TMSK: Text-Miner Software Kit

Nitin Indurkhya

Contents

1

Preliminaries

1.1 Overview

The Text-Miner Software Kit (TMSK) is a comprehensive software package for predictive text mining. It includes routines for preprocessing XML-based text documents and provides implementations of all the key tasks described in the book *Text Mining: Predictive Methods for Analyzing Unstructured Information* (see http://www.data-miner.com for details on how to buy this book). The algorithms and other details behind the implementation shall not be discussed in this guide, which focuses on how the user can use this particular implementation. Table 1.1 summarizes the tasks accomplished by TMSK (it does not include rule-based document classification, a task done by the companion RIKTEXT software that is also available from http://www.data-miner.com). For reference, the sections in the book that describe the algorithms are also shown.

These tasks are accomplished by a number of different routines. Some of the routines, for example, the one for End-of-Sentence Detection, are not directly accessible to end-users but instead are sub-modules used by end-user routines. The list of end-user routines available in TMSK are listed in Table 1.2. Let us briefly review them:

mkdict This routine generates a dictionary from a set of documents. Either a global dictionary or a local dictionary (specific for a category) can be generated. The user specifies the size of the desired dictionary. There are many parameters that

Text to Vectors	Tokenization	Section 2.3
	Stemming	Section 2.4
	End-of-Sentence Detection	Section 2.5
	Dictionary Creation	Section 2.11
	Vector Generation	Section 2.11.1
Prediction	Naive Bayes	Section 3.5.4
	Linear Models	Section 3.5.5
Information Retrieval	Document/Query Matcher	Section 4.7
Finding Structure	K-means Clustering	Section 5.2.1
Information Extraction	Named Entity Identification	Section 6.2.2

Table 1.1: Tasks Accomplished by TMSK

Category	Task	TMSK Routine
Text to Vectors	Dictionary Creation	*mkdict*
	Vector Generation	*vectorize*
Prediction	Naive Bayes	*nbayes*, *testnbayes*
	Linear Models	*linear*, *testline*
Information Retrieval	Document/Query Matcher	*matcher*
Finding Structure	K-means Clustering	*kmeans*
Information Extraction	Named Entity Identification	*tagNames*

Table 1.2: End-user Routines in TMSK

the user can adjust to guide the dictionary generation process.

vectorize This routine converts a set of documents into sparse vectors based on a given dictionary. The documents can be labeled or unlabeled. The user can optionally generate an inverted index of the set of documents as well. Vectors can be labeled as belonging to a category or not. Or they can be unlabeled. There are many parameters that the user can adjust to guide the vectorization process.

nbayes This routine builds a naive-bayes binary classifier from a set of labeled vectors.

testnbayes This routine applies a naive-bayes classifier generated by *nbayes* to new vectors and splits the documents corresponding to the vectors into two parts – those classified as positive by the classifier, and those classified as negative. The new vectors can be labeled or unlabeled.

linear This routine builds a binary linear classifier from a set of labeled vectors. There are several parameters that the user can adjust to vary the linear classifier obtained.

testline This routine applies a linear classifier generated by *linear* to new vectors and splits the documents corresponding to the vectors into two parts – those classified as positive by the classifier, and those classified as negative. The new vectors can be labeled or unlabeled.

matcher This routine matches a given document to those in a specified database and returns a user-specified number of documents that are closest to the given document. If the given document consists of just a sequence of keywords, *matcher* behaves like a small search engine.

kmeans This routine performs k-means clustering (the user specifies k, the number of clusters) on a set of documents. An XML tag with the cluster number is added to each document. A description of each cluster is also given in terms of its most important words.

tagNames This routine takes a document and tags all the named entities in it. Each named entity is categorized into one of four types of entities – Person, Organization, Location and Miscellaneous. The output document has color-coded tags (the user can adjust the colors) for convienient viewing in a browser. Optionally, a full list

of the named entities is also generated (for indexing purposes). The tagging is done using a pre-trained classifier.

The various parameters to adjust the behaviour of the routines are specified in a single *tmsk.properties* file. This file also allows the routines to communicate and share information about the structure of the documents under analysis.

1.2 Typographic Conventions

In this guide, for all examples demonstrating computer interaction the fontsize is smaller than the regular text font. The commandline prompt is shown as:

```
%
```

but naturally the actual prompt will differ from system to system (and even user to user). Commands to be typed by the user on a computer terminal immediately follow the prompt and take up the rest of the line. They will be in monospaced type. For example, if the user were to type the command *java mkdict*, this will be shown as:

```
% java mkdict
```

Output of programs is shown in monospaced type immediately after the user input. For example, the following shows the output of the program when the user types *java mkdict* without any arguments. the program displays a description of the correct syntax.

```
% java mkdict
mkdict: usage:
java mkdict size dictionaryfile
java mkdict size category dictionaryfile
```

1.3 Installation

TMSK is available for any hardware that has a java interpreter (version 1.3.1 or higher). Its modules are run as java applications with command-line arguments. Java must be installed on the hardware. To verify that java is installed on your system, is accessible and is the appropriate version, type “java -version” in a shell window (for PC-Windows, an MS-DOS or command-prompt window). If you don’t have java, you first need to obtain and install it (a process that we do not describe here. Refer to your administrator or consult web resources for java installation). TMSK will not run without java.

Assuming that you have java running on your system, the TMSK installation consists of the following steps:

- Create a directory named *tmsk* and download the appropriate install file from the data-miner.com website to this directory:
  - Windows: Working in an MS-DOS or command-prompt window, save as installtmskj.exe; type “cd \tmsk” to connect to the directory; and finally type “installtmskj” to run the installtmskj.exe program which will install TMSK.

  - Unix/Linux: Save as tmskj.tgz; connect to directory *tmsk*; and finally type the command "tar -xvzf tmskj.tgz" which will install TMSK.

- Make sure that the file *tmsk.zip* appears in the *tmsk* directory. **DO NOT UNZIP** *tmsk.zip*.

- Add the full path for tmsk.zip to the environment variable CLASSPATH. Also add the current directory to the CLASSPATH. The process depends on your operating system. For example, in Windows XP, click on the System icon in Control Panel (in the *Performance and Maintenance* category), then on the Advanced tab click *Environment Variables*, then click *New* to add variable CLASSPATH with value .;c:\tmsk\tmsk.zip (if if tmsk.zip resides in the directory c:\tmsk) or, if CLASSPATH already exists, click *Edit* and add .;c:\tmsk\tmsk.zip to the existing value of CLASSPATH. The same process works for Windows NT/2000. For older PC-Windows 9x, add "set classpath=.;c:\tmsk\tmsk.zip" to the autoexec.bat file. For other operating systems, consult their documentation on how to add environment variables.

Notes: The TMSK programs are run from shell windows (For PC-windows, in an MS-DOS or command-prompt window). The environment changes may not take effect until the next login or restart. A drive other than c: may be used. Make sure *both* the current directory and the tmsk.zip directory are in the CLASSPATH.

1.3.1 Verifying the Installation

If the installation was successful, the user should be able to type the command *java mkdict* without arguments in any directory and get the output shown in Section 1.2. If, instead, an error message is obtained, make sure that the tmsk directory has the tmsk.zip file and that the CLASSPATH variable is properly set and has taken effect.

1.3.2 Cleaning Up

Following successful installation, the following files may be deleted or moved to backup storage:

- For Unix/Linux: tmskj.tgz

- For Windows: installtmskj.exe

2

Data Formats

In this chapter we describe the format of the data used (and generated) within TMSK. In general, the user need only be concerned with the format of the input data. However, since all output data generated by TMSK are simple text files, it is useful to know their format as well.

2.1 User-specified Data

The user must provide input data to TMSK for analysis. These consist of: text documents to be analyzed, list of stopwords and stem dictionary. Only the text documents are mandatory input; the list of stopwords and the stem dictionary are optional.

2.1.1 Documents

TMSK analyses text documents in XML format. XML is reviewed in Chapter 2 of the book. The input set of documents must be in the same consistent format. TMSK makes some assumptions about the format of input documents:

- Each document is marked off from other documents by a distinguishing tag. The user specifies this tag in the properties file. For example, <DOC> might the distinguishing tag for a set of documents.
- Only the text in certain sections of the documents is of interest. The user specifies the tags of these sections in the properties file. For example, the user might specify that text marked off by <SUBJECT> and <TEXT> should be considered for analysis.
- Documents may be labeled and the user must specify the tag used to identify labels in the documents. For example, labels might be tagged by <TOPICS>. If there are multiple labels, each individual label might have have a different tag. But the tag of interest to us is the tag that includes *all* the labels.

An example XML document is shown in Figure 2.1 where the document is identified by <DOC>, the relevant text by <TITLE> and <ABSTRACT>, and the labels by <TOPICS>. Note that there are multiple labels, each tagged with <TOPIC>.

```
<DOC>
<TEXT>
<TITLE>
Solving Regression Problems with Rule-based Classifiers
</TITLE>
<AUTHORS>
<AUTHOR>Nitin Indurkhya</AUTHOR>
<AUTHOR>Sholom M. Weiss</AUTHOR>
</AUTHORS>
<ABSTRACT>
We describe a lightweight learning method that induces an ensemble
of decision-rule solutions for regression problems.  Instead of
direct prediction of a continuous output variable, the method
discretizes the variable by k-means clustering and solves the
resultant classification problem. Predictions on new examples are
made by averaging the mean values of classes with votes that are
close in number to the most likely class.  We provide experimental
evidence that this indirect approach can often yield strong results
for many applications, generally outperforming direct approaches
such as regression trees and rivaling bagged regression trees.
</ABSTRACT>
<TOPICS><TOPIC>regression</TOPIC><TOPIC>learning</TOPIC></TOPICS>
</TEXT>
</DOC>
```

Figure 2.1: An XML Document

Since the XML documents are ASCII text and are clearly marked off from each other, they can be simply put in one large file and the file provided as input to TMSK. Alternatively the file could be compressed using zip and the zip file provided to TMSK. If multiple files must be provided to TMSK, the files can all be zipped together and the single toplevel zipfile provided to TMSK. TMSK recognizes zip files as those with the extension *.zip* and treats all other files as ASCII files. Note that the zip files can be arbitrarily nested. In summary, there are two ways of providing a set of XML documents as input to TMSK:

- A single ASCII text file with all the documents. For example, a file *inputdocs.xml* might contain all the XML documents to be analyzed.

- A single zipfile (with the extension *.zip*) which contains one or more files of documents to be analyzed. These files can themselves be either zip files or ASCII files. For example, *threedocs.zip* can be an input file that contains *firstdoc.xml* (an ASCII file with one document) and *twodocs.zip* (a zipfile which in turn contains two documents: *seconddoc.xml* and *thirddoc.xml*).

The documents are assumed to be well-formed XML and TMSK does not validate the XML or check for errors in the nesting of tags. It is the responsibility of the user to validate the input before passing it to TMSK.

```
A
AN
the
for
it
about
for
by
```

Figure 2.2: A Stopword File

```
was be
had have
knelt kneel
knew  know
fungi fungus
```

Figure 2.3: A Stemmer Dictionary

2.1.2 Stopwords

Stopwords are used primarily by the routine *mkdict* in the dictionary creation process and consist of words to be ignored for dictionary creation. Usually they consist of common words that don't have much value for distinguishing between documents. These stopwords are provided in a stopwords file. The words are listed one per line. Case is not relevant and words can be in upper or lower case or a mix of both. For example, *ABOUT*, *about* and *About* all refer to the same word. A sample stopword file is shown in Figure 2.2.

The stopword file can also be used to discard inappropriate words that result from tokenization. For instance, the string +/- keeps showing up as a word and we wish to exclude it, simply adding it to the stopwords file would ensure that it is excluded from the dictionary.

2.1.3 Stem Dictionary

The Stem Dictionary is used as an aid by the stemmer to find stems of words. A sample stem dictionary is shown in Figure 2.3. It consists of two words on each line (separated by an arbitrary number of whitespace characters) – the second word is a stem for the first. Typically, the stemmer dictionary consists of stems that cannot be deduced by the stemmer. Thus, there is no need to include the line *books book* because the stemmer can readily deduce this on its own. Lines that don't contain exactly two words are discarded by the stemmer.

One may elect to add entries to the stemmer to force certain stems. For example, the stemmer may be stemming a certain word incorrectly. Or perhaps we don't wish a word to be stemmed at all. In the latter case, for example, adding the line *organization organization* to the stemmer dictionary would prevent the word *organization* from getting stemmed.

```
profits
company
General:Motors
share
```

Figure 2.4: An Example Dictionary

2.2 TMSK-generated Data

The TMSK routines generate a number of datafiles that are then used as input for other TMSK routines. In this section, we describe the format of these files. Under normal circumstances, there is not much need to know the format of these files – they can be generated and directly provided as input to the appropriate routines. However, the files are text files and advanced users may want to edit some of them manually.

2.2.1 Dictionaries

The TMSK routine *mkdict* generates dictionaries. The format of the dictionary is as follows:

- A series of words, one per line.
- "Regular" words are in lower-case.
- Multi-words consist of "regular" words separated by the ":" character and preserve case. For example, *United:States* is a multi-word.

Figure 2.4 illustrates a sample dictionary with four words, one of which is a multi-word.

2.2.2 Sparse Vectors

TMSK processes documents into a format we call sparse vector form. This format is shared with RIKTEXT (Rule Induction Kit for Text) that is available in conjunction with TMSK. The TMSK routine *vectorize* creates sparse vectors from XML text documents.

The documents are converted into a spreadsheet format where each row corresponds to a document, and each column corresponds to a word from a dictionary. Individual cells in the spreadsheet are filled with frequency counts (number of times the word appears in the document). For more details, the user may refer to Chapters 2 and 3 of the book. Typically, the number of words (columns) is very large and for a given document (row), most of the words do not apply and the corresponding cells are zero. Hence it is more efficient to store only the information of the non-zero cells (the cell number and its value). This is the sparse vector form. Figure 2.5 gives a simple example of the sparse vector form that corresponds to a spreadsheet.

The format of the sparse vector file is as follows:

- All the non-zero pairs for a document appear on the same line.

Spreadsheet				Sparse Vectors
0	15	0	3	(2,15) (4,3)
12	0	0	0	(1,12)
8	0	5	2	(1,8) (3,5) (4,2)

Figure 2.5: Spreadsheet to Sparse Vectors

- The non-zero pairs for each document should be in increasing order of column numbers.
- The non-zero pairs are separated by white space.
- Each non-zero pair is formatted as first the column number, then an character, and finally the frequency.

For example, the vector file corresponding to the spreadsheet of Figure 2.5 would be as follows:

```
2@15 4@3
1@12
1@8 3@5 4@2
```

2.2.2.1 Labeled and Unlabeled Vectors

For training a classifier, we need labeled documents. Having obtained a classifier, we would like to use it to classify unlabeled (new) documents. *vectorize* produces both labeled and unlabeled vectors in the following format:

- If documents are labeled, then the labels appears at the front of the corresponding vectors (separated by whitespace from the non-zero pairs).
- Only binary labels are permitted – documents either belong to a class (label=1) or not (label=0). A categorizer is built for the positive class (cases with label=1).
- In a vector file, all vectors must be of the same type (labeled or unlabeled).

The earlier example of a vector file shown consisted of unlabeled vectors. A hypothetical labeled vector file corresponding to the spreadsheet of Figure 2.5 (with hypothetical labels) might be as follows (the first and last case belong to the class, the middle case does not belong to the class):

```
1 2@15 4@3
0 1@12
1 1@8 3@5 4@2
```

2.2.3 Inverted Index

The sparse vector file shows the nonzero features and frequencies organized in document order. The inverted index contains exactly the same information, but organized in feature order – for each feature, it lists the documents in which the feature is nonzero and also lists the relevant frequency. Since this information is sparse, the inverted index uses several arrays of pointers to store this information to facilitate easy access. We shall describe this index generated using the example of Figure 2.5. For this data, the inverted index file generated by TMSK would be as follows:

```
4       6       3
0       2       3       4
1       2       0       2       0       2
12      8       15      5       3       2
234     144     93
```

The first line contains the sizes of the numbers in the index. For this example, there are 4 features, 6 non-zero frequencies and 3 cases.

We shall get to the second line later. The third and fourth lines contain the frequency and document information organized in feature-order. The third line contains the document numbers (counting from 0), while the fourth line contains the corresponding frequencies. For example, feature 1 has 2 non-zero frequencies: 12 for document 2 (labeled as 0), and 8 for document 3 (labeled as 2). Both these lines contain exactly the same number of elements – the number of non-zero frequencies. The second line contains pointers (counting from 0) to the start of the documents (in lines 3 and 4) for each feature. Thus, feature 1 starts from location 0 (the first element), feature 2 starts from location 2 (the third element), and so on. This line contains as many entries as there are features.

Finally the last line contains a normalization weight for each document. This normalization weight is the sum of the squared frequencies for the document. For example, for the third document, the weight is $8^2 + 5^2 + 2^2 = 93$.

2.2.4 Naive Bayes Classifier Weights

The weights file generated by *nbayes* is used exclusively by *testnbayes*. The general format of the file is as follows:

```
Number of positive class documents
Number of negative class documents
Number of features (m)
For each feature, number of positive class documents with non-zero frequencies.
For each feature, number of negative class documents with non-zero frequencies.
```

For our example, if we assume that the first and third documents are in the positive class, then we will get the following weights file:

```
2
1
4
1
1
1
2
1
0
0
0
```

As can be seen, there is one negative class document with a non-zero frequency for the first feature; the other features have no negative class documents with non-zero frequencies and hence their corresponding entries are 0.

2.2.5 Linear Classifier Weights

The linear classifier weights file is generated by *linear* and is used exclusively by *testline*. It basically contains the weights for the linear classifier (one weight for each feature and a constant. The constant bias is stored at the end). The first line of the weights file has to be one of *tf*, *tf*idf* or *binary* and indicates the feature transformations done to the training data. This helps the *testline* routine determine what feature transformations need to be done to the new documents before applying the linear classifier.

2.2.6 Named-Entity Classifier Weights

The routine *tagNames* uses a weights file called *tmskner.wts* in the tmsk.zip jar archive. It is in a proprietary format and should not be edited or removed (otherwise *tagNames* will no longer operate as expected). As such, its format should not concern the user too much.

3

The Properties File

The TMSK routines share a number of common parameters. Also, there are a number of options that are not likely to be altered much for individual routines. All these parameters and options are convieniently specified in a *tmsk.properties* file. A sample properties file is shown in Figure 3.1. A few characteristics of this file should be obvious:

- Comments can be included and begin with the # character.
- Blank lines are allowed for readability.
- Non-blank lines that are not comments are parameter assignments.
- Parameter assignments are of the type *parameter=value*.
- The parameters can be listed in any order.

The file shown in Figure 3.1 is by no means complete. There are many more parameters that can be specified. Table 3.1 gives a complete list of all the parameters that are specified in the properties file. While it may seem that there are a very large number of parameters to specify, in fact it can be seen that many have default values (and so are optionally specified); others are routine-specific and need to be listed only if the particular routine is being used. The parameters can be logically grouped into sets. In the following sections we shall use such a logical grouping to describe the parameters in greater detail.

3.1 Word Identification: Customize the Tokenizer

There are three optional parameters that control how the tokenizer identifies words in text:

- *word-delimiter*: This parameter specified the characters to be used to delimit words. All the characters are listed together in one string. It defaults to " \n\t\r,.;:!?()<>[]+\"\'" which pretty much covers common usage of words.

```
# this tag identifies individual documents. case sensitive.
doctag=REUTERS

# these are the tags for the text to be used. case sensitive.
bodytags=TITLE BODY

#labeltag is the tag for the categories/labels
labeltag=TOPICS

# input can be a zip file (with extn .zip) or an xml file
infile=reut2-001.sgm

# input dictionary file
dictionary=new.dx

# stopwords to ignore for dictionary creation
stopwords=stop.wds

# stem dictionary used by stemmer. if file name is blank, stemming not done.
stems=stems.list
```

Figure 3.1: A Simple tmsk.properties File

Parameter Name	Brief Description
word-delimiters	A string of characters that can separate words (optional)
whitespace-chars	A string of whitespace characters in text (optional)
sentence-delimiters	A string of the non-whitespace end-of-sentence delimiters (optional)
doctag	XML tag for individual documents
bodytags	XML tags of the text to be analyzed
labeltag	XML tag of document labels (if any)
infile	Name of the input text file of XML documents
dictionary	Name of the dictionary file
stopwords	Name of the stopwords file (optional)
stems	Name of the stemming dictionary file (optional)
vectorfile	Name of the sparse vector file
indexfile	Name of inverted index file corresponding to vectorfile
multi-word-length	Maximum number of words in a multi-word (optional)
multi-word-span	Maximum distance between words in a multi-word (optional)
signif-level	Significance-level for finding multi-words (optional)
minimum-frequency	Minimum frequency of words in the output dictionary (optional)
probability-threshold	Parameter for the *testnbayes* routine (optional)
reject-threshold	Parameter for the *testnbayes* routine (optional)
feature-type	Possible values: binary (0 or 1), tf (term frequency), or tf*idf (weight) (optional)
lambda	Parameter for the *linear* routine (optional)
learning-rate	Parameter for the *linear* routine (optional)
linear-iterations	Parameter for the *linear* routine (optional)
decision-threshold	Parameter for the *linear* routine (optional)
per-color	Parameter for the *tagNames* routine (optional)
loc-color	Parameter for the *tagNames* routine (optional)
org-color	Parameter for the *tagNames* routine (optional)
misc-color	Parameter for the *tagNames* routine (optional)

Table 3.1: Summary of Parameters in tmsk.properties File

- *whitespace-chars*: This parameter specifies the characters to be treated as whitespace. This parameter defaults to " \n\t\r". Again, this is a good default and users to modify it only if they are clear what they are doing.

- *sentence-delimiters*: These are non-whitespace characters that can follow an end-of-sentence period. Ordinarily, these are only white space characters (i.e. the default value of this parameter is null), but there can be text-specific examples where other values need to be considered. For example, in certain marked-up documents a tag can occur immediately after the end-of-sentence period (as in *...are the new world champions.*</*TEXT*>) and for such documents one must specify *sentence-delimiters="<"* in the properties file.

3.2 XML Characteristics of Input Documents

Input text documents are in XML format. The following parameters allow the user to specify the relevant tags:

- *doctag*: This specifies the name of the tag that marks off documents from each other. For example, if <DOC> is the distinguishing tag, then it can be specified as *doctag=DOC*.

- *bodytags*: This specifies the names of the tags that contain the text of interest. Multiple names are separated by whitespace (usually a single space). For example, if one is interested in text within <TITLE> and <BODY>, then this is specified as *bodytags=TITLE BODY*.

- *labeltag*: This specifies where the document labels are located. For example, if the labels are tagged by <TOPICS>, this is specified as *labeltag=TOPICS*. Note that the labeltag to specify must be the one that includes *all* of the labels. For example, individual labels may be tagged with <TOPIC>, but if all the <TOPIC> labels are contained within <TOPICS>, then <TOPICS> is defined as the labeltag, not <TOPIC>.

3.3 Input/Output Files

While most input/output files for the various routines are specified on the commandline itself, some data files that will not vary much during a text mining session are specified as parameters in the properties file. These files are:

- *infile*: This specifies the name of the file that contains the XML documents being mined. This can either be a simple text file with all the XML documents, or it can be a zip file (with the extension *.zip*) that contains all the XML documents (either in flat files or in zip files). It is strictly an input file and is never modified by any TMSK routine.

- *dictionary*: This specifies the name of the dictionary file to be used as input.

- *stopwords*: This specifies the name of the stopwords file. It is strictly an input file and is never modified by the TMSK routines. If this property is missing, or the file is not specified, it is assumed that there is no stopwords file.
- *stems*: This specifies the name of the stemmer dictionary. It is strictly an input file and is never modified. If this property is missing or the file is not specified, it is assumed that there is no stemmer dictionary.
- *vectorfile*: This specifies the name of the file that contains the sparse vectors. Usually the vectors will correspond to the XML documents in the file specified as *infile*. The vectors can be labeled or unlabeled (but must consistently be of the same type).
- *indexfile*: This specifies the name of the inverted index file corresponding to the sparse vectors. It is created by *vectorize* only if the file name is specified. The *matcher* routine requires the indexfile. For other programs, it is often optional (having access to this file speeds up certain routines).

3.4 Multi-Word Features

Multi-word features are generated by both *mkdict* and *vectorize*. These features are controlled by the following parameters:

- *multi-word-length*: This specifies the maximum number of words in a multi-word. Clearly, this must be greater than 1.
- *multi-word-span*: This specifies the maximum distance between the a multi-word's words in the original text. Clearly, this cannot be less than the *multi-word-length*.
- *signif-level*: This specifies the significance-level to use in selecting multi-words for inclusion in the dictionary. The default value is 0.0.

These features must be specified if multi-words are to be generated. Incorporating multi-words can slow down the routines as well as require significantly larger memory resources. Caution is advisable.

3.5 *mkdict*-specific Parameters

- *minimum-frequency*: This specifies the minimum frequency of words for inclusion into the dictionary. The default value is 1.

3.6 *vectorize*-specific Parameters

vectorize has no private parameters. The *indexfile* (if specified) is an output parameter and will be overwritten if it already exist.

Note that *feature-type* is NOT a parameter for *vectorize* which always generates vectors with frequency counts. The conversion to other types of features is done by other routines that use the vectors.

3.7 *nbayes*-specific Parameters

nbayes has no private parameters.

3.8 *testnbayes*-specific Parameters

- *probability-threshold*: This specifies the probability-threshold for making a positive classification. It is an optional parameter (default value is 0.5).
- *reject-threshold*: This specifies the threshold that must be exceeded in order to make a classification. If the threshold is not reached, the document is not classified. It defaults to 0.5 if not specified.

3.9 *linear*-specific Parameters

- *feature-type*: This can be one of *binary*, *tf* or *tf*idf*. The default is *tf*.
- *lambda*: The default value is 0.001, but it can be set to a smaller positive value (for example, 0.0001) if necessary.
- *learning-rate*: The default value is 0.25, but it can be set to another positive real number as well.
- *linear-iterations*: The default value is 40. Set it to a different positive integer if necessary.
- *decision-threshold*: The default value is 0.0. Changing this affects recall and precision. Negative values boost recall (and reduce precision).

3.10 *testline*-specific Parameters

- *decision-threshold*: The default value is 0.0. Changing this affects recall and precision. Negative values boost recall (and reduce precision).

3.11 *matcher*-specific Parameters

- *feature-type*: This can be one of *binary*, *tf* or *tf*idf*. The default is *tf*.

3.12 *kmeans*-specific Parameters

- *feature-type*: This can be one of *binary*, *tf* or *tf*idf*. The default is *tf*.

3.13 *tagNames*-specific Parameters

- *per-color*: The default color for highlighting PER named-entities is *blue*. Change it to a different color here.
- *loc-color*: The default color for highlighting LOC named-entities is *green*. Change it to a different color here.
- *org-color*: The default color for highlighting ORG named-entities is *red*. Change it to a different color here.
- *misc-color*: The default color for highlighting MISC named-entities is *orange*. Change it to a different color here.

4

Running TMSK

In this chapter we will describe each of the TMSK routines in detail and explain how they can be used.

4.1 Creating Dictionaries: *mkdict*

4.1.1 Syntax

The routine *mkdict* creates dictionaries from input XML documents. The syntax is as follows:

```
java mkdict size dictionaryfile
java mkdict size category dictionaryfile
```

If the name of a category is specified, then a *local dictionary* is generated from only the documents of that category. The category is a string and if it contains whitespace, it must be enclosed within quotes to pass it intact to the program. Note that *size* is an integer greater than 1. The dictionary generated is written to *dictionaryfile*.

All other parameters are taken from the *tmsk.properties* file:

- The input file of XML documents is specified with the *infile* parameter.
- Other non-optional parameters are: *doctag* and *bodytags*. If the documents are labeled and a local dictionary is sought, then *labeltag* must also be specified.
- Optional parameters are: *stopwords, stems, minimum-frequency, multi-word-length, multi-word-span, signif-level, word-delimiters, whitespace-chars, sentence-delimiters*.

Note that if stopwords are used to filter the dictionary, these are removed AFTER the words are generated. Hence the size of the dictionary will be less than or equal to *size*.

4.1.2 Examples

Lets say we wish to generate a dictionary from the articles of August 20, 1996 in the Reuters RCV1 distribution. Set parameters in the tmsk.properties file as follows:

```
doctag=newsitem
bodytags=title text
labeltag=metadata
infile=19960820.zip
```

Now, *java mkdict 200 newreut.dx* will generate a global dictionary of 200 words and store it in the file *newreut.dx*. If a local dictionary for the category *ECAT* is sought, then run *java mkdict 200 ECAT ecat.dx* instead which will save the words in *ecat.dx*. The optional parameters can be set to obtain different dictionaries. Note that if stopwords are removed, then in the above examples, the sizes of the dictionaries generated will be less than 200.

4.2 Generating Vectors: *vectorize*

4.2.1 Syntax

The routine *vectorize* creates sparse vectors from input XML documents based on a dictionary. The syntax is as follows:

```
java vectorize vectorfile
java vectorize category vectorfile
```

The vectors are written to the file specified by *vectorfile*. If the name of a category is specified, labeled vectors are produced. Each vector is labeled a "1" if the corresponding document belongs to the category, or a "0" if it does not.

The category is a string and if it contains whitespace, it must be enclosed within quotes to pass it intact to the program. If no category is specified, unlabeled vectors are produced.

All other parameters are taken from the *tmsk.properties* file:

- Input documents are in *infile*, the dictionary is in *dictionary*.
- Other non-optional parameters are: *doctag* and *bodytags*. If the vectors are to be labeled , then *labeltag* must also be specified.
- Optional parameters are: *indexfile, stems, multi-word-length, multi-word-span, word-delimiters, whitespace-chars, sentence-delimiters*.

The optional *indexfile* is worth noting. This corresponds to the inverted index of the vectorfile and is generated by *vectorize* if specified. It is a required input for the *matcher* routine.

4.2.2 Examples

Lets say we have generated a dictionary, *nreut.dx* from the articles of August 20, 1996 in the Reuters RCV1 distribution and wish to vectorize the articles of September 1, 1996 using this dictionary. Set the parameters in tmsk.properties as:

```
doctag=newsitem
bodytags=title text
labeltag=metadata
```

```
infile=19960901.zip
dictionary=nreut.dx
vectorfile=sept1.vec
```

Now, *java vectorize* will generate unlabeled vectors in the file *sept1.vec*. To get labeled vectors for the class *ECAT*, run *java vectorize ECAT* instead. To get an indexfile as well, set the indexfile parameter in the properties file. Other optional parameters should typically be the same as used in the dictionary generation stage.

4.3 Generating Bayes Classifiers: *nbayes*

4.3.1 Syntax

The routine *nbayes* generates a naive-bayes classifier. The syntax is as follows:

```
java nbayes outputfile
```

The *outputfile* will contain the naive-bayes classifier. Input to *nbayes* is provided via the properties file:

- The vectors are obtained from the file specified as the *vectorfile* parameter. Note that the vectors must be labeled.
- Optionally, the *indexfile* is also accessed if available. Having the indexfile speeds up computation slightly.

4.3.2 Examples

Lets say we have generated vectors, *sept1.vec*, and wish to get a naive-bayes classifier based on these vectors. Set the parameter in tmsk.properties as:

```
vectorfile=sept1.vec
```

Now, *java nbayes sept1.nb* will generate a naive-bayes classifier in the file *sept1.nb*.

4.4 Applying Bayes Classifiers: *testnbayes*

4.4.1 Syntax

The routine *testnbayes* applies a naive-bayes classifier to new documents and creates two files – one with positive predictions, the other with negative predictions. The syntax is as follows:

```
java testnbayes wtsFile positiveCasesFile negativeCasesFile
```

The *wtsFile* is the file generated by *nbayes*. All other parameters are taken from the *tmsk.properties* file:

- Input documents are in *infile*, the corresponding vectors are in *vectorfile* and the *doctag* specifies how to separate the documents in *infile*.
- Optional parameters are: *indexfile, probability-threshold, reject-threshold*.

Having *indexfile* speeds up the program slightly. The two thresholds, *probability-threshold* and *reject-threshold* default to 0.5 if not specified. If the vectors are already labeled, performance metrics (precision, recall) are also written to the terminal.

4.4.2 Examples

Lets say we have generated a naive-bayes classifier, *sept1.nb*, and wish to apply it to the documents of August 20, 1996 in the Reuters RCV1 distribution. Set the parameter in tmsk.properties as:

```
doctag=newsitem
infile=19960820.zip
vectorfile=aug20.vec
```

Now, *java testnbayes sept1.nb pcases ncases* will apply the classifier in *sept1.nb* and split the documents in *19960820.zip* into *pcases* (the positive classifications) and *ncases* (the negative classifications) using 0.5 as the probability and reject thresholds.

4.5 Generating Linear Classifiers: *linear*

4.5.1 Syntax

The routine *linear* generates a linear classifier. The syntax is as follows:

```
java linear outputfile
```

The *outputfile* will contain the linear classifier weights. Input to *linear* is provided via the properties file:

- The vectors are obtained from the file specified as the *vectorfile* parameter.Note that the vectors must be labeled.
- Optionally, the *indexfile* is also accessed if available. Having the indexfile speeds up computation slightly.
- Optionally, the following parameters can also be specified: *lambda, learning-rate, decision-threshold, feature-type, linear-iterations*. If they are not specified, default values are used: lambda=0.001, learning-rate=0.25, linear-iterations=40, decision-threshold=0.0 and feature-type=*tf*.

4.5.2 Examples

Lets say we have generated vectors, *sept1.vec*, and wish to get a linear classifier based on these vectors. Set the parameter in tmsk.properties as:

```
vectorfile=sept1.vec
```

Now, *java linear sept1.lc* will generate a linear classifier with default learning parameters in the file *sept1.lc*.

4.6 Applying Linear Classifiers: *testline*

4.6.1 Syntax

The routine *testline* applies a linear classifier to new documents and creates two files – one with positive predictions, the other with negative predictions. The syntax is as follows:

```
java testline wtsFile positiveCasesFile negativeCasesFile
```

The *wtsFile* is the file generated by *linear*. All other parameters are taken from the *tmsk.properties* file:

- Input documents are in *infile*, the corresponding vectors are in *vectorfile* and the *doctag* specifies how to separate the documents in *infile*.
- Optional parameters are: *indexfile, decision-threshold*.

Having *indexfile* speeds up the program slightly. The *decision-threshold* allows precision-recall tradeoff and defaults to 0.0 if not specified (specifying a negative value favors recall over precision). If the vectors are already labeled, performance metrics (precision, recall) are also written to the terminal.

4.6.2 Examples

Lets say we have generated a linear classifier, *sept1.lc*, and wish to apply it to the documents of August 20, 1996 in the Reuters RCV1 distribution. Set the parameter in tmsk.properties as:

```
doctag=newsitem
infile=19960820.zip
vectorfile=aug20.vec
```

Now, *java testnbayes sept1.lc pcases ncases* will apply the classifier in *sept1.lc* and split the documents in *19960820.zip* into *pcases* (the positive classifications) and *ncases* (the negative classifications) using 0.0 as the decision threshold.

4.7 Retrieving Documents: *matcher*

4.7.1 Syntax

The routine *matcher* matches a given document to a set of documents and retrieves the closest matches. The syntax is as follows:

```
java matcher document num-matches outputFile
```

Here *document* is the given document, *num-matches* is a positive integer that specifies how many of the closest matches to retrieve and *outputFile* is a HTML file that contains the matched documents (it can be viewed using a browser). All other parameters are taken from the *tmsk.properties* file:

- The input set of documents from which the matched documents are retrieved is *infile*, inverted index of the corresponding document vectors is in *indexfile*, the corresponding dictionary is in *dictionary* and the *doctag* specifies how to separate the documents in *infile*.

- The optional parameter is *feature-type*. It defaults to *tf* if not specified.

4.7.2 Examples

Lets say we have want to find the 10 closest matches to the document *2286newsML.xml* from the articles of August 20, 1996 in the Reuters RCV1 distribution. Lets say we use a dictionary *glob.dx* to vectorize the articles and get an inverted index *aug20.idx*. Set the parameter in tmsk.properties as:

```
doctag=newsitem
infile=19960820.zip
indexfile=aug20.idx
dictionary=glob.dx
```

Now, *java matcher 2286newsML.xml 10 matches.html* will put the 10 best matches to *2286newsML.xml* into the file *matches.html*.

4.8 Clustering Documents: *kmeans*

4.8.1 Syntax

The routine *kmeans* clusters a set of documents into a given number of clusters using the k-means algorithm. The syntax is as follows:

```
java kmeans k outputFile
```

Here k is a positive integer, the number of clusters. The *outputFile* will contain the input documents with each document having a new tag, <*tmsk:kmeans*>, that specifies the cluster number (from 1 to k) to which it has been clustered. All other parameters are taken from the *tmsk.properties* file:

- The input set of documents to be clustered is *infile*, the corresponding vectors are in *vectorfile*, the corresponding dictionary is in *dictionary* and the *doctag* specifies how to separate the documents in *infile*.

- The optional parameters are *indexfile, feature-type*. Having *indexfile* speeds up the program slightly. *feature-type* defaults to *tf* if not specified.

Cluster details (key descriptive words) are written to the terminal and can be redirected to a file and saved for future reference.

The clustering is done on the vectors. If labeled vectors are provided, then the labels are simply ignored for the clustering process. The number of documents should match the number of vectors, but if there are more documents than vectors, as many documents as there are vectors are tagged with the corresponding cluster numbers and the remaining documents are simply left as given.

4.8.2 Examples

Lets say we have want to cluster the articles of August 20, 1996 in the Reuters RCV1 distribution into 10 clusters. Lets say we use a dictionary *glob.dx* to vectorize the articles and get vectors *aug20.vec*. Set the parameter in tmsk.properties as:

```
doctag=newsitem
infile=19960820.zip
indexfile=aug20.idx
dictionary=glob.dx
```

Now, *java kmeans 10 clustered.xml* will cluster the articles in *19960820.zip* into 10 clusters and write the retagged articles to the output file *clustered.xml*.

4.9 Extracting Named-Entities: *tagNames*

4.9.1 Syntax

The routine *tagNames* takes a document and identifies named-entities in it. Four types of named-entities are recognized: PERson, ORGanization, LOCation and MISCellaneous. The syntax is as follows:

```
java tagNames inFile outFile
java tagNames inFile outFile namesFile
```

If *namesFile* is specified, all the named-entities (along with their category: PER, ORG, LOC or MISC) are saved in this file for future use. *outFile* is the same in *inFile* except it has all the named-entities tagged with HTML color-codes that allow visual inspection of the named-entities in a browser. The optional parameters are taken from the *tmsk.properties* file:

- *word-delimiters, whitespace-chars, sentence-delimiters, per-color, org-color, loc-color, misc-color*. Default values are used for these parameters.

4.9.2 Examples

Lets say we want to find the named-entities in the document *2286newsML.xml* of the Reuters RCV1 distribution. Lets say we want to save them to a file *newreut.ne* as well. Lets say we are happy with the default color-codes for the named-entities. *java tagNames 2286newsML.xml ner.html newreut.ne* will put named-entities in *newreut.ne* and also produce *ner.html*, a copy of *2286newsML.xml* with color-coded named-entities, that can be viewed using a browser.

5

An Extended Example

In this chapter we present an extended example of TMSK in action. The purpose is to show how this software can be used for text mining.

5.1 The Data

The data we use is the OHSUMED collection of abstracts gathered from MEDLINE. At least at the moment, the data can be downloaded from: ftp://medir.ohsu.edu/pub/ohsumed and probably other places (search the web).

The corpus that we use is actually an arbitrary selection from the total OHSUMED corpus. Since the OHSUMED collection is not in XML fromat, a special processing program was necessary to transform the data.

A java program to convert the OHSUMED file to XML is included in Appendix A. Obviously it is not good for other data, but still, it illustrates how to go about the conversion.

Once the data is in XML format, it is ready to be processed by TMSK. The relevant XML attributes in the tmsk.properties file are:

```
doctag=DOC
bodytags=TITLE TEXT
labeltag=SUBJECTS
sentence-delimiters="\"<"
```

We set ourselves the task of finding a good linear classifier for the category *Rabbits* (we choose the linear classifier as it is generally more accurate than Naive Bayes). We set up training and test sets. There were 11,561 training cases and 3158 test cases, approximately a 4:1 split. The uneven numbers arise because some of the originally selected test documents had no text content and were discarded.

5.2 Initial Experiments

To start, we will generate a global dictionary with 1000 features, without deleting stop words and without stemming.

```
% java mkdict 1000 global.dt
```

We will use this dictionary *global.dt* to generate train and test vectors, *train.vec* and *test.vec*. Note that *infile* will be different for the two vectorize calls shown below:

```
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
```

We will not show the changes to the tmsk.properties file unless there is something notable about the change.

Having obtained train/test vectors, we can try the linear program with default parameters:

```
% java linear rabbit.wts
precision: 74.6377      recall: 95.3704         f-measure: 83.7398
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 38.2353      recall: 55.3191         f-measure: 45.2174
```

The two files *rabpos.txt* and *rapneg.txt* contain the positive and negative predictions of the linear classifier in *rabbit.wts*.

The performance of this classifier on the test set is certainly not impressive. As a first step, let us generate a new dictionary by deleting stop words. This results in a dictionary of 884 words. Using this dictionary, we re-generate train/test vectors and retrain the linear classifier:

```
% java mkdict 1000 global.dt
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
% java linear rabbit.wts
precision: 94.8357      recall: 93.5185         f-measure: 94.1725
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 41.3793      recall: 51.0638         f-measure: 45.7143
```

The result is a bit better than before, but the improvement is slight. Let us next try with stemming, a procedure which tends to increase the frequency of features by neutralizing grammatical variants. The resulting dictionary has 886 stems.

```
% java mkdict 1000 global.dt
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
% java linear rabbit.wts
precision: 97.7064      recall: 98.6111         f-measure: 98.1567
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 67.3469      recall: 70.2128         f-measure: 68.7500
```

This is a real improvement over our previous results. And is a good initial benchmark.

5.3 Further tuning

Another dictionary option to try is generating multi-word features. We will leave the stop word and stemming options as they are, but specify multi-word phrase of length two with a possible separation of 4 words. The multi-word option requires significantly larger memory resources.

```
% java -Xmx500m mkdict 1000 global.dt
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
% java linear rabbit.wts
```

```
precision: 97.6959      recall: 98.1481         f-measure: 97.9215
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 66.0000      recall: 70.2128         f-measure: 68.04
```

We see a very slight degradation in our resulting classifier. It appears the extra cost of phrase processing is not worthwhile.

Returning to the use of stop words and stemming but no phrases, we can try both reducing and increasing the number of features. Reducing the feature set to a nominal 500 features gives the following result:

```
% java mkdict 500 global.dt
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
% java linear rabbit.wts
precision: 64.4444      recall: 26.8519         f-measure: 37.9085
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 21.7391      recall: 10.6383         f-measure: 14.2857
```

Reducing the feature set is apparently a very bad idea for this data. We can also try doubling the feature set size:

```
% java mkdict 2000 global.dt
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
% java linear rabbit.wts
precision: 99.5392      recall: 100.0000        f-measure: 99.7691
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 58.1818      recall: 68.0851         f-measure: 62.7451
```

Increasing the feature set size is not as disastrous as decreasing it was, but the results on the test set are somewhat weaker, primarily because of reduced precision.

We can also vary the method of weighting features. Resetting to stemming, stop words and 1000 features, we change the weights to binary from the default of total frequency.

```
% java mkdict 1000 global.dt
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
% java linear rabbit.wts
precision: 94.1704      recall: 97.2222         f-measure: 95.6720
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 74.4681      recall: 74.4681         f-measure: 74.4681
```

This is the best result yet on the test set. We should try the other weighting option, tf*idf:

```
% java mkdict 1000 global.dt
% java vectorize Rabbits train.vec
% java vectorize Rabbits test.vec
% java linear rabbit.wts
precision: 97.7273      recall: 99.5370         f-measure: 98.6239
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 55.7692      recall: 61.7021         f-measure: 58.5859
```

This result is worse than simple total frequency and substantially worse than binary features.

Another possible improvement is to change the parameters in the linear classifier. The one possibly affecting classification performance is the size of the search space, lambda. We can try setting it to 0.0001 and rerun the linear classifier:

```
% java linear rabbit.wts
precision: 100.0000     recall: 100.0000        f-measure: 100.0000
% java testline rabbit.wts rabpos.txt rabneg.txt
precision: 57.4074      recall: 65.9574         f-measure: 61.3861
```

This is also not a terribly good idea.

5.4 Conclusion

From the experiments above, it appears that the best classifier for the topic *Rabbits* results from stop word elimination, stemming and 1000 binary features. Of course, the result might be different for a different topic or for topics in a different data set. While this is by no means the best solution for this data, it illustrates the process involved in iteratively trying different parameter values and examining the results.

Appendix A

```
/* ohsumed2XML                                */
/** formats the OHSUMED files into XML         */
/*  java ohsumed2XML inputfile outputfile      */
/*                                             */
/* This program is specific to the OHSUMED     */
/* files but is an example of how to convert   */
/* other non-xml files                         */
/*                                             */
/* The OHSUMED files have an indicator tag     */
/* the data on the following line(s):          */
/*
/* .I 274314                                                      */
/* .U                                                             */
/* 91002386                                                       */
/* .S                                                             */
/* Br J Dermatol 9101; 123(3):365-73                              */
/* .M                                                             */
/* Adult; Female; Human; Male; Nails/AH/US; Ultrasonics; Water.   */
/* .T                                                             */
/* Ultrasound velocity in human fingernail and effects of hydration:  */
/* validation of in vivo nail thickness measurement techniques.   */
/* .P                                                             */
/* JOURNAL ARTICLE.                                               */
/* .W                                                             */
/* Distal nail thickness was measured using an electronic micrometer  */
/* and both distal and proximal nail ultrasound times were recorded   */
/*                       ......                                   */
/* .A                                                             */
/* Finlay AY; Western B; Edwards C.                               */
/*                                                                */
/*                                          */
/* The XML version:                         */
/*                                          */
/* <DOC>                                                          */
/* <TITLE>                                                        */
/* Ultrasound velocity in human fingernail and effects of hydration:  */
/* validation of in vivo nail thickness measurement techniques.   */
/* </TITLE>                                                       */
/* <AUTHOR>                                                       */
/* Finlay AY; Western B; Edwards C.                               */
/* </AUTHOR>                                                      */
/* <SUBJECTS>                                                     */
/* <SUBJECT>                                                      */
/* Adult                                                          */
```

```
/* </SUBJECT>                                                          */
/* <SUBJECT>                                                          */
/* Female                                                             */
/* </SUBJECT>                                                         */
/* <SUBJECT>                                                          */
/* Human                                                              */
/* </SUBJECT>                                                         */
/* <SUBJECT>                                                          */
/* Male                                                               */
/* </SUBJECT>                                                         */
/* <SUBJECT>                                                          */
/* Nails                                                              */
/* </SUBJECT>                                                         */
/* <SUBJECT>                                                          */
/* Ultrasonics                                                        */
/* </SUBJECT>                                                         */
/* <SUBJECT>                                                          */
/* Water                                                              */
/* </SUBJECT>                                                         */
/* </SUBJECTS>                                                        */
/* <TEXT>                                                             */
/* Distal nail thickness was measured using an electronic             */
/* micrometer and both distal and proximal nail ultrasound            */
/* times were recorded in 20 volunteers (10 male, 10 female),         */
/*          ...                                                       */
/* </TEXT>                                                            */
/* </DOC>                                                             */

import java.text.*;
import java.util.*;
import java.io.*;

class ohsu2XML {
public static void main(String[] args) {

   String line;
    int outlen = 72; // set output line length
    int linect;
    int posi;
    int marker;
    int ixent;
    boolean htmlflg;
    boolean textflg;
    boolean longflg = false;
   String str;

   String title;
   String body;
   String byline;
   String subject;
   String subjec;
   String source;
   StringBuffer text = new StringBuffer();

    try {
     if (args.length<2 || args.length>2)
  throw new tmskException("usage:\njava ohsumed2XML infile outfile");
```

```
     }
catch (tmskException e1) {System.out.println("mkdict: "+e1.getMessage());}
      BufferedReader in = null;
      PrintWriter out = null;
      PrintWriter pw = null;
    try {
      FileReader inpf = new FileReader(args[0]);
      // Create buffered reader
       in = new BufferedReader(inpf);
       out = new PrintWriter(System.out);

   } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
    try {
 pw = new PrintWriter(new FileWriter(args[1]));
 pw.println("<?xml version=\"1.0\" encoding=\"ISO-8859-1\" standalone=\"yes\"?>");
  pw.println("<CORPUS>");
  System.out.println("ready to process");
  str = "x"; // set for initial test below
  while (str != null){
       linect = 0;
        htmlflg = false;
      body = "";
      byline = "";
      subject = "";
      source = "";
      title = "";
      text.setLength(0);
       textflg = false;
      try {
      str = in.readLine(); // read first .I
      } catch (IOException e) {
 e.printStackTrace();
      }
      try {
  while ((str = in.readLine()) != null) {
        if (str.startsWith(".I")){
           break; //a new document starts here
         }
        if (str.startsWith(".U")) {
            continue;    // skip
           }
         else if (str.startsWith(".S")){
             source = in.readLine();  // source of doc
             continue;
             }
         else if (str.startsWith(".A")){
             byline = in.readLine();  // authors
             continue;
             }
         else if (str.startsWith(".T")){
             title = in.readLine(); // title
             continue;
             }
         else if (str.startsWith(".M")){
           // need to strip final . if there
```

```
            subject = in.readLine();  // the topics of the doc
           if (subject.endsWith(".")) {
             subject = subject.substring(0,subject.length() - 1);
             }
          subject = subject + ";";
            continue;
           }
        else if (str.startsWith(".W")){ // text
             while (!body.endsWith(".")) {
            body = body + in.readLine();
             }
            continue;
            }
        else  { // a skippable line
            continue;
          }
      }
     } catch (IOException e) {
e.printStackTrace();
     }
       if (body.length() == 0) continue;
  // now put saved data into XML file
        text.append(body);  // set stringbuffer
       int bodlen = body.length();
      // write out sgml
      pw.println("<DOC>");
      pw.println("<TITLE>");
      pw.println(title);
      pw.println("</TITLE>");
      pw.println("<AUTHOR>");
      pw.println(byline);
      pw.println("</AUTHOR>");
      pw.println("<SUBJECTS>");
 // iterate over the subjects, separated by semicolon

    StringTokenizer tksu = new StringTokenizer(subject,";");
     while (tksu.hasMoreTokens()){
       subjec = tksu.nextToken();
     // Increase frequency of subject by dropping suffix
      if (subjec.indexOf("/") > 0) {
        subjec = subjec.substring(0,subjec.indexOf("/"));
       }
        subjec = subjec.trim();
        subjec = subjec.replace(' ','_');
      pw.println("<SUBJECT>");
      pw.println(subjec);
      pw.println("</SUBJECT>");
     }
      pw.println("</SUBJECTS>");
      pw.println("<TEXT>");
  // instead of one long line, print text in short lines
     int nech = 0;
     int sch = 0; // starting point
     int ech = 60; // arbitrary ending point
   while(ech < bodlen) {
       nech = ech;
      for (int itx = ech; itx > sch; itx--){
```

```
        if (!(text.charAt(itx) == ' ')) {
           nech = itx;
        }
       else break;
       }
       ech = nech - 1;
       line = text.substring(sch,ech);
       pw.println(line);
       sch = nech;
       ech = sch + 60;
     }
       pw.println("</TEXT>");
       pw.println("</DOC>");
   }
   } catch (IOException e) {
      e.printStackTrace();
    }


    try {
       pw.println("</CORPUS>");
      out.close();
      in.close();
      pw.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
 }
}
```