A SOFTWARE INFRASTRUCTURE FOR RESEARCH IN TEXTUAL DATA MINING

LARS E. HOLZMAN
Computer Science and Engineering, Lehigh University
Bethlehem, PA 18015
leh7@lehigh.edu

TODD A. FISHER
Computer Science and Engineering, Lehigh University
Bethlehem, PA 18015

LEON M. GALITSKY
Computer Science and Engineering, Lehigh University
Bethlehem, PA 18015

APRIL KONTOSTATHIS
Computer Science and Engineering, Lehigh University
Bethlehem, PA 18015

WILLIAM M. POTTENGER
Computer Science and Engineering, Lehigh University
Bethlehem, PA 18015

Received (29 February 2004)

Few tools exist that address the challenges facing researchers in the Textual Data Mining (TDM) field. Some are too specific to their application, or are prototypes not suitable for general use. More general tools often are not capable of processing large volumes of data. We have created a Textual Data Mining Infrastructure (TMI) that incorporates both existing and new capabilities in a reusable framework conducive to developing new tools and components. TMI adheres to strict guidelines that allow it to run in a wide range of processing environments – as a result, it accommodates the volume of computing and diversity of research occurring in TDM. A unique capability of TMI is support for optimization. This facilitates text mining research by automating the search for optimal parameters in text mining algorithms. In this article we describe a number of applications that use the TMI. A brief tutorial is provided on the use of TMI. We present several novel results that have not been published elsewhere. We also discuss how the TMI utilizes existing machine-learning libraries, thereby enabling researchers to continue and extend their endeavors with minimal effort. Towards that end, TMI is available on the web at hddi.cse.lehigh.edu.

1. Introduction

The Textual Data Mining (TDM) field was born in part from the necessity to mine large amounts of text automatically. The field involves both supervised and unsupervised approaches to learning that are employed in applications such as the automatic detection of trends in textual data[1]. Such learning tasks are costly both in terms of implementation

and computation due to a number of factors, including the sparse nature of some textual data representations.

As a result, researchers often spend substantial time creating tools necessary to access, clean and model textual data. Many tools exist but are often specific to the application for which they were designed or are not publicly available. In response we have designed and implemented a Text Mining Infrastructure (TMI), which we have placed in the public domain. Our TMI supports familiar tools and extends their capabilities.

In section 2 we discuss our framework for Textual Data Mining using a simplified example. Some of the basic terminology adopted in the TMI is outlined in section 3. In section 4 we show the practical advantages of the TMI in a discussion of several novel TDM applications that are built upon the TMI framework. TMI can be used for traditional textual processing applications, such as search and retrieval, as well for applications that merge text processing and machine learning, such as Emerging Trend Detection. Section 5 discusses the advantages of TMI in comparison to other TDM systems. In section 6 we provide a brief tutorial on the use of TMI by examining a simple example. We offer conclusions and outline future work in sections 7 and 8.

2. Background and Example Application

The TMI extends and enhances the Hierarchical Distributed Dynamic Indexing[2] system developed under the direction of William M. Pottenger, Ph.D. In order to facilitate our research in TDM, we first undertook a study of several text mining applications. Based on this study, we identified three basic components that appear regularly in TDM applications: a repository of unprocessed text data (also known as a corpus), a relational mapping between documents and their features, and one or more machine-learning methods.

Figure 1 depicts these three components for the TMI implementation of a research system for the detection of emerging trends in textual data. The figure outlines the emerging trend detection process and demonstrates how the various TMI components interact.

In the first step shown in Figure 1, items (e.g., documents) in repositories are parsed and words are tagged with their part of speech prior to the extraction of significant features. In this application, clusters of textual features are identified using an unsupervised learning technique that results in the generation of a Semantic Model[3]. Statistical attributes from the Semantic Model, along with a truth set composed of both emerging and non-emerging trends, are used to produce a training set for a machine-learning algorithm.

In the example depicted in Figure 1, the machine learning algorithms in the WEKA[4] library are used to identify emerging trends. The system also provides interfaces to MLC++[5] and other libraries. A researcher can easily design and integrate algorithms into TMI. The performance of a given application is evaluated using standard metrics such as precision and recall.

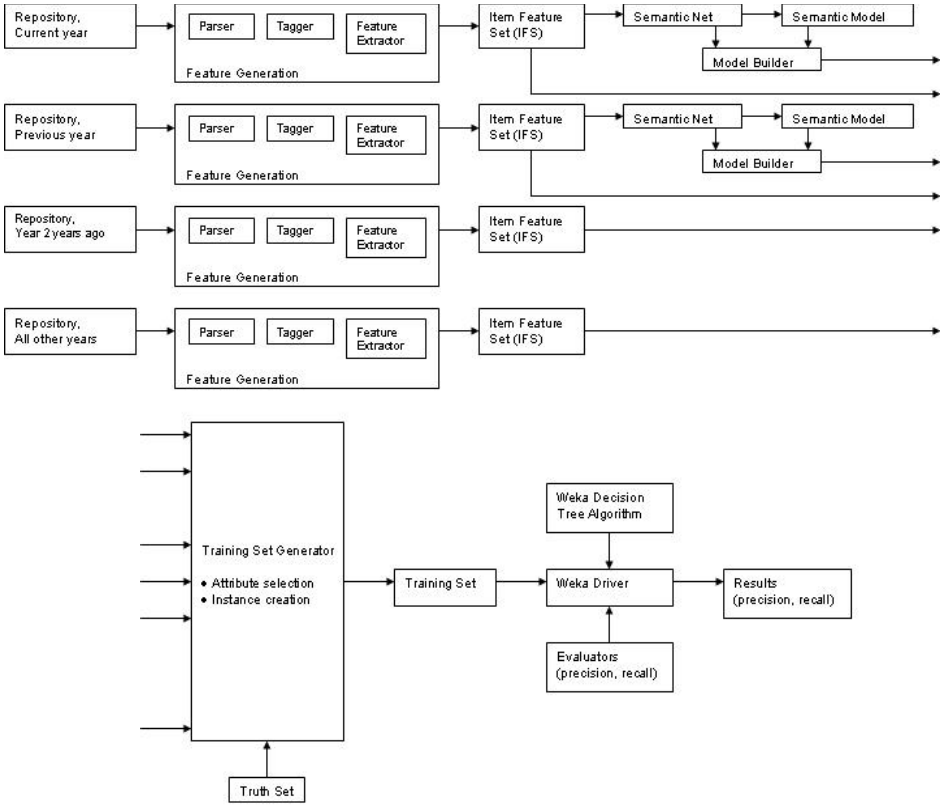

Fig. 1. The emerging trend detection driver using TMI

The entire process depicted in Figure 1 is fully automated in TMI and is described in more detail in Section 4.1. A unique capability of TMI not shown in Figure 1 is support for optimization. Applications implemented using TMI can be automatically optimized across a variety of parameters and optimization targets. Support for optimization facilitates text mining research by automating the search for optimal parameters in, for example, machine learning algorithms.

3. An Overview of TMI

In this section we give an overview of key aspects of TMI. We focus on the concepts that are most relevant to users of the system. A more detailed tutorial for the use of TMI is available on the web at hddi.cse.lehigh.edu.

3.1. *Support for multiple platforms*

TMI supports many variants of the Unix operating system as well as Windows 98, NT, 2000 and XP. The GCC 3.2 and Visual Studio .Net 2003 C++ compilers are supported in these environments, respectively. Through the JNI, TMI also supports Java SDK 1.3 and 1.4. Many of these capabilities are employed in the example application discussed in Section 4.1 on emerging trend detection.

3.2. *Use of a common object interface*

A set of abstract interfaces was created to represent the framework for TDM described above. These interfaces include representations for Repositories, Items, Features, and the other concepts that will be discussed in the applications presented in Section 4[a].

3.3. *Use of a component framework*

TMI uses a component framework to allow rapid and intuitive design of experiments. Components form the building blocks for the experiments, with each component representing some processing operation. Thus, each component is responsible for either producing an object or producing a scalar output. To perform its operation a component may use any number of other objects or scalar inputs. The user can dynamically query a component for information about its inputs and outputs.

3.4. *Use of configurations*

To enable these generic components to perform specific experiments they must be configured. A *Configuration* object is provided for this purpose. This object holds a number of scalars that can be used by the component. Components may have optional or required inputs which obtain their values from the configuration. The values in the

[a] For a complete listing of the interfaces that are provided and descriptions of their use see the online documentation at http://hddi.cse.lehigh.edu.

configuration need not be singular values and ranges and sets of values can be specified for looping and optimization applications.

3.5. *Evaluation*

TMI implements interfaces in support of a variety of evaluation methodologies and metrics. For example, TMI supports the evaluation of search and retrieval applications using gold standard collections. Such gold standard collections can be employed to automatically assess performance of an information retrieval algorithm based on the metrics of precision and recall. TMI also supports the widely used method of n-fold cross-validation for evaluation.

3.6. *Use of component paths*

To further ease experiment design TMI introduces the concept of a component path. A component path describes the serial execution of a set of components. A component path is itself a component, and this is useful because it allows more complex driver design through abstraction. For example, a component path may be designed for ten-fold cross-validation and then reused in other component paths.

It is possible to loop over a particular component in the execution path. This allows a researcher to automatically iterate over a set of values for a parameter. The ability to loop over a set of object dependencies is also provided. We have found it useful to loop over a number of different feature generators, for instance, to perform the same experiment on words, noun phrases, and collocations.

The component path also provides an optimization framework. In this framework the researcher chooses a set of parameters and an evaluation metric for a particular component. An optimization method is then selected and used to identify the values of the provided parameters that maximize (or minimize) the given metric.

4. Applications

The development of TMI was guided by several applications in different domains of TDM. In this section we review several of these applications. We describe the emerging trend detection system in depth and briefly discuss the others. These applications have been implemented using TMI and we hope by presenting them to shed light on the utility of the TMI system.

We begin with some notation: A *driver* is a component that assembles and coordinates other components of TMI in a specific task. A driver instantiates the components in a particular experiment, describes their dependencies, places them in a component path, and then executes the component path.

Repositories are divided into *items*. An item is a textual object such as a document. Each item is composed of *features*. For example, a noun phrase (feature) may occur in a single article (item), in a collection of documents (repository).

Many applications require a relational representation of the textual data. We term one form of this mapping an Item Feature Set (IFS). The IFS maintains the relationship between items and features (such as which features occur in which items). This data structure serves as an interface from the repository to one or more machine-learning methods.

4.1. *Emerging Trend Detection*

A trend can be classified as emerging if it is growing in interest and utility over time. XML is an example of such a trend that emerged in the mid-90s. Emerging Trend Detection (ETD) in TDM is an active research area, and to the best of our knowledge our efforts represent the only attempts to fully automate the process of trend detection. To this end we have developed a new approach for ETD using the TMI. The methods developed are extensions of those reported in[6]. An overview of previous research and commercial systems that can be used to track trends can be found in[1].

At a high level, the operation of the TMI ETD driver is depicted in Figure 1. Its purpose is to apply both unsupervised and supervised learning techniques in the construction of a model capable of predicting the emergence of trends. The goal is to employ the model in a classification task such as technology forecasting.

The data for the experiments reported in this article was a selection of INSPEC®[7] abstracts drawn from the fields of Data Mining and Object Oriented Programming. The data was represented in an XML format. An XML Repository Builder component was used to create repositories from the INSPEC® source. This supported selection of relevant years. We used the Abstract field as our item text.

4.1.1. *Feature Generation.*

Maximal noun-phrases were used for features in these experiments. TMI was used to parse and tag the text in the XML repositories with part-of-speech markups per Ref. 8. The feature extractor was based on a lexical analyzer created with GNU's Flex (Ref. 9) [b].

4.1.2. *Construction of Item Feature Sets*

Four item feature sets (IFSs) were built: one for the target year (e.g., 2003), one for the year prior to the target year, one for the year two years prior to the target year, and one

[b] An early prototype of the feature generator was presented in Bader et al. (Ref. 13).

for remaining prior years. Three tables were created in each IFS to efficiently store occurrence statistics. The first contained all the items in the repository and the second contained all the features in those items. The third contained a set of item-feature relations between the first two tables. Item-feature relations record item-feature-specific information such as the occurrence of a specific feature in a specific item. Unique IDs were also assigned to items and features so they could be represented uniformly throughout the system and located efficiently.

4.1.3. *Unsupervised Model Construction*

The first machine-learning process performed unsupervised clustering of features in two stages: semantic network generation and semantic network partitioning (clustering).

Semantic Network Generation

A semantic network is a graph of nodes connected by weighted arcs. In this case, each node represented an extracted feature and arc weights between nodes were computed using an asymmetric similarity measure[10]. Since each IFS essentially consisted of an inverse item index, similarity measures based on co-occurrence frequencies were straightforward to calculate.

Semantic Network Partitioning

One approach to network partitioning involved an unsupervised learning process that discovered significant regions of semantic locality (sLoc) within each of the semantic networks [3].

A second attempt employed results from Ref. 11 in which a new framework for understanding Latent Semantic Index[12] was established. This framework is based upon the concept of term co-occurrences and, using this framework, an alternate term clustering approach was applied.

Classes were derived from two TMI interfaces to accomplish this: *Semantic Model*, which represented the partitioning process, and *Cluster*, which represented the regions of semantic locality.

4.1.4. *Supervised Model Construction*

The second machine-learning process performed was supervised – in this case, decision tree induction.

Attribute Interface

The TMI *Attribute* interface supports attributes in one of four forms: nominal, Boolean, integer, or real. An example attribute in this application is the occurrence

frequency of a feature (e.g., "XML") two years prior to the target year. As noted in section 4.1.2, this frequency information is computed during formation of item feature sets. Attribution occurs in this case by passing the IFS to an attribute object. An attribute must be implemented such that a check is performed that ensures that the type and range of the value is valid.

Truth Set Interface

Truth Set is an interface that supports input of labeled data. For example, in this application the truth set consisted of feature/classification pairs. Features (e.g., noun phrases) were classified as emerging or non-emerging by a domain expert.

TMI incorporates a standard for truth sets. Each set consists of triples comprised of a textual representation of the object being classified, a classification for the object, and a relevance measure.

It will be helpful at this point to clarify the difference between a truth set triple and an instance. Traditionally an instance (or exemplar) is a single example of the concept to be learned in a machine-learning application. A triple in a truth set is related to, but not the same, as an instance. In this application, a single triple was a feature/classification pair such as {"XML", emerging, 1.0}. Each triple was used to generate actual instances comprised of time-sensitive statistical attribute values associated with the feature.

The TMI provides a parser that generates a truth set object from an input file created by a domain expert. This input file format (TFF) and the concept of a truth set are specific to the TMI.

Training Set Interface

A training set is represented in the TMI as a set of instances, where instances consist of a set of attributes with an optional class. Any attribute can be nondestructively ignored or activated at the training set level to support attribute subset selection. An attribute subset selection interface exists, but was not used in this driver. The training set interface allows access to a training set in one of three ways: through a WEKA or MLC++ file format, or directly by iterating through the instances in memory. Each instance in the training set in this application consisted of seven attributes:

- the occurrence of the feature in the target year
- the occurrence of the feature in the year preceding the target year
- the occurrence of the feature two years before the target year
- the occurrence of the feature in all years before the target year
- the number of features in the cluster containing the feature in the target year
- the number of features in the cluster containing the feature in the year preceding the target year
- the number of words in the feature longer than length four

The first four attributes were computed from the item feature sets discussed in section 4.1.2, the second two from the models discussed in section 4.1.3, and the last one from the feature itself as an estimate of the degree of semantics it bears.

Training Set Generation Interface

To ease the use of the various machine-learning methods available in TMI, a training set generator abstract class was designed to support creation of a training set from a truth set. As noted, in this application, the truth set contained features and their domain expert classifications (emerging or non-emerging). The generator first employed the parser to read in the domain expert truth triples from a file. Given the item feature sets and the models discussed previously, the generator then created a training set consisting of the seven attributes listed above, along with a nominal value for the classification.

Machine-Learning Algorithm Interface

Since drivers may use multiple machine-learning algorithms within the same framework, a separate machine-learning algorithm abstract class was designed in TMI. This interface supports access to both predefined and programmer implemented training and classification methods, whether supervised or unsupervised in nature.

In this application an instantiation of a machine-learning algorithm class used JNI to launch the JVM and load a training set using WEKA utilities. The driver then called the J48 decision tree classifier in the WEKA library to perform training and testing using the training set [4].

4.1.5. *Evaluation in ETD*

An *Evaluator* interface was designed in TMI to handle the many forms of evaluation required for validation of various techniques and algorithms in TDM. A machine-learning driver, for example, can employ evaluators to test a model that results from training.

In this application, precision, recall and F_β were employed using true and false positives (TP and FP) and false negatives (FN) where *precision = TP/(TP+FP)* and *recall = TP/(TP+FN)*, and F_β averages precision and recall when $\beta=1$. The machine-learning driver evaluated the decision tree using ten-fold cross validation implemented in TMI.

4.1.6. *Experiments in ETD.*

The TMI driver for ETD was compiled and executed using Visual Studio .Net 2003 in a Windows XP environment on a 2.0 GHz Pentium 4 with 512 MB of main memory. Five repositories were used for experimentation. The first four repositories were formed from the INSPEC® database using the search term “data mining”. The repositories were

formed by varying the current year for the experiments between 1996, 1997, 1998, and 1999 (referred to as INSPEC® 96, 97, 98, and 99 respectively). All years after the target year were discarded. The fifth repository was generated from INSPEC® using the search term "object oriented software engineering" (OOSE repository).

An example decision tree is presented in Figure 2. This decision tree was generated using the LSI clustering method and the INSPEC® 98 repository. Note that the time sensitive Concepts_in_Cluster (number of features in the cluster of a feature in the target year) is used in this decision tree.

Both clustering methods were successful in automatically detecting emerging trends. A summary of the results can be found in Table 1. In the table the P columns report precision, the R columns report recall, and the F_β columns report F_β (β=1). As detailed in Ref. 14, the results are of high quality overall with F_β greater than or equal to eighty percent in all cases.

```
Occurrences_in_All_Noncurrent_Years <= 2
|  Long_Words_In_Feature <= 1
|  |  Concepts_in_Cluster <= 14: notrend
|  |  Concepts_in_Cluster > 14: trend
|  Long_Words_In_Feature > 1: trend
Occurrences_in_All_Noncurrent_Years > 2
|  Occurrences_in_Current_Year <= 21
|  |  Occurrences_in_All_Noncurrent_Years <= 3
|  |  |  Occurrences_in_Year_Before_Previous_Year <= 1: notrend
|  |  |  Occurrences_in_Year_Before_Previous_Year > 1: trend
|  |  Occurrences_in_All_Noncurrent_Years > 3: notrend
|  Occurrences_in_Current_Year > 21
|  |  Long_Words_In_Feature <= 2: trend
|  |  Long_Words_In_Feature > 2: notrend
```

Fig. 2. Decision tree for the detection of emerging trends

Table 1. Automatic ETD results

	P sLoc	P LSI	R sLoc	R LSI	F_β sLoc	F_β LSI
INSPEC 96	0.90	0.87	0.79	0.89	0.84	0.88
INSPEC 97	0.83	0.82	0.77	0.93	0.80	0.87
INSPEC 98	NA[c]	0.71	NA[c]	0.93	NA[c]	0.81
INSPEC 99	0.82	0.79	0.84	0.92	0.83	0.85
OOSE	0.81	0.86	0.90	0.93	0.85	0.89

[c] Results are not reported for the INSPEC® 98 repository using sLoc because these results were not available at the time of writing.

4.1.7. *Summary of ETD Application*

The ETD driver was written completely in C++ and used the JNI interface to access the WEKA library in multiple components. The driver compiled successfully under GCC 2.96, GCC 3.1, GCC 3.2 and Microsoft Visual Studio 6.0 and .Net 2003. It is composed of a set of relatively small components that can be reused in other applications. The LSI clustering was added to the existing ETD framework in less than a day of programming. Parameters that could be varied automatically in the process of optimization include the semantic network threshold, the semantic network pruning parameter, and the attributes used in the training set. This example also illustrates the use of TMI to test competing algorithms, in this case sLoc and our LSI-based clustering technique.

4.2. *Massively Parallel Feature Extraction*

One of the primary tasks in mining distributed textual data is feature extraction. The widespread digitization of information has created a wealth of data that requires novel approaches to feature extraction in a distributed environment. We have designed and implemented a massively parallel model for feature extraction in a highly distributed environment using TMI.

In previous work, we have shown that speedups linear in the number of processors are achievable for applications involving reduction operations such as feature extraction[15]. We are also in the process of validating an analytical model for estimating communication and execution time complexity with empirical observations based on the extraction of features from a large number of pages on the World Wide Web. In this paper we present for the first time our approach using the TMI architecture in this application.

In this application, processors act in concert to download items, extract features, and build an item feature set in a parallel-pipelined fashion. There are three TMI components in this application. The first component handles the generation of features per the algorithms discussed in section 4.1.1. The second performs a merge operation in the construction of the item feature set. The third performs a multithreaded download of data from various web servers such as the US Government Patent and Trademark Office server. All three components are integrated into a Single Program Multiple Data (SPMD) C++ Message Passing Interface (MPI) framework.

4.2.1. *Multithreaded Download.*

In this application we implemented a multithreaded crawler that allows the results from many web sites to be retrieved in parallel. In this way we effectively use the web as a massive repository. By overlapping network access with on-the-fly conversion from

HTML to XML, we achieved the throughput required to sustain feature generation and item feature set construction in the parallel pipeline.

4.2.2. *Feature Generation*

Per the design goals of TMI, as noted the ETD feature generation component discussed in section 4.1.1 was reused in this application. The feature generation component was enhanced with the development of an XML parser to handle multiple types of data. Thus, the generator used a variety of feature extraction techniques in order to process the different types of data involved (e.g., author lists vs. titles, etc.).

4.2.3. *Construction of Item Feature Sets*

The second component implemented using TMI was the construction of item feature sets. Figure 3depicts the merge operation that implements item feature set construction in the context of the parallel-pipeline model of execution.

The leaf nodes of different shades in Figure 3 represent the execution of the feature extraction task and the interior nodes of each reduction tree represent the merging operation. This figure depicts execution on eight processors. The arrow edges represent the communication that takes place between processors. Dotted lines combined with arrow edges together form reduction trees that compute item feature sets.

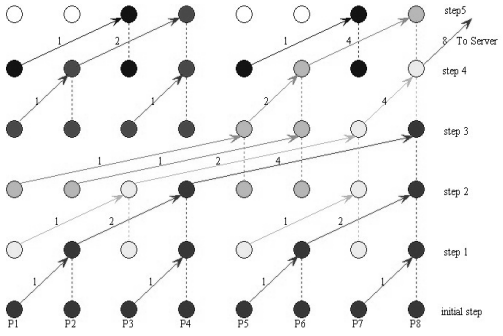

Fig. 3. Item Feature Set construction in Parallel-Pipeline model of execution

4.2.4. *Summary of Massively Parallel Feature Extraction*

The massively parallel feature extraction TMI components were written in C++. They compiled successfully under GCC 2.7, GCC 3.1, and Microsoft Visual Studio 6.0. The components were targeted at and executed on the Los Lobos computational grid at the

National Computational Science Alliance High Performance Computing Center at the University of New Mexico in Albuquerque, NM. Los Lobos consists of 256 IBM NetFinity 4500R dual processor servers running the Red Hat Linux operating system and linked via a Myrinet interconnect.

Preliminary results from execution on Los Lobos confirm the near-linear speedups predicted by the model reported in [15]. Further tests have been run on the National Center for Supercomputing Applications IA-32 grid, producing both consistent and improved results from those obtained on Los Lobos.

This second application highlights the reuse of the ETD feature generator with a distributed variant of the item feature set construction phase implemented in the merge operation. We were able to achieve this readily using TMI despite the very different execution environments of these two applications.

4.3. *Classification of Emotions in Chat*

In this application of TMI, we present an approach to discovering the emotion present in Internet chat messages. The approach uses tools to reconstruct the speech from chat messages and tallies the number of distinct phonemes in each message. The method uses a vector of the phoneme counts as well as other statistically derived attributes in an N-Nearest-Neighbor Instance-Based Learning model.

TMI was used as a utility to assist in the process of training set development for this application. This presents a relevant example of how TMI can be used to assist with text processing to allow a more traditional data mining approach to be applied to textual data. This application's organization is similar to the ETD driver in this respect. Further details can be found in Ref. 16.

4.3.1. *Results of Emotion Classification*

This application performed well showing results exceeding 90% F_β (β=1) in many cases. Table 2 portrays partial results obtained. These results were obtained by varying the number of nearest neighbors used for classification between 10 and 42. Three different attribute subsets were used as indicated by Full, Phonemes, and Subset. The full attribute set used all the variables including the statistical ones; the phonemes set used only the phonemes; and the final subset used a number of automatic attribute subset selection techniques to identify an optimal subset. The results reported utilized ten-fold cross validation on a set of 1201 messages.

Table 2. K-Nearest-Neighbor instance based learning to identify emotion in chat messages

		Neutral			*Happy*			
		Prec.	**Recall**	**F-beta**	**Prec.**	**Recall**	**F-beta**	**k**
Full	**Avg**	0.841	0.985	0.9074	0.829	0.594	0.6911	
	Max	0.840	0.993	0.9101	0.902	0.597	0.7185	37
Phonemes	**Avg**	0.840	0.978	0.9035	0.778	0.606	0.6814	
	Max	0.840	0.979	0.9042	0.792	0.613	0.6911	23
Subset	**Avg**	0.839	0.987	0.9072	0.859	0.595	0.7030	
	Max	0.841	0.993	0.9101	0.902	0.597	0.7185	37

4.3.2. *Summary of Emotion Detection in Chat Messages*

This application was written in C++ and successfully compiled using Microsoft Visual Studio 6.0. It was not suitable for use in a Linux environment due to the reliance on a Microsoft library for speech production. The components were executed on a Pentium IV desktop PC. In this research we have shown that phonemes are related to the emotion expressed in a chat message. We have also shown that a machine learning model can use this information to detect emotion with reasonable accuracy. This will provide a strong foundation for further higher-level analysis of chat data.

TMI proved useful in designing this application because it enabled easy conversion of raw tagged textual data into a highly structured format. This is a good example of the utility of TMI for handling the many complexities that arise in the fields of text mining and statistical natural language processing.

5. Related Work

In this section we discuss research in the development of tools for automating machine learning and text processing tasks. In some cases, the TMI provides an extension of these tools. We will also contrast other systems with the TMI.

5.1. *WEKA*

WEKA is a machine-learning library developed at the University of Waikato [4]. It provides Java implementations of several methods for machine learning, data preprocessing, and evaluation. WEKA uses the Attribute-Relation File Format (ARFF) for input of training and testing data.

WEKA is, however, primarily a generic machine learning library, and as such lacks specific support for certain functionality critical to text mining research. WEKA, in fact, does not provide any functionality outside of machine learning and some very basic experiment design capabilities. Notably absent for our purposes are pre/post-processing capabilities, and training set generation. Finally, WEKA is written in Java, which limits its performance in various ways.

WEKA through its experimenter interface also provides some limited optimization capability. In particular, it is possible to combine a number of training sets with various machine learning methods and a combination of options (which set parameter values). The experimenter interface then generates results for each possible combination. This is a naïve method of search that does not employ optimization. The experimenter also lacks the ability to optimize entire applications such as our ETD example above; rather it only isolates the machine learning step.

Nonetheless, TMI supports WEKA at several junctures in order to provide a flexible environment for rapid prototyping of text mining algorithms and applications. For instance, a TMI training set object can save itself as a WEKA ARFF file, and can be seamlessly loaded into WEKA. Or, a WEKA method can be wrapped in a machine-learning algorithm as with the ETD driver and used in a TMI optimization loop.

5.2. *MLC++*

MLC++ [5] is another well-known machine-learning library. The most recent version is available from SGI for use by data mining researchers. It is comparable to WEKA but does not have the same scope in that it only supports supervised learning. MLC++ is however written in C++ and presents opportunities for use in data-intensive applications that WEKA may handle less readily. As a result, TMI also supports seamless access to MLC++.

5.3. *GATE*

The General Architecture for Textual Engineering (GATE) [17] is similar to TMI in some respects. Using the Java beans component model, it however focuses on feature generation. The highly developed GATE library contains many advanced methods for extracting and tagging textual features.

GATE offers more flexibility than the current feature generation algorithms implemented in TMI. GATE does not, however, provide a development environment in which entire applications can be evaluated in an optimization loop. This is a critical point that needs to be emphasized, because it is for this reason that we undertook the development of TMI. In essence, we have found that it is not practical to evaluate a text-mining algorithm outside the context of an actual application. For this reason, we felt that the time was ripe to develop a framework that facilitated both algorithm development and evaluation in the context of applications.

5.4. *D2K*

Data to Knowledge[18] is a rapid prototyping data flow environment for data mining applications written in Java. Like TMI, D2K provides modules for assembly into a data mining system. It can also filter data and visualize results.

TMI offers several advantages over D2K. First, TMI is targeted specifically at text mining, and as discussed earlier, thus has intrinsic support for textual feature extraction and textual clustering. Second, as with WEKA, D2K is a Java-based environment and as such suffers from similar limitations. Finally, TMI has been designed from the outset to support optimization, something which none of these competing systems have as an explicit design goal.

5.5. *YALE*

Yet Another Learning Environment (YALE) [19] is another Java based machine learning library. An important capability that is similar to (albeit simpler than) our concept of a "component" is the YALE concept of an "operator". An operator (similar to a C++ operator) performs a basic operation. Operators can be embedded in the same way that component paths are in TMI. The data dependencies of such paths are however not explicitly established as in TMI. All execution is performed sequentially with a few exceptions. Specifically, YALE supports feature subset selection and parameter search. Similar to WEKA, the parameter search can only be performed using a naïve method of search. Experiments are described either through a GUI or XML configuration.

YALE is lacking a number of properties necessary for the complexity of TDM. Primarily, there is the need in TDM applications to describe more complex data and execution paths. Due to the deficiencies described above this is not possible with YALE. Secondly, naïve search for optimal parameters becomes of little or no use with complex applications such as ETD that often occur in TDM. Finally, YALE essentially supports only the manipulation of training sets, and as such is clearly oriented at machine learning and not TDM in general.

6. A Brief Tutorial

In this section we will provide a brief tutorial for the use of TMI. A more in depth tutorial is provided at: http://hddi.cse.lehigh.edu/docs/tutorial.pdf. The brief tutorial provided here only gives a brief sketch of how to create an experiment in TMI. Anyone who wishes to do serious work in TMI should read the full tutorial. This brief tutorial will examine the construction of a sample TDM system and then the conversion of this system to TMI code.

6.1. *The System Description*

The example system that will be described here is implemented in the ClusterTest driver that is supplied with the default install of TMI. The ClusterTest driver performs the following operation: load a set of files, extract words from the files, create a graph from these words using a co-occurrence metric for arc weight, and finally partition this graph to create semantic clusters of the words. This process has a number of applications including emerging trend detection and document clustering.

The method that will be used to design the example system has three main steps. First, the components to be used are selected from the component library. Next, diagrams are constructed to describe the relationship and configuration of these components. Finally, these diagrams are converted to C++ code that utilizes TMI. Converting the diagrams to code is straightforward as there are ways to directly describe the relationships that occur in the diagrams through code. We plan to automate this process using a Graphical User Interface that we are currently designing.

6.2. *The Components*

The first step to creating a TMI experiment is selecting the proper components to perform the desired experiment. To select these components one must decide how to obtain the data, what operations to perform on the data, and in most cases how to evaluate the results of these operations.

A number of components will be used to build the example system. A brief description is provided here of each of these components.

FileRepBuilder: Builds a Repository from a list of files
CCTagger: Brill's part of speech Tagger [20].
WordExtractor: Extracts word Features from text
GenFg: Coordinates Feature generation/extraction
GenIfs: Coordinates statistics about Items and Features
CoNetImp: Builds a SemanticNetwork which is represented as an asymmetric graph with features as nodes and a co-occurrence metric used for arc weight between the nodes.
TarModel: Creates a semantic model by partitioning the SemanticNetwork described above using the sLoc algorithm [3].
ModelBuilder: Coordinates the unsupervised feature clustering process

6.3. *Data and Execution Flow Design*

After the proper components are selected they must be organized in two ways. First, a model of the data flow must be constructed. All TDM applications on some level involve

progressive processing from a data source to some eventual result. The data flow model for the example we are examining can be seen in Figure 4. This model uses three types of relations represented by light lines, dark lines with one dot, and dark lines with two dots. The light lines represent inputs that must be obtained from a configuration object. The single dot lines represent a dependency in which the object with the dot is using the object without a dot. The double dot lines represent that the object to the right is using as an input an output from the object on the left. Thus, FileRepBuilder takes an input LOCATION which is the location of the file to build a repository out of and has an output REPOSITORY, the produced repository, which is used by GenIfs.

Next, an execution flow diagram is designed. This diagram contains the same items that are present in the data flow model. An order of execution is designed for these components. In the simplest case this will be a sequential execution which will be represented by lines with arrows indicating the order of execution. This is the case for our sample application and the diagram can be seen in Figure 5. Not all objects must be included as the execution of some objects is implicit (e.g., CoNetImp and TarModel in the diagram).

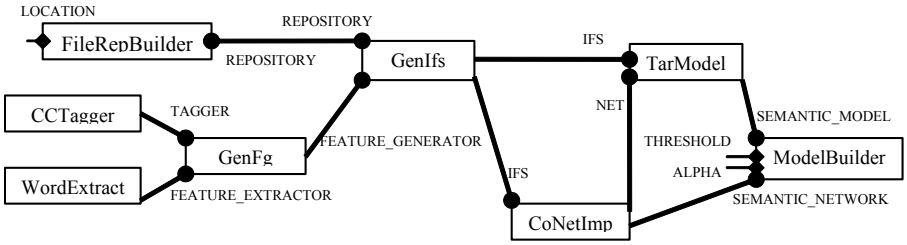

Fig. 4. Data Flow Diagram

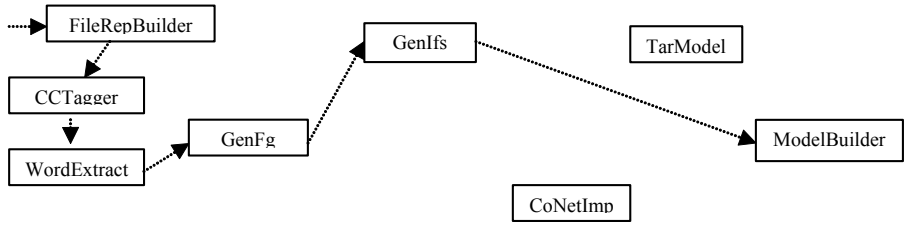

Fig. 5. Execution Flow Diagram

6.4. *Coding the Data Flow Diagram*

Components can be declared in the code in any order as long as all components exist before they are used. Configurations are also declared here to account for the necessary

inputs. The first Component in Figure 4 is the FileRepBuilder. It is instantiated with the following code:

```
TdmPtr<RepositoryBuilder>
    rm(new FileRepBuilder("…"));
```

Notice that this is simply the instantiation of the FileRepBuilder object which is represented in Figure 4. The configuration can be created as follows:

```
TdmPtr<Configuration> cRm = new
   Configuration("…");
cRm->setParameter("LOCATION",input);
```

This sets the parameter LOCATION to the value held in the value input. The rest of the components are created similarly. With these components created it is possible to begin describing the relationships between the components. An example is that the GenFg (instantiated as fg) uses the CCTagger (instantiated as cc). This is described as follows:

```
fg->usesAs("TAGGER",cc.generic());
```

In this example TAGGER is the name of the input on fg that cc is being used for. Another type of relationship that is described is the relationship where a product of a component is used as an input, for example in the case of the RepositoryBuilder (instantiated as rm) and GenIfs (instantiated as ifs). This relationship is described as follows:

```
ifs->usesProductAs
      ("REPOSITORY",*rm,"REPOSITORY");
```

This code declares that the output REPOSITORY of rm is used to satisfy the input REPOSITORY of ifs. The complete code for this example can be found in the full tutorial available online and is included with the default install of TMI.

6.5. *Coding the Execution Flow Diagram*

Since the execution of this example is sequential it is straightforward to code the execution flow. The first step is to define a component path as follows:

```
TdmPtr<TdmComponentPath> path =
   new TdmComponentPath(…,true);
```

The first value is the name of the component path and the second value specifies whether to use debug or quiet execution. Once the component path is created the components must be added in order. For this example the code would look like this:

```
path->addSequential(rm->component(),cRm);
path->addSequential(cc->component());
path->addSequential(fe->component());
path->addSequential(fg->component());
path->addSequential(ifs->component());
path->addSequential(mb->component(),mbConf);
```

Notice that the components which are connected in the execution diagram occur in this code in the order that they are to be executed. Also, the components which require inputs have the appropriate Configurations specified. Finally, the process method is called to begin the execution path specified utilizing the data flow that has been designed. This will begin execution of the experiment that has been designed.

7. Future Work

It is our hope that TMI will prove useful to the point that it becomes the basis for a standard framework for textual data mining that co-exists with and leverages other frameworks such as WEKA and MLC++. In this way we hope to provide a suitable platform for advancing TDM research. We anticipate that sharing TMI openly will promote the advancement of the field.

One of the main remaining tasks that lie ahead is the formal incorporation of customized optimization algorithms in TMI. Currently only a small class of the available optimization algorithms are suitable for our purpose. We have experience in the field of optimization [21,22] and plan to release a version of TMI that includes a variety of 'off-the-shelf' optimization algorithms. The current version provides only one such algorithm, a gradient based bound constrained quasi-Newton method.

Finally, we are developing a parallel component path. This will take advantage of the modular properties of our component architecture to allow accurate description of experiments. In particular, native parallelism will be described in the parallel component path definition. This parallelism will then be exploited when the driver is executed. We expect to support both shared memory and message passing environments in this way in the next release of TMI.

8. Conclusion

We have detailed a novel infrastructure and library that meet a real need on the part of textual data mining (TDM) researchers. Our framework supports multiple platforms, large data sets, existing tools and reusable components. We have offered a number of novel techniques in this infrastructure including the ability to design arbitrarily complex systems and perform advanced optimization. Meanwhile, our system remains conducive to rapid prototyping and research. We have discussed several applications that are already using TMI successfully and illustrated these advances. Our goal is to continue to refine TMI into a standard framework that can be widely used for TDM research and development. To that end we have officially released TMI, available online at hddi.cse.lehigh.edu.

9. Acknowledgments

This research was supported in part by NSF CISE EIA grant number 0087977. The authors wish to thank family and friends for their love and support, as well as the faculty and support staff in the Computer Science and Engineering Department at Lehigh University. The authors would also like to thank the following co-workers for their contributions to this work: Jirada Kuntraruk, Christopher J. Crowe, Daniel G. and Sarah E. Darr, Eric D. Miller, Faisal M. Khan, Mark R. Aevermann and Eduardo J. Freyre. Team members Dan and Sarah Darr, Eric Miller and co-author William M. Pottenger wish to express their sincere gratitude to their Lord and Savior, Yeshua (Jesus) the Messiah (Christ) for their salvation.

References

1. Kontostathis, April, Leon Galitsky, William M. Pottenger, Soma Roy and Daniel J. Phelps. A Survey of Emerging Trend Detection in Textual Data Mining. In *A Comprehensive Survey of Text Mining,* Michael Berry, Ed., Springer-Verlag. 2003. Available on WWW: www.cse.lehigh.edu/~billp/pubs/ETDArticle.ps.gz

2. Hierarchical Distributed Dynamic Indexing. Available on WWW: hddi.cse.lehigh.edu

3. Bouskila, F. D. and William M. Pottenger. The Role of Semantic Locality in Hierarchical Distributed Dynamic Indexing. In the *Proceedings of the International Conference on Artificial Intelligence (IC-AI'2000)*, Las Vegas, NV, June 2000. Available on WWW: www.cse.lehigh.edu/~billp/HDDI/sloc.ps.gz

4. Witten, I.H. and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, San Francisco, CA. 2000. Available on WWW: www.cs.waikato.ac.nz/ml/weka

5. Machine Learning in C++. Available on WWW: www.sgi.com/tech/mlc/

6. Pottenger, William M. and Ting-hao Yang. Detecting Emerging Concepts in Textual Data Mining. In *Computational Information Retrieval*, Michael Berry, Ed., SIAM, Philadelphia, PA, August 100`. Available on WWW: www.cse.lehigh.edu/~billp/pubs/SIAMETD.ps.gz

7. INSPEC® Bibliographic Information Service. Available on WWW: www.iee.org/Publish/INSPEC/

8. Brill, E. A Simple Rule-based Part of Speech Tagger, In the *Proceedings of the Third Conference on Applied Natural Language Processing*, Trento, Italy, March/April 1992.

9. Fast Lexical Analyzer Generator. Available on WWW: www.gnu.org/software/flex/

10. Pottenger, William M., Yong-Bin Kim and Daryl D. Meling. HDDI™: Hierarchical Distributed Dynamic Indexing. In *Data Mining for Scientific and Engineering Applications*, Robert Grossman, Chandrika Kamath, Vipin Kumar and Raju Namburu, Eds., Kluwer Academic Publishers, July 2001. Available on WWW: www.cse.lehigh.edu/~billp/pubs/HDDIFinalChapter.pdf

11. Kontostathis, April and William M. Pottenger. A framework for understanding LSI performance. *Proceedings of ACM SIGIR Workshop on Mathematical/Formal Methods in Information Retrieval* (ACMSIGIR MF/IR '03). 2003.

12. Deerwester, Scott, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6): 391-407. 1990.

13. Bader, R., M. Callahan, D. Grim, J. Krause, N. Miller and William M. Pottenger. The Role of the HDDI™ Collection Builder in Hierarchical Distributed Dynamic Indexing. *Proceedings of the Textmine '01 Workshop, First SIAM International Conference on Data Mining*. April 2001. Available on WWW: www.cse.lehigh.edu/~billp/pubs/HDDICB.doc

14. Kontostathis, April. A Term Co-occurrence Based Framework for Understanding LSI: theory and practice. Ph.D. Dissertation, Department of Computer Science and Engineering at Lehigh University, August, 2003.

15. Kuntraruk, Jirada, and William M. Pottenger. Massively Parallel Distributed Feature Extraction in Textual Data Mining Using HDDI™. In the *Proceedings of The Tenth IEEE International Symposium on High Performance Distributed Computing* (HPDC-10). San Francisco, CA, August 2001. Available on WWW: www.cse.lehigh.edu/~billp/pubs/IEEEArticle.ps.gz

16. Holzman, Lars E. and William M. Pottenger. Classification of Emotions in Internet Chat: An Application of Machine Learning Using Speech Phonemes. 2003. Available on WWW: www.lehigh.edu/~leh7/papers/EmotionClassification.pdf

17. General Architecture for Text Engineering. Available on WWW: www.dcs.shef.ac.uk/research/groups/nlp/gate/

18. Data To Knowledge. Available on WWW: www.ncsa.uiuc.edu/TechFocus/Projects/NCSA/D2K_-_Data_To_Knowledge.html

19. Yet Another Learning Environment. Available on WWW: yale.cs.uni-dortmund.de/index.eng.html

20. Brill, E. Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part of Speech Tagging. Computational Linguistics. 1995.

21. Pinto, Vivek D., William M. Pottenger, and William "Tilt" Thompkins. A Survey of Optimization Techniques Being Used in the Field. In the *Proceedings of the Third International Meeting on Research in Logistics (IMRL 2000).* Quebec, Canada, May. Available on WWW: www.cse.lehigh.edu/~billp/pubs/SurveyOfOptimization.doc

22. Pinto, Vivek D., William M. Pottenger, and William "Tilt" Thompkins. A Multi-Level Multi-Objective Optimization Of a Stochastic Enterprise Resource Planning Model Using Simulated Annealing. To appear in the *European Journal of Operational Research*, Roman Slowinski, Ed. 2002