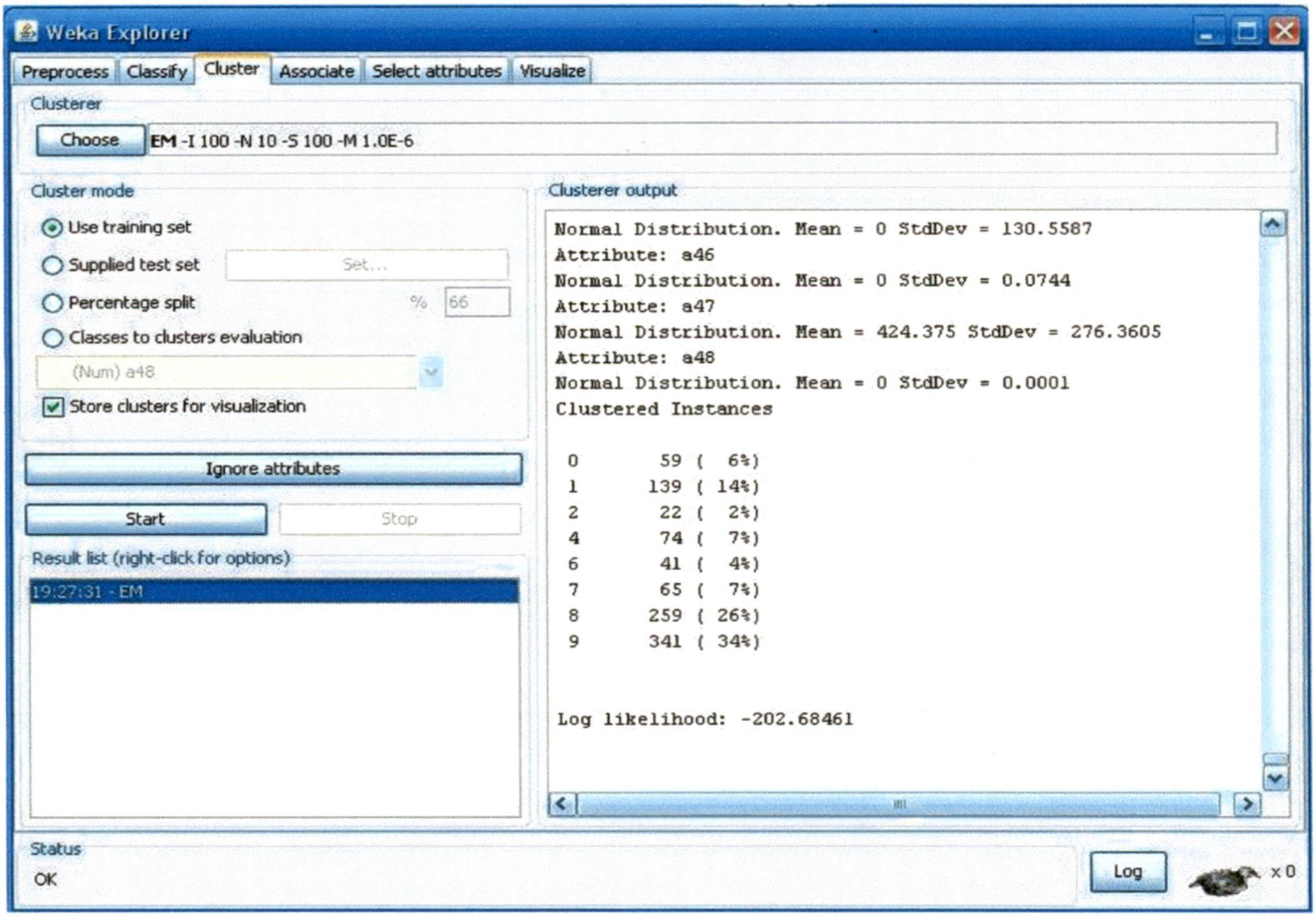


Fig. 5.7 Snapshot of the WEKA clustering tool

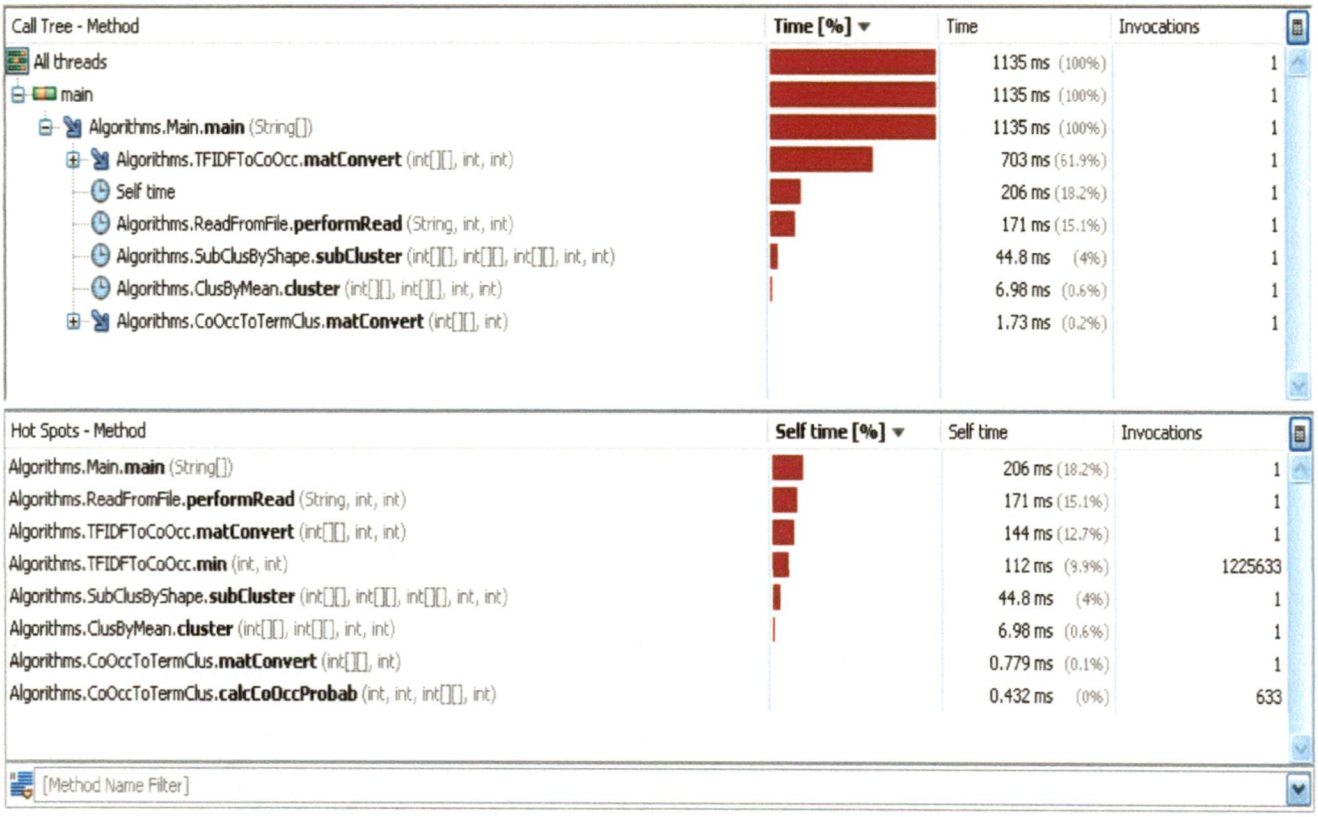


Fig. 5.8 Snapshot of the NetBeans CPU profiler

Table 5.2 Comparison of running times (in milliseconds)

Case	ND	NT	NC	Proposed scheme	Simple *k*-means	EM	Density-based	Farthest-first
1	1000	50	16	927	1016	952	971	1007
1	2000	50	16	2177	2691	2291	2347	2545
1	3000	50	16	3298	3849	3458	3511	3801
1	4000	50	16	4472	5055	4498	4527	4888
1	5000	50	16	5653	6316	5723	5911	6089
Avg.	**3000**	**50**	**16.0**	**3305.4**	**3785.4**	**3384.4**	**3453.4**	**3666.0**
2	1000	50	16	1135	1475	1913	1698	3959
2	2000	50	16	2261	3016	2453	2782	2946
2	3000	50	16	3362	3875	4092	3045	3631
2	4000	50	16	4682	4790	4699	4850	4751
2	5000	50	16	5775	5905	5604	5811	5764
Avg.	**3000**	**50**	**16.0**	**3443.6**	**3812.2**	**3752.2**	**3637.2**	**4210.2**
3	1000	200	57	12645	13789	12680	13003	14804
3	2000	200	57	27341	30067	28394	29561	29872
3	3000	200	57	42091	49007	44509	46712	47222
3	4000	200	57	57802	69691	59012	63423	61571
3	5000	200	57	72524	89880	74789	77820	71453
Avg.	**3000**	**200**	**57.0**	**42480.6**	**50486.8**	**43876.8**	**46103.8**	**44984.4**
3	5000	40	25	4222	6169	4357	4562	5982
3	5000	80	58	13046	14849	12971	13428	13991
3	5000	120	58	28286	29238	29264	28327	30320
3	5000	160	58	47172	50550	48954	48932	50113
3	5000	200	57	72524	80891	81823	82341	79256
Avg.	**5000**	**120**	**51.2**	**33050.0**	**36339.4**	**35473.8**	**35518.0**	**35932.4**
4	1000	48	2	1115	17	16	17	17
4	2000	48	3	1877	16	16	17	17
4	3000	48	1	3036	16	16	18	16
4	4000	48	2	4061	17	16	17	16
4	5000	48	2	5723	16	16	16	17
Avg.	**3000**	**48**	**2.0**	**3162.4**	**16.4**	**16.0**	**17.0**	**16.6**
5	1000	216	1	20313	16	18	17	18
5	2000	216	1	39360	17	16	17	17
5	3000	216	1	58224	16	16	17	17
5	4000	216	1	77844	16	18	17	17
5	5000	216	1	95383	16	16	16	17
Avg.	**3000**	**216**	**1.0**	**58224.8**	**16.2**	**16.8**	**16.8**	**17.2**

ND = Number of documents, **NT** = Number of terms, **NC** = Number of clusters, **EM** = Expectation Maximization, **Avg.** = Average

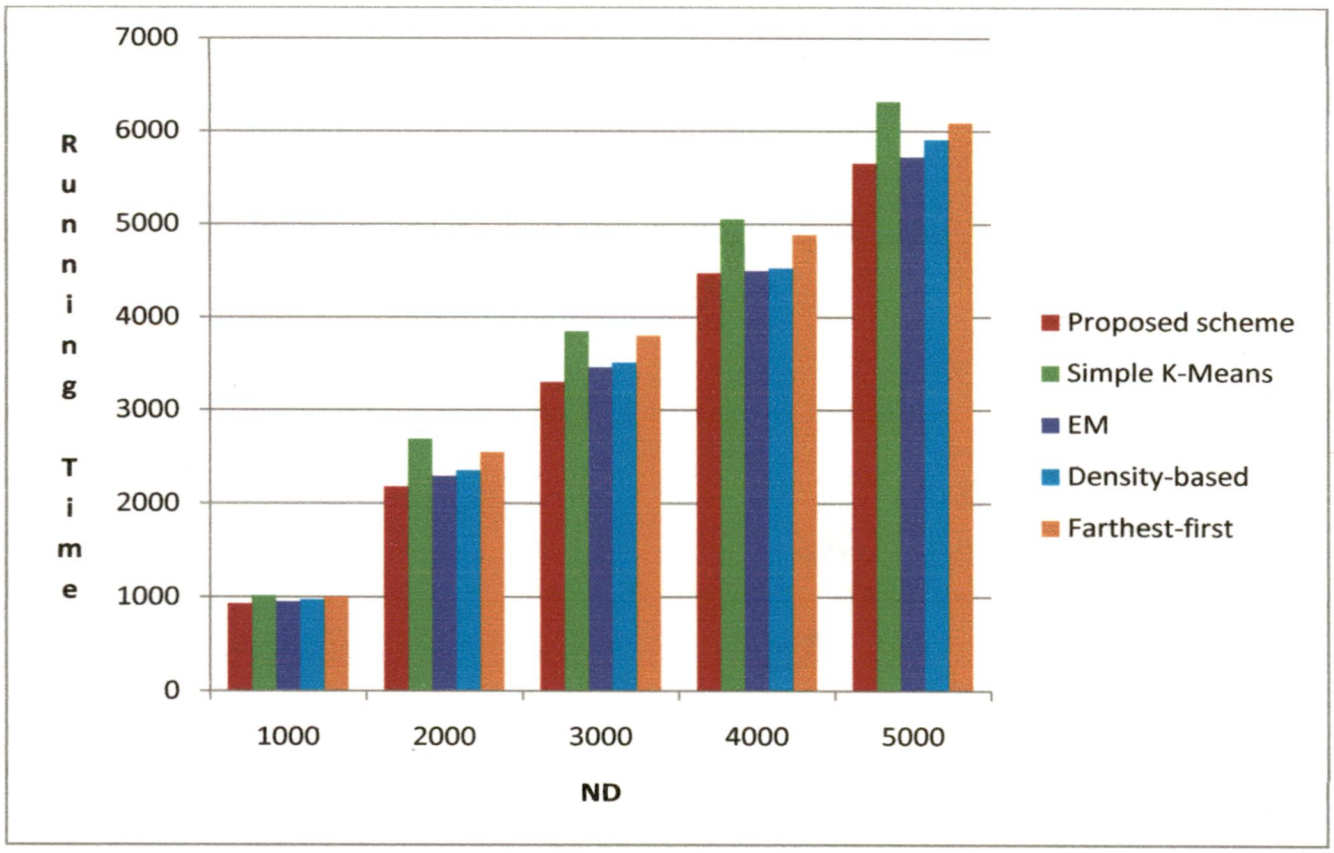


Fig. 5.9 Running times for Case 1 data (in ms)

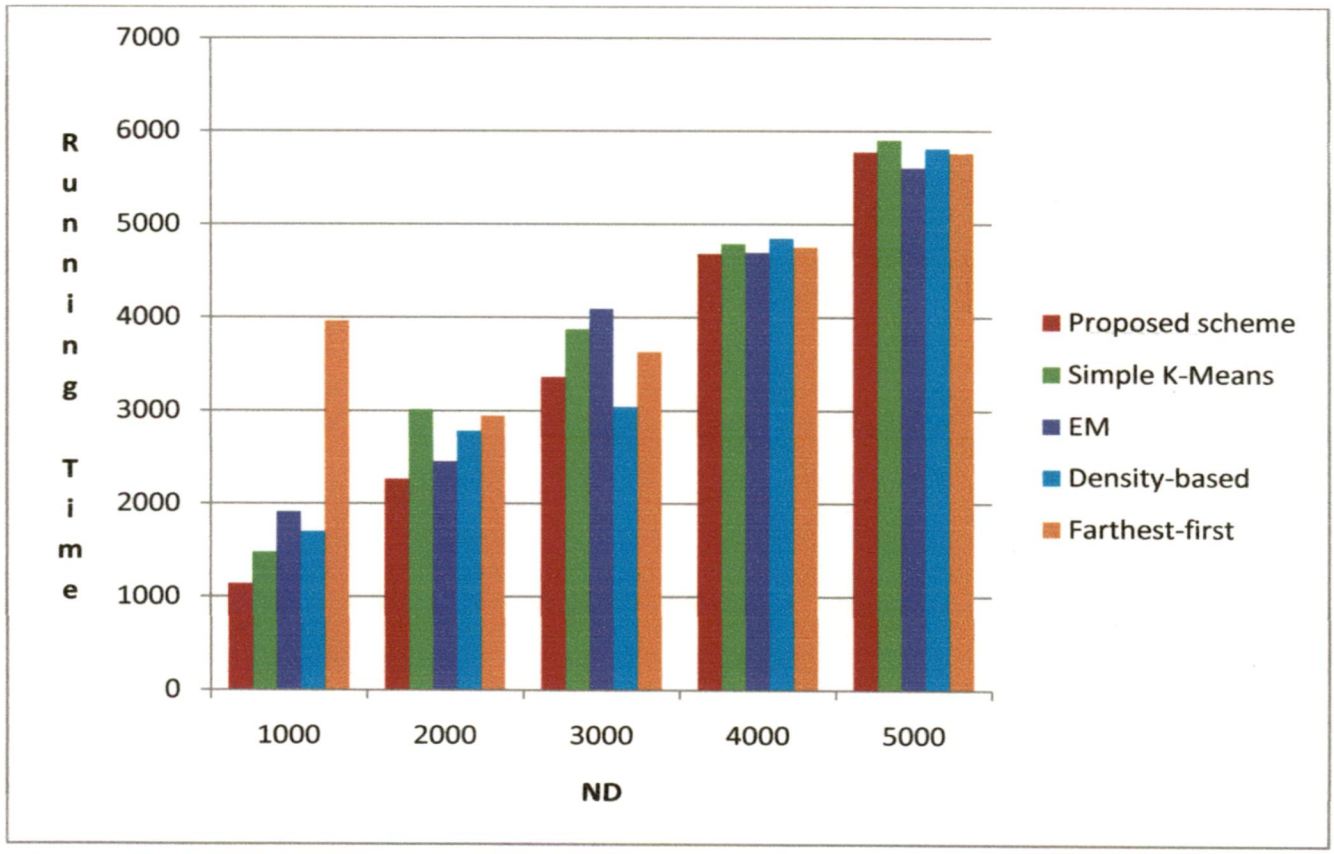


Fig. 5.10 Running times for Case 2 data (in ms)

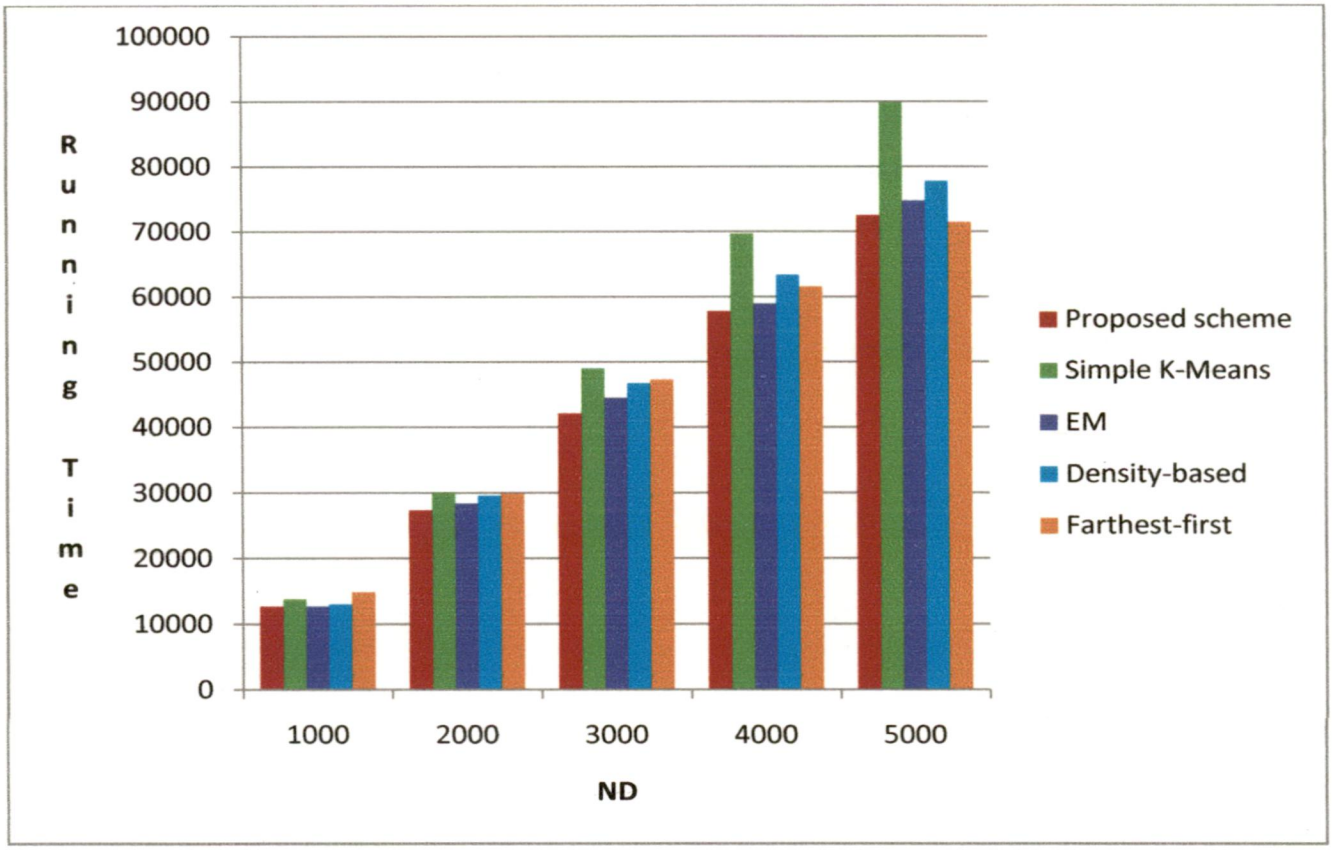


Fig. 5.11 Running times for Case 3 data (Part I) (in ms)

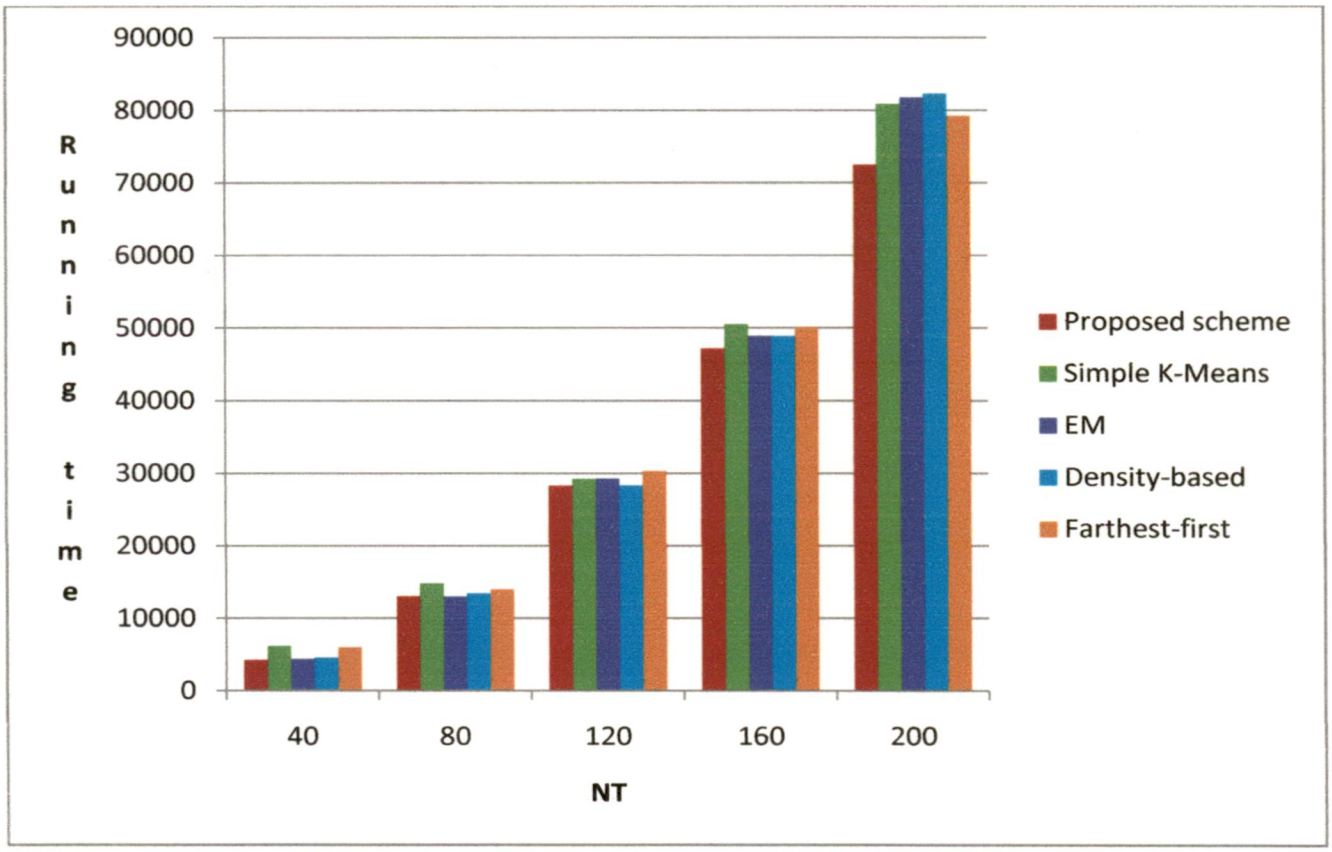


Fig. 5.12 Running times for Case 3 data (Part II) (in ms)

5.5 Analysis of Time Complexity

In this section, we will give the time complexity of our algorithm in the document clustering and the document sub-clustering levels. Let the number of documents and terms be m and n respectively. The approximate total running of our algorithm in the document clustering level, $T(m, n)$, is $O(mn^2)$, in the best, the worst, and the average cases. So the proposed algorithm in this level has a running time which varies linearly with the number of documents and quadratically with the number of terms, for all the three cases.

For the sub-clustering level, the analysis is performed relative to a single cluster as it is a process associated with each cluster independently. Let there be p documents and q terms in the cluster. Let the total approximate running time be denoted by $T(p, q)$. Then $T(p, q)$ is $O(p)$ in the best case and $O(3^{q-1}p)$ in the worst and the average cases. So the proposed algorithm in the document-sub-clustering level has a running time which varies linearly with the number of documents p in the cluster in the best case, and with the order of $3^{q-1}p$ in the worst and the average cases.

Chapter 6
CONCLUSION AND FUTURE WORK

In this final chapter of our report, we present our conclusions and the scope for future work on this topic.

6.1 Conclusion

In this work, we have proposed a novel two-level text clustering method based on the naïve Bayesian concept and shape pattern matching. In the first level, clusters are detected in the document set. Unlike traditional clustering algorithms, we first proceed to cluster the term-set based on their co-occurrence in the dataset. When a term is found to co-occur non-trivially with terms of more than one cluster, we use the naïve Bayesian concept of conditional independence to assign the term uniquely to one of the clusters. The basis of this term-clustering operation is to bring out the underlying semantic linkages between the terms. The clustering of the documents is then performed on the basis of these term-clusters using simple arithmetic mean computations on the TF-IDF values corresponding to the various clusters. Knowledge of semantic relationships within the terms helps in producing better clusters qualitatively. The sparse matrix representation is used wherever possible to reduce memory usage, as most of the data matrices used for stepwise computational purposes are not densely populated. The document clusters provide us with a broad grouping of the documents. In the second level, we exploit shape pattern-based similarity to find sub-clusters within the document clusters. Shape patterns inherent in the document vectors reflect the relative importance of the terms present within the document. They are used as a discriminatory measure to group documents within a cluster such that documents within a sub-cluster have the same relative importance attached to their terms.

We performed an exhaustive comparison between the running times of our scheme and the traditional clustering algorithms available in WEKA. Our results show that the running time of our algorithm is significantly better than the others.

This is because all the standard algorithms tend to find clusters on a global basis, treating the entire document vector as a unit entity. As a result, they have to constantly deal with vectors of a very high dimensionality. Our algorithm tries to find local entities (term-clusters) within the term-set first and then clusters the documents on the basis of these local entities. Thus we look at local entities preserving the global structure of the document vector. It also detects the major clusters successfully in large datasets when a major number of the terms are of trivial importance, their stray non-zero values acting as distractors trying to destabilize the structure of well-formed clusters. Moreover, our clustering scheme does not require any user input or domain knowledge. It detects the inherent clusters present within the dataset based on semantically linked terms. The number of document clusters does not vary with the dataset size, as long as the term-set is kept fixed. This has a big advantage of managing a large corpus with a reasonable number of clusters (since number of terms << number of documents).This is demonstrated by our results. It is also a logical conclusion from the fact that if our initial dataset size is reasonably large, then if the dictionary is not expanded by adding new terms, new clusters whose documents are semantically linked are also less likely to be produced.

Our algorithm will be computationally expensive and will not work well when there is a large degree of co-occurrence between the terms, causing terms to be candidates for almost every initial term-cluster. But in these situations, the structures of the clusters are not well-defined; and as such any clustering algorithm would produce poor results.

6.2 Future Work

We conclude this report with suggestions for future work on this topic. This work ends with the detection of the clusters. As future work, we may devise efficient indexing methods that would allow us to store the cluster information and retrieve details relevant to a few clusters only (which we may want to work with). This will highly contribute to the saving of computational space required, and subsequently in the scalability of the overall process for dealing with large document sets. As a possible drawback, the memory requirements of our algorithm are still somewhat

high, as we had a trade-off between space and time. This is a potential area of improvement. Documents may be allowed to belong to multiple clusters, sorted in a decreasing order of probability. Dependence among terms may be introduced, in which case the use of Bayesian belief networks has to be made. Improving upon our design in which we increase the number of documents manually keeping the number of terms constant, a system may be designed which will dynamically adapt itself when new documents and terms are added automatically from a data source. This would make the system capable of dealing with incremental or streaming data.

The concept of shape pattern-based similarity may be applied to other text mining operations. We may also introduce more precision if we analyze a single shape pattern further by its gradients. For example, the transition 'up' can be made more specific by introducing 'increasing', 'slowly increasing', and 'quickly increasing'. This would simply mean introducing gradient thresholds before the determination of the shapes.

REFERENCES

[1] J. Han, M. Kamber, "Data Mining: Concepts and Techniques", *Second Edition, Elsevier Inc., Rajkamal Electric Press,* 2006, pp. 1-628.

[2] L. Yanjun, L. Congnan, S. M. Chung, "Text Clustering with Feature Selection by Using Statistical Data", *IEEE Transactions on Knowledge and Data Engineering, IEEE Journal, Volume 20, Issue 5,* May 2008, pp. 641-652.

[3] J. B. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations", *Proc. Fifth Berkeley Symposium on Mathematical Statistics and Probability, University of California Press, 1967, Volume 1,* pp. 281-297.

[4] L. Xinwu, "Research on Text Clustering Algorithm Based on K_means and SOM", *International Symposium on Intelligent Information Technology Application Workshops 2008, IITAW 2008,* 21-22 Dec. 2008, pp. 341-344.

[5] T. Kohonen, "The Self-organizing Map", *International Journal of Neurocomputing, Elsevier Science B. V., Volume 21, Issues 1-3,* 6 Nov. 1998, pp. 1-6.

[6] X. Liu, P. He, H. Wang, "The Research of Text Clustering Algorithms Based on Frequent Term Sets", *Proc. 2005 International Conference on Machine Learning and Cybernetics 2005, Volume 4,* 18-21 Aug., 2005, pp. 2352-2356.

[7] I. Rish, "An Empirical Study of the Naïve Bayes Classifier", *Proc. Seventeenth International Joint Conference on Artificial Intelligence, 2001, IJCAI '01,* 4-10 Aug. 2001, pp. 1-7.

[8] B. Wang, S. Zhang, "A Novel Text Classification Algorithm based on Naïve Bayes and KL-divergence", *Proc. Sixth International Conference on Parallel*

and Distributed Computing, Applications, and Technologies, 2005, PDCAT 2005, 5-8 Dec. 2005, pp. 913-915.

[9] I.Z. Batyrshinl, L.B. Sheremetova, "Perception-based Approach to Time Series Data Mining", *A Research Program in Applied Mathematics and Computing (PIMAyC), Forging the Frontiers - Soft Computing, Applied Soft Computing, Mexican Petroleum Institute, Mexico, Volume 8, Issue 3,* Jun. 2008, pp. 1211-1221.

[10] A. Bagnall, C. A. Ratanamahatana, E. Keogh, S. Lonardi, G. Janacek, "A Bit Level Representation for Time Series Data Mining with Shape Based Similarity", *Data Mining and Knowledge Discovery (DMKD) Journal, Springer Netherlands, Volume 13, Number 1,* Jul. 2006, pp. 11-40.

[11] Y. Weng, Z. Zhu, "Time Series Clustering Based on Shape Dynamic Time Warping Using Cloud Models", *Proc. International Conference on Machine Learning and Cybernetics 2003, Volume 1,* 2-5 Nov., 2003, pp. 236-241.

[12] X. Junling, X. Baowen, Z. Weifeng, C. Zifeng, Z. Wei, "A New Feature Selection Method for Text Clustering", *Wuhan University Journal of Natural Sciences 2007 (WUJNS 2007), Volume 12, No. 5,* 2007, pp. 921-916.

[13] X. He, D. Cai, H. Liu, W.-Y. Ma, "Locality Preserving Indexing for Document Representation", *Proc. Twenty-seventh Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 2004, SIGIR 2004,* 25-29 Jul. 2004, pp. 96-103.

[14] G. Salton, A. Wong, C. S. Yang, "A Vector Space Model for Automatic Indexing", *Communications of the ACM, Volume 18, Issue 11,* Nov. 1975, pp. 613-620.

[15] M. T. Heath, "Sparse Matrix Computations", *Proc. Twenty-third IEEE Conference on Decision and Control, 1984, Volume 23, Part 1,* Dec. 1984, pp. 662-665.

[16] K. W. Church, P. Hanks, "Word Association Norms, Mutual Information and Lexicography", *Computational Linguistics, Vol.6, No.1,* 1990, pp. 22-29.

[17] English stopwords, http://www.webconfs.com/stop-words.php

[18] C.J. Rijsbergen, S.E. Robertson, M.F. Porter, "An Algorithm for Suffix Stripping", *Readings in Information Retrieval, Morgan Kaufmann Multimedia Information and Systems Series,* 1997, pp. 313-316.

[19] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, R. Harshman, "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science, 1990, ASIS 1990, Volume 41, Issue 6,* 1990, pp. 391 - 407.

[20] T. Hofmann, "Probabilistic Latent Semantic Indexing", *Proc. Twenty-second Annual International SIGIR Conference on Research and Development in Information Retrieval, SIGIR 1999,* 15-19 Aug. 1999, pp. 50-57.

[21] I. R. Silva, J. N. Souza, K. S. Santos, "Dependence among Terms in Vector Space Model", *Proc. International Database Engineering and Applications Symposium 2004 (IDEAS 2004),* 7-9 Jul. 2004, pp. 97-102.

[22] WEKA, http://www.cs.waikato.ac.nz/ml/weka/

[23] National Science Foundation (NSF), http://www.modelselect.inf.ethz.ch/

[24] I. Guyon, A. Saffari, G. Dror, G. Cawley, "Analysis of the IJCNN 2007 Agnostic Learning versus Prior Knowledge Challenge", *Proc. International Joint Conference on Neural Networks, 2007, IJCNN 2007, Volume 21, No. 2-3,* 2008, pp. 544-550.

[25] NetBeans IDE, http://www.netbeans.org/

LIST OF PUBLICATIONS

[1] **Rishiraj Saha Roy** and Durga Toshniwal, "Using Shape Patterns for Clustering Unstructured Text Documents", *Proc. 18th International Conference on Software Engineering and Data Engineering 2009 (SEDE 2009)*, 22-24 Jun. 2009, Las Vegas, Nevada, USA. (**Paper accepted** for presentation and publication in the conference proceedings; Conference proceedings will be indexed in INSPEC and DBLP)

[2] **Rishiraj Saha Roy** and Durga Toshniwal, "A Hierarchical Clustering Scheme for Unstructured Text Data", *Proc. 2009 International Conference on Information and Knowledge Engineering (IKE 2009)*, 13-16 Jul. 2009, The 2009 World Congress in Computer Science, Computer Engineering and Applied Computing, WORLDCOMP 2009, Las Vegas, Nevada, USA. (**Paper accepted** for presentation and publication in the conference proceedings; Conference proceedings will be indexed in INSPEC, DBLP, and IET)

[3] **Rishiraj Saha Roy** and Durga Toshniwal, "A Two-level Text Clustering Scheme using Naïve Bayesian Concept and Shape Pattern Matching", *Special Issue on "Knowledge Engineering and Management for the Intelligent Enterprise", International Journal of Knowledge Engineering and Data Mining (IJKEDM)*, Inderscience Publishers, Seventh Framework Programme, 2009. (Paper submitted and currently under review)

APPENDIX A:

SOURCE CODE LISTING

Main.java

```
package Algorithms;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main
{
   public static void main(String[] args) throws IOException
   {

      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
      System.out.println("\nWelcome to the project A TWO-LEVEL TEXT CLUSTERING SCHEME
BASED ON NAIVE BAYESIAN CONCEPT AND SHAPE PATTERN MATCHING.");
      System.out.println("\nPress ENTER to BEGIN...");
      int NumDocs=0;  // No. of documents
      int NumFeats=0; // No. of features

      int[][] TFIDFMat = new int[NumDocs][NumFeats];

      String str=new String();

//    Datasets used for validation

//    str = "example.txt";

//    str = "dataset1.txt";
//    str = "dataset2.txt";

//    str = "dataset3_1000_50NC.txt";
//    str = "dataset3_2000_50NC.txt";
//    str = "dataset3_3000_50NC.txt";
//    str = "dataset3_4000_50NC.txt";
//    str = "dataset3_5000_50NC.txt";

//    str = "dataset4_1000_50C.txt";
//    str = "dataset4_2000_50C.txt";
//    str = "dataset4_3000_50C.txt";
//    str = "dataset4_4000_50C.txt";
//    str = "dataset4_5000_50C.txt";

//    str = "dataset5_1000_200C.txt";
//    str = "dataset5_2000_200C.txt";
//    str = "dataset5_3000_200C.txt";
//    str = "dataset5_4000_200C.txt";
//    str = "dataset5_5000_200C.txt";

//    str = "dataset5_5000_40C.txt";
//    str = "dataset5_5000_80C.txt";
//    str = "dataset5_5000_120C.txt";
```

```
//      str = "dataset5_5000_160C.txt";
//      str = "dataset5_5000_200C.txt";

//      str = "ada_train_1000.txt";
//      str = "ada_train_2000.txt";
//      str = "ada_train_3000.txt";
//      str = "ada_train_4000.txt";
//      str = "ada_train_5000.txt";

//      str = "sylva_train_1000.txt";
//      str = "sylva_train_2000.txt";
//      str = "sylva_train_3000.txt";
//      str = "sylva_train_4000.txt";
        str = "sylva_train_5000.txt";

        FileInputStream fin=new FileInputStream(str);
        int index=0;            // Will read characters from file
        int ctr=0;              // Will count the number of commas
        boolean flag=false;     // Checks whether NumFeats has been determined
        while(true)
        {
           index = fin.read(); // Read one character
           if(index == -1)     // EOF encountered
              break;
           else if(index == ',')
              ctr++;           // Counting number of commas helps us calculate
                               // NumDocs
           else if(index==10 || index==13) // LF or CR
           {
              if(flag == false)
              {
                 // Determination of NumFeats - required only once - first
                 // time
                 NumFeats = ctr + 1;
                 flag = true;
              }
           }
        }
        fin.close();            // Close input file
        NumDocs = ctr/(NumFeats - 1);
        ctr = 0;                // Reset ctr

        TFIDFMat = ReadFromFile.performRead(str, NumDocs, NumFeats);

        int[][] CoOccMat = new int[NumFeats][NumFeats];
        // Co-occurrence matrix; will contain information about the
        // co-occurrence of a pair of terms in a document
        CoOccMat = TFIDFToCoOcc.matConvert(TFIDFMat, NumDocs, NumFeats);

        int[][] FinalTermClusMat = new int[NumFeats + 1][NumFeats + 1];
        // In the worst case, all terms may lie in one cluster
        // So no. of columns has to be made NumFeats
        // Made efficient using sparse matrix

        // The +1s are done to accomodate the end-of-cluster markers

        FinalTermClusMat = CoOccToTermClus.matConvert(CoOccMat, NumFeats);
```

```
        int[][] DCM = new int[NumDocs][3];  // Document Cluster Matrix
        // Column 1 will store document id
        // Column 2 will store cluster no.
        // Column 3 will store sub-cluster no.

        DCM = ClusByMean.cluster(TFIDFMat, FinalTermClusMat, NumDocs, NumFeats);
        // Column 2 entries have been inserted

        DCM = SubClusByShape.subCluster(TFIDFMat, FinalTermClusMat, DCM, NumDocs,
NumFeats);
        // Column 3 entries have been inserted now

        System.out.println("COMPLETE CLUSTERING PROCESS COMPLETED.");

        System.out.println("\nOPERATION SUMMARY:");
        System.out.println("Name of feature extracted TFIDF file: " + str);
        System.out.println("Number of documents: " + NumDocs);
        System.out.println("Number of features: " + NumFeats);
        int NumClus=1;
        // No. of document clusters
        // No. of times value changes in 2nd column of DCM - NumClus is the
        // counter
        for(int i=0; i<NumDocs-1; i++)
        {
          if(DCM[i][1] != DCM[i+1][1])
            NumClus++;
        }
        System.out.println("Total number of clusters: " + NumClus);
        int NumSubClus=DCM[NumDocs-1][2] + 1;        // Numbering starts at 0
        System.out.println("Total number of sub-clusters (across all clusters): " + NumSubClus);
        System.out.println("Average number of sub-clusters per cluster: " +
((double)NumSubClus/(double)NumClus));
        double NDC=(double)NumDocs/(double)NumClus;    // No. of documents per
                                                       // cluster
        System.out.println("Average number of documents per cluster: " + NDC);
        double NDSC=(double)NumDocs/(double)NumSubClus; // No. of documents per
                                                       // cluster
        System.out.println("Average number of documents per sub-cluster: " + NDSC);

        System.out.println("\nTHIS DISSERTATION WORK HAS BEEN PROGRAMMED BY:");
        System.out.println("Rishiraj Saha Roy");
        System.out.println("M.Tech. I.T. (2nd Year)");
        System.out.println("Enrolment No.: 074708");
        System.out.println("Department of Electronics and Computer Engineering");
        System.out.println("Indian Institute of Technology Roorkee.");

        System.out.println("\nTHANK YOU!\n");

    }
}
```

ReadFromFile.java

```
package Algorithms;

import java.io.FileInputStream;
import java.io.IOException;
```

```
public class ReadFromFile
{

  public static int[][] performRead(String FileName, int NumRows, int NumCols) throws IOException
  {
    // NumRows is the no. of rows in the matrix into which data from file is
    // read into
    // NumCols is the no. of columns in the matrix into which data from file
    // is read into

    int[][] NewMat = new int[NumRows][NumCols];

    FileInputStream fin=new FileInputStream(FileName);

    int[] TwoChars=new int[2];     // Will store current and last
                                   // previously read characters from file
                                   // Necessary to distinguish between
                                   // single ASCII 10 and (ASCII 13 and 10)
                                   // as pair
    TwoChars[0] = 0;               // Initialization
    TwoChars[1] = 0;
    double temp=0.0;               // Stores temporary integer derived
    double j=0.0;                  // Used for 10's exponent
    boolean flag=true;             // Indicates when comma or end-of-line
                                   // is met
    int p=0;                       // Row index of NewMat
    int q=0;                       // Col index of NewMat

    outer:
    while(true) // Outer loop
    {
      temp = 0.0;
      while (true)
      {
        TwoChars[0] = TwoChars[1];
        TwoChars[1] = fin.read();    // Read one ASCII character
        // System.out.println(TwoChars[1] + "\n");
        // Useful for knowing ASCII values of non-printing (CR, LF,
        // etc.) and special characters like ','

        if (TwoChars[1] == -1)       // EOF encountered
        {
          break outer;   // Break with label
        }
        else if(TwoChars[1]==10 && TwoChars[0]==13)
        {
          // Number has already been evaluated due to preceding CR
          // Move on to next character
          continue;
        }
        else if (((char)TwoChars[1]==',') || (TwoChars[1]==13) || (TwoChars[0]!=13 &&
TwoChars[1]==10))
        // Next no. or end of line encountered
        // ASCII of 13 corresponds to a carriage return (CR) (ENTER)
        // ASCII of 10 corresponds to paragraph mark, often present in
        // documents instead of newline or ENTER (Line Feed LF)
        // So we have to evaluate number now
        {
```

```
                flag = false;
                break;
            }
            else if(TwoChars[1]>=48 && TwoChars[1]<=57) // character is a digit
            {
                TwoChars[1] = (TwoChars[1]) - '0';
            }

            temp = (temp * 10) + TwoChars[1];
            // Calculating integer extracted
        }
        if (flag == false)
        {
            NewMat[p][q] = (int)temp;
            q++;

            if (q % (NumCols) == 0)
            {
                p++;
                q = 0;
            }
        }

        flag = true;
    }

    fin.close();   // Close input file

    System.out.println("\nReading matrix from file..... Done.\n");

    return NewMat;
  }
}
```

TFIDFToCoOcc.java

```
package Algorithms;

public class TFIDFToCoOcc
{
  public static int[][] matConvert(int[][] TFIDFMat, int NumDocs, int NumFeats)
  {
    int[][] CoOccMat = new int[NumFeats][NumFeats];
    // Co-occurrence matrix; will contain information about the
    // co-occurrence of a pair of terms in a document
    // To avoid redundancy, storing of CoOccMat[i][j] is enough and we
    // need not store CoOccMat[j][i]; also, CoOccMat[i][i] is meaningless
    // As a result, CoOccMat is an upper triangular matrix
    // We fill the diagonal elements with -1 as markers to aid in future
    // computation
    for(int i=0; i<NumFeats; i++)
    {
      for(int j=0; j<NumFeats; j++)
      {
        if(i == j)
          CoOccMat[i][j] = -1;
      }
    }
```

```
      int MinVal=0; // Will store minimum of two compared values
      for(int i=0; i<NumFeats-1; i++)        // Column index 1 for TFIDFMat
      {
         for(int j=i+1; j<NumFeats; j++)     // Column index 2 for TFIDFMat
         {
            for(int k=0; k<NumDocs; k++)    // Row index for TFIDFMat
            {
               // Co-occurrence for a pair of terms
               // Minimum of 2 values taken from TFIDFMat
               MinVal = TFIDFToCoOcc.min(TFIDFMat[k][i], TFIDFMat[k][j]);
               CoOccMat[i][j] = CoOccMat[i][j] + MinVal;
            }
            MinVal = 0; // Reset MinVal
         }
      }
      return CoOccMat;
   }

   public static int min(int a, int b)
   {
      if(a <= b)
         return a;
      return b;
   }
}
```

CoOccToTermClus.java

```
package Algorithms;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class CoOccToTermClus
{
   public static int[][] matConvert(int[][] CoOccMat, int NumFeats) throws IOException
   {
      // Term and feature are used interchangeably
      int[][] IntlTermClusMat = new int[NumFeats][NumFeats];
      // Will store initial term clusters

      // To begin with, each term is a cluster centre, before observing
      // co-occurrence patterns

      for(int i=0; i<NumFeats; i++)
      {
         for(int j=0; j<NumFeats; j++)
         {
            if(CoOccMat[i][j] != 0)
            {
               IntlTermClusMat[i][j] = 1;
               IntlTermClusMat[j][i] = 1;
               // Term i belongs to initial cluster j and vice versa
            }
         }
      }
```

```
// Since we have avoided redundancy by using an upper triangular
// matrix, term i can occur only in clusters i or less, i.e., term 2
// can occur only in clusters 0, 1, or 2

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
// br may be used to halt output later for stepwise checking

int[][] TermClusMat = new int[NumFeats][NumFeats ];

// So now we will assign a term uniquely to a cluster using naive
// bayesian theory of probability
// The transpose of the IntlTermClusMat, or equivalently reading the
// IntlTermClusMat columnwise gives us the initial clusters that a
// particular term belongs to

boolean[] WhetherAssigned = new boolean[NumFeats];
for(int i=0; i<NumFeats; i++)  // Initialization
   WhetherAssigned[i] = false;
// Boolean array to store whether every feature has been assigned to
// a cluster
// After probability calculations, those features which do not co-occur
// with any other feature, are assigned to their own clusters

TermClusMat[0][0] = 1;  // Assumption: first term belongs to first
                        // cluster; value set
WhetherAssigned[0] = true;

double probab=0.0;     // Will store the individual probabilities
double ProbabProd=1.0; // Product of individual probabilities
double MaxProbab=0.0;  // Will store maximum conditional probability
                       // of term belonging to particular cluster
int MaxCluster=-1;     // Will store final cluster number of term
for(int i=0; i<NumFeats; i++)      // Column index for IntlTermClusMat
{
//    System.out.println("i loop entered for term " + i);
   MaxProbab = 0.0;              // Reset MaxProbab
   for(int j=0; j<NumFeats; j++)  // Row index for IntlTermClusMat
   {
//       System.out.println("j loop entered for cluster " + j);
      ProbabProd = 1.0;      // Reset ProbabProd
      if(IntlTermClusMat[j][i] == 1)
      {
         // Term i belongs to initial cluster j
         if(i==0 && j==0)
            break;         // term 0 already assigned to cluster 0

         // Now we have to calculate probability of term i belonging
         // to cluster j
         // According to the naive bayes theory, this is given by the
         // product of the individual probabilities of term i
         // co-occurring with each term of cluster j

         for(int k=0; k<NumFeats; k++)  // k is a column index
         {
//             System.out.println("k loop entered for term " + i + " with term " + k);
            if(IntlTermClusMat[j][k]==1 && i!=k)
            {
               // term i co-occurs with term k
```

```
                    // terms i and k are distinct
                    probab = CoOccToTermClus.calcCoOccProbab(i, k, CoOccMat, NumFeats);
//                      System.out.println("Term " + i + ", Term " + k + " co-occurrence probability: " +
probab);
//                  br.readLine();
//                  To halt output for checking
                    ProbabProd = ProbabProd * probab;
                }

            }
            if(ProbabProd > MaxProbab)
            {
              MaxProbab = ProbabProd;
              MaxCluster = j; // Cluster to which probability of
                        // belonging maximum
              WhetherAssigned[i] = true;
            }
//          System.out.println("ProbabProd for Term " + i + ", Cluster " + j + ": " + ProbabProd);
//          br.readLine();
            probab = 0.0;  // Reset probab
          }
        }
        if(WhetherAssigned[i]==true && i!=0)   // Term 0 already assigned
          TermClusMat[MaxCluster][i] = 1;
      }

      for(int i=0; i<NumFeats; i++)
      {
        if(WhetherAssigned[i] == false)
        {
          TermClusMat[i][i] = 1;
        }
      }

      int[][] FinalTermClusMat=new int[NumFeats + 1][NumFeats + 1];
      // Converting to bag-of-words representtion
      // The +1s are for end markers

      int RowPtr=0;      // Row index for FinalTermClusMat
      int ColPtr=0;      // Column index for FinalTermClusMat
      boolean flag=false; // Check whether any points exist in a particular
                  // initial cluster
      for(int i=0; i<NumFeats; i++)
      {
        ColPtr = 0;    // Reset ColPtr
        for(int j=0; j<NumFeats; j++)
        {
          if(TermClusMat[i][j] == 1)
          {
            FinalTermClusMat[RowPtr][ColPtr] = j;
            FinalTermClusMat[RowPtr][ColPtr+1] = -1;
            // End of current cluster
            FinalTermClusMat[RowPtr+1][0] = -1;
            // End of all clusters
            flag = true;
            ColPtr++;
          }
        }
```

```
        if(flag == true)
        {
          RowPtr++;
        }
        flag = false;
      }
      RowPtr = 0; // Reset RowPtr
      return FinalTermClusMat;
    }

    public static double calcCoOccProbab(int term1, int term2, int[][] CoOccMat, int NumFeats)
    {
      double probab=0.0;
      int MinVal=TFIDFToCoOcc.min(term1, term2);  // Will store lower numbered
                                                  // term
      int MaxVal=0;                               // Will store higher
                                                  // numbered term
      if(term1 == MinVal)
        MaxVal = term2;
      else
        MaxVal = term1;
      int numr=CoOccMat[MinVal][MaxVal];  // As it is an upper triangular
                                          // matrix, value will be found at
                                          // this location only
      // numr is the numerator term for the probability
      // No. of co-occurrences of term1 with term2
      int denr=0;
      // denr is the denominator term for the probability, hence not
      // initialized to 0
      // No. of co-occurrences of term1 with all other terms
      // Denominator calculation
      for(int i=0; i<term1; i++)
      {
        denr = denr + CoOccMat[i][term1];   // 0 entries (terms with which
                                            // term1 does not co-occur) do
                                            // not affect sum
      }
      for(int j=term1+1; j<NumFeats; j++)
      {
        denr = denr + CoOccMat[term1][j];   // 0 entries (terms with which
                                            // term1 does not co-occur) do
                                            // not affect sum
        // Does not add the -1 at (i, i) position
      }

      if(numr == 0)   // Laplacian correction
      {
        numr++;
        denr++;
      }
      if(denr != 0)
        probab = (double)numr / (double)denr;
      else
        probab = 0.0;

      return probab;
    }
}
```

ClusByMean.java

```
package Algorithms;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ClusByMean
{
  public static int[][] cluster(int[][] TFIDFMat, int[][] FinalTermClusMat, int NumDocs, int
NumFeats) throws IOException
  {
    int[][] DCM = new int[NumDocs][3];
    // Document Cluster Matrix
    // Column 1 will store document id
    // Column 2 will store cluster no.
    // Column 3 will store sub-cluster no.

    for(int i=0; i<NumDocs; i++)
    {
      DCM[i][0] = i;  // Filling in document id
                // Remaining 2 column values are not known yet
    }
    double mean=0.0;   // Will store mean TFIDF value for a cluster
    double MaxMean=0.0; // Will store max of these means to determine
                // final cluster
    int ctr=0;        // Will count number of terms in cluster for
                // division of sum
    int TermNo=0;      // Will store term no. of term in cluster
    int val=0;        // Will store required TFIDF value
    int sum=0;        // Will store sum of corresponding TFIDF values

    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    // br may be used to halt output later for stepwise checking

    for(int i=0; i<NumDocs; i++)
    {
      // Each document has to be put into its respective cluster
      for(int p=0; p<NumFeats; p++)        // p is row index for
                              // FinalTermClusMat, i.e.,
                              // cluster no.
      {
        if(FinalTermClusMat[p][0] == -1)   // End of all clusters
        {
          break;
          // Cannot be encountered at the very beginning in case of
          // valid FinalTermClusMat
          // Loop will compulsorily exit on break condition and not
          // on for loop completion
        }
        for(int q=0; q<NumFeats; q++)      // q is column index for
                              // FinalTermClusMat, i.e.,
                              // term no.
        {
          if(FinalTermClusMat[p][q] == -1)
          {
            break;                  // End of current cluster
```

```
                    // Cannot be encountered at the very beginning in case
                    // of valid FinalTermClusMat
                    // Loop will compulsorily exit on break condition and
                    // not on for loop completion
                }
                TermNo = FinalTermClusMat[p][q];
                ctr++;
                val = TFIDFMat[i][TermNo];
                sum = sum + val;
//                System.out.println("Sum: " + sum);
//                br.readLine();
            }
            if(ctr != 0)
            {
                mean = (double)sum / (double)ctr;
//                System.out.println("Mean: " + mean);
            }
            if(mean >= MaxMean)
            {
                MaxMean = mean;        // Update MaxMean
                DCM[i][1] = p;         // Update cluster no. of document
            }
            mean = 0.0;                // Reset mean after dealing with one
                                       // cluster
            sum = 0;                   // Reset sum after dealing with one
                                       // cluster
            ctr = 0;                   // Reset ctr after dealing with one
                                       // cluster
        }
        MaxMean = 0.0;                 // Reset MaxMean after dealing with
                                       // one document
//        br.readLine();
    }

    return DCM;
  }
}
```

SubClusByShape.java

```
package Algorithms;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class SubClusByShape
{
  public static int[][] subCluster(int[][] TFIDFMat, int[][] FinalTermClusMat, int[][] DCM, int
NumDocs,int NumFeats) throws IOException
  {

    int[][] IDCM=new int[NumDocs][5];  // Initial Document Cluster Matrix
                                       // Will contain number of terms in
                                       // respective cluster of document
                                       // in 3rd column
                                       // Will contain shape identifiers
```

```
                    // in the fourth column

// Copying necessary information to IDCM
for(int i=0; i<NumDocs; i++)
{
   IDCM[i][0] = DCM[i][0];
   IDCM[i][1] = DCM[i][1];
}

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
// br may be used to halt output later for stepwise checking

int temp0=0;   // Temporary variables for current and future use
int temp1=0;
int temp2=0;
// Sorting documents by cluster
for(int i=0; i<NumDocs; i++)
{
   for(int j=i; j<NumDocs; j++)
   {
      // Clusters in ascending order
      if(IDCM[i][1] >= IDCM[j][1])
      {
         // Swapping first column - document id
         temp0      = IDCM[i][0];
         IDCM[i][0]  = IDCM[j][0];
         IDCM[j][0]  = temp0;

         // Swapping second column - cluster id
         temp1      = IDCM[i][1];
         IDCM[i][1]  = IDCM[j][1];
         IDCM[j][1]  = temp1;
      }
   }
}

int ClusCtr=0;        // Counts the number of clusters
for(int i=0; i<NumFeats; i++)
{
   if(FinalTermClusMat[i][0] == -1)
      break;
   ClusCtr++;
}
int[] NTC=new int[ClusCtr]; // Number of Terms in Cluster
                    // Array will store number of terms in each
                    // cluster
// Counting number of terms in each cluster
int ctr=0;                  // Counter
for(int i=0; i<NumFeats; i++)      // i is row index - cluster number
{
   if(FinalTermClusMat[i][0] == -1)
      break;     // End of all clusters
   for(int j=0; j<NumFeats; j++)  // j is column index - term number
   {
      if(FinalTermClusMat[i][j] == -1)
         break; // End of current cluster
      ctr++;
   }
```

```
    NTC[i] = ctr;  // Assign ctr
    ctr = 0;       // Reset ctr
}
ctr = 0;           // Reset ctr for future use

String CurShape=new String();
CurShape="";       // Initialization
// Will store shape pattern of current document

String[] ShapeList=new String[NumDocs + 1];
// Will contain list of unique shapes, in worst case all documents
// may have different shapes to the varying length of their clusters
// +1 for end marker
// Hence the size of the array

ShapeList[0] = "end";  // End marker - end of all shapes
                       // Initially at the beginning - no shapes in
                       // list

int ClusNum=0;         // Will store cluster number of document
                       // being processed
int term1=0;           // Terms used to determine up, down, or level
int term2=0;
int val1=0;            // Corresponding term1, term2 values in
int val2=0;            // TFIDFMat
int ptr=0;             // Used as index to traverse ShapeList
// Main loop - shape processing for all documents
for(int i=0; i<NumDocs; i++)
{
    ClusNum = IDCM[i][1];
    IDCM[i][2] = NTC[ClusNum];
    if(IDCM[i][2] == 1) // 1 term in cluster
    {
        IDCM[i][3] = 0; // Shape pattern 0 corresponds to nil or no
                        // shape as there is only one term in cluster
                        // No further sub-cluster
    }
    else
    {
        int p=0;       // Column index for FinalTermClusMat

        while(true)
        {
            if(FinalTermClusMat[ClusNum][p+1] == -1)
            {
                break;
            }
            term1 = FinalTermClusMat[ClusNum][p];
            term2 = FinalTermClusMat[ClusNum][p+1];
            val1 = TFIDFMat[i][term1];
            val2 = TFIDFMat[i][term2];
            if(val1 < val2)
                CurShape = CurShape + "U";
            else if(val1 > val2)
                CurShape = CurShape + "D";
            else if(val1 == val2)
                CurShape = CurShape + "L";
```

```
            p++;
        }

        while(true)
        {
            // br.readLine();
            // Traversing ShapeList array to check whether uncovered
            // pattern new or already present in list
            if((ShapeList[ptr]).equals("end") == true)
            {
                // End of list encountered
                // Pattern is new
                ShapeList[ptr] = CurShape;
                IDCM[i][3] = ptr;
                ShapeList[ptr + 1] = "end"; // Push end marker by one
                break;
            }
            else if((ShapeList[ptr]).equals(CurShape) == true)
            {
                IDCM[i][3] = ptr;   // Insert column 3 value - shape
                                    // pattern id
                break;
            }
            ptr++;                  // Try next shape
        }

        ptr = 0;        // Reset ptr
        CurShape = "";  // Reset CurShape
    }
}

ptr = 0;

// Now we sort in ascending order of shape indices within clusters
int temp3=0;    // More temporary variables
int temp4=0;
for(int i=0; i<NumDocs; i++)
{
    for(int j=i; j<NumDocs; j++)
    {
        if((IDCM[i][1]==IDCM[j][1]) && (IDCM[i][3]>=IDCM[j][3]))
        {
            temp0 = IDCM[i][0];         // Swap document id
            IDCM[i][0] = IDCM[j][0];
            IDCM[j][0] = temp0;

            // No need to swap column 1 as it is equal by condition

            temp2 = IDCM[i][2];         // Swap number of terms in
            IDCM[i][2] = IDCM[j][2];    // cluster
            IDCM[j][2] = temp2;

            temp3 = IDCM[i][3];         // Swap shape id
            IDCM[i][3] = IDCM[j][3];
            IDCM[j][3] = temp3;

            // No need to swap fifth column as it is empty
        }
```

```
    }
}

// Sub-clustering starts
IDCM[0][4] = 0;  // Document 1 belongs to 1st sub-cluster (Number 0)
for(int i=1; i<NumDocs; i++)
{
    if((IDCM[i][3]==IDCM[i-1][3]) && (IDCM[i][1]==IDCM[i-1][1]))
    {
        // If shapes match and clusters match
        IDCM[i][4] = IDCM[i-1][4];  // Same sub-cluster
    }
    else if((IDCM[i][3]!=IDCM[i-1][3]) || (IDCM[i][1]!=IDCM[i-1][1]))
    {
        // If shapes do not match or clusters do not match
        IDCM[i][4] = IDCM[i-1][4] + 1;  // New sub-cluster
    }
}

// Sorting documents within a particular sub-cluster by document id
for(int i=0; i<NumDocs; i++)
{
    for(int j=i; j<NumDocs; j++)
    {
        // Documents in ascending order
        if((IDCM[i][0]>=IDCM[j][0]) && (IDCM[i][4]==IDCM[j][4]))
        {
            // Swapping first column - document id
            temp1      = IDCM[i][0];
            IDCM[i][0] = IDCM[j][0];
            IDCM[j][0] = temp1;

            // Swapping second column - cluster id
            temp2      = IDCM[i][1];
            IDCM[i][1] = IDCM[j][1];
            IDCM[j][1] = temp2;

            // Swapping third column - number of terms in cluster
            temp3      = IDCM[i][2];
            IDCM[i][2] = IDCM[j][2];
            IDCM[j][2] = temp3;

            // Swapping fourth column - shape id
            temp4      = IDCM[i][3];
            IDCM[i][3] = IDCM[j][3];
            IDCM[j][3] = temp4;

            // No need to swap fifth column as it is equal by condition
        }
    }
}

// Copying back necessary information before return to main
for(int i=0; i<NumDocs; i++)
{
    DCM[i][0] = IDCM[i][0];
    DCM[i][1] = IDCM[i][1];
    DCM[i][2] = IDCM[i][4];
```

```
        }

        return DCM;
    }
}
```

SparseMatrix.java

```
public class SparseMatrix extends AbstractMatrix {
    protected double[][] nzValues;

    protected int[][] columnIndices;

    protected int[] nzCounters;

    /**
     * @param colCount number of columns
     * @param rowCount number of rows
     */
    public SparseMatrix(int colCount, int rowCount) {
        super(colCount, rowCount);
        this.nzValues = new double[rowCount][];
        this.columnIndices = new int[rowCount][];
        this.nzCounters = new int[rowCount];
    }

    /**
     * Gets values at specified location
     *
     * @param column column index
     * @param row    row index
     * @return value
     */
    public double get(int column, int row) {
        if (this.columnIndices[row] == null) {
            return 0.0;
        }
        int columnIndex = binarySearch(this.columnIndices[row], 0,
                this.nzCounters[row] - 1, column);
        if (columnIndex < 0) {
            return 0.0;
        }
        return this.nzValues[row][columnIndex];
    }

    /**
     * Performs a binary search for a given value in sorted integer array. The
     * only difference from <b>Arrays.binarySearch()</b> is that this function
     * gets <i>start</i> and <i>end</i> indexes. The array <strong>must</strong>
     * be sorted prior to making this call.  If it is not sorted, the results
     * are undefined.
     *
     * @param array      array to scan
     * @param startIndex start index of sub-array
     * @param endIndex   end index of sub-array
     * @param value      key to find
```

```
     * @return index of the search key, if it is contained in the list;
     *         otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
     *         <i>insertion point</i> is defined as the point at which the key
     *         would be inserted into the list: the index of the first element
     *         greater than the key, or <tt>list.size()</tt>, if all elements in
     *         the list are less than the specified key.  Note that this
     *         guarantees that the return value will be &gt;= 0 if and only if
     *         the key is found.
     */
    private static int binarySearch(int[] array, int startIndex, int endIndex,
                                    int value) {
      if (value < array[startIndex]) {
        return (-startIndex - 1);
      }
      if (value > array[endIndex]) {
        return (-(endIndex + 1) - 1);
      }

      if (startIndex == endIndex) {
        if (array[startIndex] == value) {
          return startIndex;
        }
        else {
          return (-(startIndex + 1) - 1);
        }
      }
      int midIndex = (startIndex + endIndex) / 2;
      if (value == array[midIndex]) {
        return midIndex;
      }

      if (value < array[midIndex]) {
        return binarySearch(array, startIndex, midIndex - 1, value);
      }
      else {
        return binarySearch(array, midIndex + 1, endIndex, value);
      }
    }

    /**
     * Sets value at specified location
     *
     * @param column column index
     * @param row    row index
     * @param value  value
     */
    public void set(int column, int row, double value) {
      if (this.columnIndices[row] == null) {
        // first value in this row
        this.columnIndices[row] = new int[2];
        this.nzValues[row] = new double[2];
        this.columnIndices[row][0] = column;
        this.nzValues[row][0] = value;
        this.nzCounters[row] = 1;
        return;
      }

      // search for it
```

```
int columnIndex = binarySearch(this.columnIndices[row], 0,
      this.nzCounters[row] - 1, column);
if (columnIndex >= 0) {
   // already setLocation, just change
   this.nzValues[row][columnIndex] = value;
   return;
}
else {
   // columnIndex = (-(insertion point) - 1)
   int insertionPoint = -(columnIndex + 1);
   // allocate new arrays
   int oldLength = this.nzCounters[row];
   int newLength = oldLength + 1;
   // check if need to allocate
   if (newLength <= this.columnIndices[row].length) {
      // just copy
      if (insertionPoint != oldLength) {
         for (int i = oldLength; i > insertionPoint; i--) {
            this.nzValues[row][i] = this.nzValues[row][i - 1];
            this.columnIndices[row][i] =
                  this.columnIndices[row][i - 1];
         }
      }
      this.columnIndices[row][insertionPoint] = column;
      this.nzValues[row][insertionPoint] = value;
      this.nzCounters[row]++;
      return;
   }

   int[] newColumnIndices = new int[2 * oldLength];
   double[] newNzValues = new double[2 * oldLength];

   if (insertionPoint == oldLength) {
      // special case - new column is the last
      System.arraycopy(this.columnIndices[row], 0, newColumnIndices,
            0, oldLength);
      System.arraycopy(this.nzValues[row], 0, newNzValues, 0,
            oldLength);
   }
   else {
      System.arraycopy(this.columnIndices[row], 0, newColumnIndices,
            0, insertionPoint);
      System.arraycopy(this.nzValues[row], 0, newNzValues, 0,
            insertionPoint);
      System.arraycopy(this.columnIndices[row], insertionPoint,
            newColumnIndices, insertionPoint + 1, oldLength
            - insertionPoint);
      System.arraycopy(this.nzValues[row], insertionPoint,
            newNzValues, insertionPoint + 1, oldLength
            - insertionPoint);
   }
   newColumnIndices[insertionPoint] = column;
   newNzValues[insertionPoint] = value;
   this.columnIndices[row] = null;
   this.columnIndices[row] = newColumnIndices;
   this.nzValues[row] = null;
   this.nzValues[row] = newNzValues;
   this.nzCounters[row]++;
```

```
    }
  }

  /**
   * Dump to standard output
   */
  public void dump() {
    System.out.println("MATRIX " + this.rowCount + "*" + this.colCount);
    for (int row = 0; row < this.rowCount; row++) {
      int[] columnIndices = this.columnIndices[row];
      if (columnIndices == null) {
        for (int col = 0; col < this.colCount; col++) {
          System.out.print("0.0 ");
        }
      }
      else {
        int prevColumnIndex = 0;
        for (int colIndex = 0;
            colIndex < this.nzCounters[row]; colIndex++) {
          int currColumnIndex = columnIndices[colIndex];
          // put zeroes
          for (int col = prevColumnIndex;
              col < currColumnIndex; col++) {
            System.out.print("0.0 ");
          }
          System.out.print(this.nzValues[row][colIndex] + " ");
          prevColumnIndex = currColumnIndex + 1;
        }
        // put trailing zeroes
        for (int col = prevColumnIndex; col < this.colCount; col++) {
          System.out.print("0.0 ");
        }
      }

      System.out.println();
    }
  }

  /**
   * Dump to standard output as integer values
   */
  public void dumpInt() {
    System.out.println("MATRIX " + this.rowCount + "*" + this.colCount);
    for (int row = 0; row < this.rowCount; row++) {
      int[] columnIndices = this.columnIndices[row];
      if (columnIndices == null) {
        for (int col = 0; col < this.colCount; col++) {
          System.out.print("0 ");
        }
      }
      else {
        int prevColumnIndex = 0;
        for (int colIndex = 0;
            colIndex < this.nzCounters[row]; colIndex++) {
          int currColumnIndex = columnIndices[colIndex];
          // put zeroes
          for (int col = prevColumnIndex;
              col < currColumnIndex; col++) {
```

```
                System.out.print("0 ");
            }
            System.out.print((int) this.nzValues[row][colIndex] + " ");
            prevColumnIndex = currColumnIndex + 1;
        }
        // put trailing zeroes
        for (int col = prevColumnIndex; col < this.colCount; col++) {
            System.out.print("0 ");
        }
    }

    System.out.println();
  }
}

/**
 * Add empty (zero) columns to this matrix
 *
 * @param columns number of columns to add
 */
public void addEmptyColumns(int columns) {
  // just as easy as that
  this.colCount += columns;
}

/**
 * Multiply this matrix by the specified column of another matrix. The
 * operation is linear in terms of count of non-zero values in the matrix
 *
 * @param matrix the second matrix
 * @param column column index in the second matrix
 * @return vector result
 */
public double[] multiply(Matrix matrix, int column) {
  if (this.getColumnCount() != matrix.getRowCount()) {
    return null;
  }
  int n = this.getRowCount();
  double[] result = new double[n];
  for (int row = 0; row < n; row++) {
    double sum = 0.0;
    // go over all non-zero column of this row
    int[] nzIndexes = this.columnIndices[row];
    int nzLength = nzCounters[row];
    if (nzLength == 0) {
      continue;
    }
    for (int colIndex = 0; colIndex < nzLength; colIndex++) {
      double c = matrix.get(column, nzIndexes[colIndex]);
      sum += (this.nzValues[row][colIndex] * c);
    }
    result[row] = sum;
  }
  return result;
}


/**
```

```
    * Get a column-wise representation of this matrix
    *
    * @return column-wise matrix
    */
   public SparseColumnMatrix getAsColumnMatrix() {
      SparseColumnMatrix result = new SparseColumnMatrix(this.colCount,
            this.rowCount);
      for (int row = 0; row < this.rowCount; row++) {
         int nzLength = nzCounters[row];
         if (nzLength == 0) {
            continue;
         }
         for (int colIndex = 0; colIndex < nzLength; colIndex++) {
            int column = this.columnIndices[row][colIndex];
            result.set(column, row, this.nzValues[row][colIndex]);
         }
      }
      return result;
   }

   public int getNzCount() {
      int allNz = 0;
      for (int i : this.nzCounters) {
         allNz += i;
      }
      return allNz;
   }
   }
}
```

APPENDIX B:

COMMON STOPWORDS IN ENGLISH

a	able	about	above	abroad
according	accordingly	across	actually	adj
after	afterwards	again	against	ago
ahead	ain't	all	allow	allows
almost	alone	along	alongside	already
also	although	always	am	amid
amidst	among	amongst	an	and
another	any	anybody	anyhow	anyone
anything	anyway	anyways	anywhere	apart
appear	appreciate	appropriate	are	aren't
around	as	aside	ask	asking
associated	at	available	away	awfully
b	back	backward	backwards	be
became	because	become	becomes	becoming
been	before	beforehand	begin	behind
being	believe	below	beside	besides
better	between	beyond	both	brief
but	by	c	came	can
cannot	cant	can't	caption	cause
causes	certain	certainly	changes	clearly
c'mon	co	co.	com	come
comes	concerning	consequently	consider	considering
contain	containing	contains	corresponding	could
couldn't	course	c's	currently	d
dare	daren't	definitely	described	despite
did	didn't	different	directly	do
does	doesn't	doing	done	don't
down	downwards	during	e	each
edu	eg	eight	eighty	either
else	elsewhere	end	ending	enough
entirely	especially	et	etc	even
ever	evermore	every	everybody	everyone
everything	everywhere	ex	exactly	example
except	f	fairly	far	farther
few	fewer	fifth	first	five
followed	following	follows	for	forever
former	formerly	forth	forward	found
four	from	further	furthermore	g

get	gets	getting	given	gives
go	goes	going	gone	got
gotten	greetings	h	had	hadn't
half	happens	hardly	has	hasn't
have	haven't	having	he	he'd
he'll	hello	help	hence	her
here	hereafter	hereby	herein	here's
hereupon	hers	herself	he's	hi
him	himself	his	hither	hopefully
how	howbeit	however	hundred	i
i'd	ie	if	ignored	i'll
i'm	immediate	in	inasmuch	inc
inc.	indeed	indicate	indicated	indicates
inner	inside	insofar	instead	into
is	isn't	it	it'd	it'll
its	it's	itself	i've	j
just	k	keep	keeps	kept
know	known	knows	l	last
lately	later	latter	latterly	least
less	lest	let	let's	like
liked	likely	likewise	little	look
looking	looks	low	lower	ltd
m	made	mainly	make	makes
many	may	maybe	mayn't	me
mean	meantime	meanwhile	merely	might
mightn't	mine	minus	miss	more
moreover	most	mostly	mr	mrs
much	must	mustn't	my	myself
n	name	namely	near	nearly
necessary	need	needn't	needs	neither
never	nevertheless	new	next	nine
ninety	no	nobody	non	none
nonetheless	no-one	nor	normally	not
nothing	notwithstanding	novel	now	nowhere
o	obviously	of	off	often
oh	ok	okay	old	on
once	one	ones	one's	only
onto	opposite	or	other	others
otherwise	ought	oughtn't	our	ours
ourselves	out	outside	over	overall
p	particular	particularly	past	per
perhaps	placed	please	plus	possible

presumably	probably	provided	provides	q
queue	quite	qv	r	rather
rd	re	really	reasonably	recent
recently	regarding	regardless	regards	relatively
respectively	right	round	s	said
same	saw	say	saying	says
second	secondly	see	seeing	seem
seemed	seeming	seems	seen	self
selves	sensible	sent	serious	seriously
seven	several	shall	shan't	she
she'd	she'll	she's	should	shouldn't
since	six	so	some	somebody
someday	somehow	someone	something	sometime
sometimes	somewhat	somewhere	soon	sorry
specified	specify	specifying	still	sub
such	sup	sure	t	take
taken	taking	tell	tends	th
than	thank	thanks	thanx	that
that'll	thats	that's	that've	the
their	theirs	them	themselves	then
thence	there	thereafter	thereby	there'd
therefore	therein	there'll	there're	theres
there's	thereupon	there've	these	they
they'd	they'll	they're	they've	thing
things	think	third	thirty	this
thorough	thoroughly	those	though	three
through	throughout	thru	thus	till
to	together	too	took	toward
towards	tried	tries	truly	try
trying	t's	twice	two	u
un	under	underneath	undoing	unfortunately
unless	unlike	unlikely	until	unto
up	upon	upwards	us	use
used	useful	uses	using	usually
v	value	various	versus	very
via	viz	vs	w	want
wants	was	wasn't	way	we
we'd	welcome	well	we'll	went
were	we're	weren't	we've	what
whatever	what'll	what's	what've	
when	whence	whenever	where	whereafter
whereas	whereby	wherein	where's	whereupon

wherever	whether	which	whichever	while
whilst	whither	who	who'd	whoever
whole	who'll	whom	whomever	who's
whose	why	will	willing	wish
with	within	without	wonder	won't
would	wouldn't	x	y	yes
yet	you	you'd	you'll	your
you're	yours	yourself	yourselves	you've
z	zero			