TMI: Text Mining Infrastructure and Library
A tutorial

Lars E. Holzman, Timothy F. Cunningham and William M. Pottenger
tmi@cse.lehigh.edu

Parallel and Distributed Text Mining Lab
Department of Computer Science and Engineering
Lehigh University

http://hddi.cse.lehigh.edu
For support email: tmi@cse.lehigh.edu
Contents

1. Preface
   1.1. Acknowledgements
   1.2. Quick Introduction

2. Installation
   2.1. Obtaining a copy
   2.2. Other libraries
       2.2.1. WEKA
       2.2.2. JNI
       2.2.3. Flex
       2.2.4. MPI
   2.3. Visual Studio Settings
       2.3.1. .NET 2003
   2.4. Visual Studio Wizards
   2.5. Linux

3. TMI
   3.1. TdmObj
   3.2. TdmPtr
   3.3. Primitive
   3.4. Descriptors
   3.5. Component
   3.6. Configuration
   3.7. TdmComponentPath
       3.7.1. Sequential Execution
       3.7.2. Configuring Input & Output Pins
       3.7.3. Looping
           3.7.3.1. Looping Over a Parameter
           3.7.3.2. Looping Over Uses Dependencies
       3.7.4. Optimization
4. An Example System
   4.1. System Description
   4.2. Relevant Components
   4.3. Assembling the Components
   4.4. Writing the Code
      4.4.1. Before
      4.4.2. Data Flow Description
      4.4.3. Execution Flow Description
      4.4.4. After

5. Creating Your Own Driver
   5.1. Adding a Driver/Test
   5.2. Using the Wizard for Visual Studio
   5.3. Updating Makefiles
   5.4. Writing the Code

6. Creating Your Own Component(s)
   6.1. Choose Your Interface
   6.2. Adding a Component
   6.3. Adding to Your Visual Studio Workspace
   6.4. Updating Makefiles
   6.5. Writing the Code
      6.5.1. Using TdmComponent

A. Appendices
   A. Component List
   B. Interface List
   C. Example Driver Table
1. Preface

1.1. Acknowledgements

This library in its current form constitutes a number of years of work from a number of motivated people. In this section we would like to acknowledge and thank as many of these people as possible. First and foremost, we would like to acknowledge all the members of the Parallel and Distributed Text Mining Lab at Lehigh University. The research in this lab has both driven work on TMI and supported it thoroughly. We would like to acknowledge the following people who were or are involved in the TMI project: Faisal M. Khan, Murat Ganiz, Christopher R. Spencer, Philip C. Garcia, Todd A. Fisher, Leon M. Galtisky, April Kontostathis, Jirada Kuntraruk, Christopher J. Crowe, Daniel G. and Sarah E. Darr, Eric D. Miller, Mark R. Aevermann, and Eduardo J. Freyre. We would also like to thank the Computer Science and Engineering Department and the PC Rossin College of Engineering and Applied Science for their support. This research was supported in part by funding from the National Science Foundation (grant numbers 0087977 and 0231768). Team members Chris Spencer, Dan and Sarah Darr, Eric Miller, and William M. Pottenger wish to express their sincere gratitude to their Lord and Savior, Yeshua (Jesus) the Messiah (Christ) for their salvation.

1.2. A Brief Introduction

Few tools exist that address the challenges facing researchers in the field of Textual Data Mining (TDM). 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. They are not extensible and not robust enough to continue to support cutting edge research in TDM.

TMI is a library that we have developed to address this need. TMI comes complete with solutions to obtain data, process this data, and then apply machine learning processes to the processed data. Furthermore, it provides the ability to do this in a highly customizable and extensible way. This library to our knowledge is the most complete research library capable of handling Textual Data Mining from data acquisition through standardized evaluation.

TMI is a research library. This means that our library experiences rapid growth and constant development and refining. One may notice a lack of polish that can be found on productized systems. This is to be expected but do not let it deter you. TMI is a very powerful and carefully designed system. A number of features are offered to allow easy access to the capabilities for experiment design and rapid prototyping.
TMI is written in C++ and is supported by the most recent Microsoft C++ and GNU C++ compilers. Specifically, we support Visual Studio .Net 2003 and g++ 3.31.

This tutorial is designed to walk you through the initial steps of installing TMI and also provides information and examples on designing an experiment and adding components to the library.

1.3. Other Documentation

While using this tutorial you may find it helpful to reference supplementary documentation available online. Specifically, we have doxygen automatic documentation for the most recent stable version available at http://hddi.cse.lehigh.edu/dox/. In addition, component documentation is available at http://hddi.cse.lehigh.edu/CompDoc/toc.html.
2. Installation

2.1. Obtaining a copy

To obtain a copy of TMI it is necessary to go to our webpage at http://hddi.cse.lehigh.edu/. Instructions are provided at the site for the download of our library. We only provide the library in source form because it is intended for developers and not end-users. The single source provided will work on all supported platforms. (Note that Visual Studio 6 is no longer supported.)

If you would like to obtain the development version of the library from our CVS repository please send email to tmi@cse.lehigh.edu with a brief explanation of why. This system MAY BE buggy, so be forewarned.

2.2. Other libraries

TMI relies on a number of other libraries to provide additional functionality. As the scope of our library is oriented specifically at the challenges that arise from textual data mining research we have done our best to employ the best of libraries from other disciplines. These libraries must first be installed before our system can be used in full.

2.2.1. WEKA

WEKA is a Machine Learning library that contains a number of useful tools. It includes versions of many commonly used machine learning algorithms including a Java version of Quinlan’s C4.5 decision tree induction algorithm. WEKA includes other useful tools for attribute subset selection, clustering, and visualization as well.

Download WEKA

WEKA is available on the web at: http://www.cs.waikato.ac.nz/ml/weka/. You can download any version but TMI has been tested with the most recent version.

Installing WEKA

To install WEKA, decompress the file you receive to a directory of your choice. In that directory the file weka.jar will be supplied. You must copy this file into (TMI)/res/WEKA for best results. The notation (TMI) refers to the absolute pathname of the directory (or folder) in which the TMI is rooted. This allows a stable relative path for
TMI to access the WEKA library. It is possible to set up your global class path configuration as well, but instructions are not supplied for this here.

2.2.2. JNI (Java Native Interface)

JNI is used to access Java tools such as WEKA directly from the TMI. It is supplied with the Java SDK.

*Download the Java SDK*

The Java SDK can be found on the WWW at [http://java.sun.com/j2se/downloads.html](http://java.sun.com/j2se/downloads.html). Follow the link for the most recent version. Then select the link for the SDK for your appropriate platform. Note that JREs will not work for this purpose.

*Install the Java SDK*

Follow the instructions provided by Sun to install the Java SDK.

2.2.2.1 Windows Settings

This section is only relevant if you are going to use a module that employs JNI. Otherwise, you may skip to Visual Studio Settings. If you are going to use JNI, the DLL file for the Java Virtual Machine must appear in your system path. To set this on Windows systems, you must go to the Control Panel (which can be found on the Start Menu of most recent Windows releases). Make sure your Control Panel is in “classic view.” Double click the “System” icon. Now, select the advanced tab. Finally, press the Environment Variables button. This will allow you to change your environment variables including your system path. Your screen should now look similar to Figure 1 (without the Edit User Variable window). Select the variable PATH in the User variables window. If it does not appear use the New button to create it. You should now see the Edit User Variable window. Add the directory (JavaSDKInstallDir)/jre/bin/client to your path. If your Visual Studio IDE is open close it and open it again to refresh the settings.
2.2.3. Flex

Flex is a lexical analysis tool. It can be used to recognize regular expressions. It is currently used in the infrastructure to extract noun phrases using parts of speech. It may prove useful for other components in the future.

*Download Flex*

Finding flex is easier for Linux than for Windows. A Bison/Flex wizard was formerly available on the WWW at [http://www.fg-soup.com/files/bfwizard-1.6.zip](http://www.fg-soup.com/files/bfwizard-1.6.zip) but is now mirrored on our website at [http://hddi.cse.lehigh.edu/bfwizard-1.6.zip](http://hddi.cse.lehigh.edu/bfwizard-1.6.zip).

*Install Flex*

Unzip, run setup.exe, and follow the instructions provided with your Flex package.

2.2.4. MPI

MPI stands for Message Passing Interface, and is a library and system that allow for distributed computing.

*Download MPI*

Install MPI

Execute mpich2.msi and install it to a directory of your choice. For each computer running MPI, you need to ensure that the password selected during the install of MPICH2 is the same. We recommend changing it from the default for security reasons.

Please also note that any application that uses the MPI capabilities of the TMI needs to be executed with mpiexec.exe, included with the download above, in order to use those capabilities.

2.3. Visual Studio Settings

2.3.1. .NET 2003

For TMI to compile you must first configure the location of the libraries you installed in section 2.2. First, open TMI in Visual Studio (by double clicking on the .sln file which can be found at (TMI)/vcpp/libtdm/libtdm.sln). Next, set the global directory and library paths. In your Visual Studio IDE, select the “Tools” menu. Select “Options.” This will open the Options window. You must open the Projects folder which can be found in a list on the left of the window. From the projects folder select “VC++ Directories.” First, set the Include directories. For this select “Include files” in the “Show directories for:” menu.

You must add these directories:
1. (JavaSDKInstallDir)\include\win32
2. (JavaSDKInstallDir)\include
3. (MPIInstallDir)\include
4. (TMI)\lib

This is shown in Figure 3.

Next, select “Library file” in the “Show directories for:” menu.

You must add these directories:
1. (JavaSDKInstallDir)\lib
2. (MPIInstallDir)\lib
3. (TMI)\vcpp\lib

Finally, select “Executable files” in the menu.

You must add the directory which flex.exe appears in (typically c:\BisonFlex\bin on a PC running Windows if you installed the Bison/Flex wizard).
You should now be able to build TMI solution using the ‘Build’ Menu. Please report errors to tmi@cse.lehigh.edu.

After the solution is built, copy all of the dll files from (TMI)\vcpp\lib to your Window’s system32 directory, or place them wherever you expect your DLL files to be found.

2.4. Visual Studio Wizards

These wizards will allow you to get up and running with TMI quickly and easily. They are only compatible with Visual Studio .NET 2003.

2.4.1. TMI Driver Wizard

Locate the install directory for Visual Studio, typically located at C:\Program Files\Microsoft Visual Studio .NET 2003. We are going to copy three files into the Vc7\vcprojects subdirectory of Visual Studio .NET 2003. Copy the following files:

1. (TMI)\wizards\TMI - Driver\TMI - Driver.ico
2. (TMI)\wizards\TMI - Driver\TMI - Driver.vsdir
3. (TMI)\wizards\TMI - Driver\TMI - Driver.vsz

To the Vc7\vcprojects subdirectory. Next, open TMI - Driver.vsz in Notepad, and edit the line regarding ABSOLUTE_PATH to show (TMI)\wizards\TMI - Driver\. This is pictured in Figure 4.
Now, open a new copy of Visual Studio .NET 2003 and select New Project from the File menu. Per Figure 5, you should see TMI - Driver under the Visual C++ Projects section.

2.5. Linux

Due to the many varieties of Linux supported we will not provide instructions here to get the necessary libraries. How to install libraries differs by Linux platform. A list of necessary libraries and links to their websites are provided here:

- Java SDK: http://java.sun.com/j2se/1.4.2/download.html
• Curl (developer version): [http://curl.haxx.se/download.html](http://curl.haxx.se/download.html)
• Flex [http://www.gnu.org/software/flex/flex.html](http://www.gnu.org/software/flex/flex.html)

Once these libraries are installed, go to the directory in which you installed TMI. Once you are in the root of the TMI directory tree, you need to change the configuration script to be executable. To do so use the following command: ‘chmod u+x autogen.sh’.

Next, you must run the configuration script: to do this type ‘./autogen.sh’. This will compile our customized version of Opt++, detect the properties of your system, and create Makefiles.

Finally, type ‘Make’. This should build the library and the appropriate test drivers. This may take a while.
3. TMI

3.1. TdmObj

The base object of all TMI classes is TdmObj. TdmObj is used to count the number of times the given object is referenced. As well, TdmObj allows access to a getDesc() method which is used to find the exact derived type of a TdmObj. This is useful for dynamic checks and debugging.

3.2. TdmPtr

TdmPtr is a reference counting pointer. This is used in many of the interfaces throughout TMI. TdmPtr has a few important caveats, though, that must be understood.

1. TdmPtr is templated and must always have a type
2. TdmPtr may only point to objects on the heap
3. TdmPtr can only point to objects derived from TdmObj and of a polymorphically related type to the templated type
4. TdmPtr can not be directly cast; instead to cast between polymorphic types the following code can be used. In this code the method polyassign (polymorphic assign) is used. The method polyassign always accepts TdmPtr<TdmObj> which is the most generalized version of a TdmPtr. The method generic() is provided with all TdmPtrs to get this most generic version.

```cpp
TdmPtr<A> a;
TdmPtr<B> b(new B());  //B is a subclass of A
a.polyassign(b.generic());
```

3.3. Primitive

TMI offers an abstract type Primitive which represents a union of the four major primitive types used in TMI: bool, int, double, and string. Using this abstract type allows a uniform access to these types. Please see the Primitive class documentation for further detail.
3.4. Descriptors

There are two types of Descriptors TdmObj_Desc and Primitive_Desc. Descriptors are used to describe objects in TMI. This is useful to allow better dynamic understanding and control of the objects.

3.4.1. TdmObj_Desc

This descriptor is used to describe an arbitrary TdmObj. It has four major pieces of information: the interface name, the derived type name, the object name, and optional. Interface name is the name of the interface (ex. Repository). The derived type name describes the most specific type (ex. GenRep). This may be left blank to indicate any derived type from the given interface. The object name is a name that may or may not be provided to the object. This is also used to further identify a derived class. Finally, optional is used to describe whether this item is necessary in a given context.

The following would create a description of an optional GenRep Repository named “REPOSITORY”.

```
TdmObj_Desc("REPOSITORY","Repository","GenRep",true);
```

3.4.2. Primitive_Desc

This descriptor is used to describe a Primitive as it is described above. The Primitive_Desc has four major pieces of information: the primitive name, the primitive type string, the primitive type, and optional. The primitive name is the given name of the primitive. The primitive type string is a textual description of the primitive type. It should be one of the members of the set {"string","bool","int","double"}. The primitive type is one of the defined values {P_STRING, P_BOOL, P_INT, P_DOUBLE}. The last two pieces of information should match (i.e. if the string type is “double” the type should be P_DOUBLE). Finally, optional is used to describe the necessity of the item in the context.

The following would create a description of a non-optional int named ‘COUNT’.

```
Primitive_Desc("COUNT","int",P_INT,false);
```
3.5. Component

TMI is a componentized library. This means the major units that are used to design systems in TMI are components. Components in our system have a very particular definition with the following major points:

(1) A Component is also a TdmObj
(2) A Component must perform at least one specific function
(3) A Component has zero or more of each of the following
   i. Input Pins - receive Primitive input and settings
   ii. Output Pins - allow access to Primitive Results
   iii. Uses Dependencies - provide access to another TdmObj
   iv. Products - allow access to created Tomboys
(4) A Component has a process method which performs the following functions
   i. Check that all input and uses have been defined and are available
   ii. Perform any necessary internal processing
   iii. Make all Output Pins available
   iv. Create all Products
(5) A Component has functions to query what input pins and uses are necessary and what output pins and products are available. It describes the pins using the Descriptors described above.

All components are derived from the abstract class Component which ensures that most of these properties are met. All other properties can be expected from Components and should be strictly enforced when programming a Component. An example Component is an XmlRepBuilder. This component is used to build a Repository (or corpus) from a set of XML files. The XmlRepBuilder has three inputs: location, itemtag, and subtags. The location input specifies a file that has a list of XML files to be read; the itemtag input is used to specify the discriminating tag for documents or items in the files; the subtags input supplies a list of tags to be used to get meta information. The XmlRepBuilder has as a product the repository. This can be visualized as follows.

It is worth noticing some of the details of this illustration. First of all, input pins are shown on the left side of the component by convention. Uses relationships would be shown here as well. Following this convention output pins and products are shown on the right side of the component. Another convention is that primitive types are shown with thin lines ending in diamonds, TdmObj types are shown as thicker lines that end in circles. This can be seen above because the input pins use the former whereas the product uses the latter. These conventions will be used throughout this tutorial.
There are a number of ways components can be connected. The following are the ways and the conventions for illustrating them.

**usesAs**
This causes component A to be used for the uses dependency of component B. The code that describes this connection is:

```
B.usesAs(Buses, A);
```

The convention for drawing this relation is a thick line where one end is a circle. The other end has no symbol indicating it is actually using the object A.

**usesOutputAs**
This connects the output pin of one component (A) to the input pin of another component (B). The code that describes this connection is:

```
B.usesOutputAs(Binput, Ainput, Aoutput);
```

The convention for drawing this relation is a thin line with diamonds on each end to indicate that pins are being used at each end.

**usesProductAs**
This connects the product of component A to the uses dependency of component B. The code that describes this connection is:

```
B.usesProductAs(Buses, Aproduct);
```

The convention for drawing this relation is a thick line where both ends are circles.

### 3.6. Configuration

TMI provides a Configuration object to set the parameters for a component as it is run. Some of the input pins will be connected to output pins from other components and provided input in that fashion. Some of the input pins will be open and still need input, though. Configuration objects provide a means to provide these inputs. A Configuration object is optionally provided to each component when the process function is called. If an input has not been provided it will then be checked for in the Configuration. Therefore, the Configuration object provides a way to dynamically change settings for objects and reuse system architectures with different settings. Configuration also provides a simple way to loop or optimize over inputs for systems as will be described later.
One can set a parameter in 3 different ways to a configuration: singular, range, and set. The code for each of these is shown below.

```cpp
Configuration cfg("my configuration")
cfg.setParameter("PARAMETER",value)         //1
cfg.setParameterRange("PARAMETER",startvalue,endvalue,stepvalue) //2
vector<Primitive> vp;
v.push_back("value1"); vp.push_back("value2"); vp.push_back("value3");
cfg.setParameterSet("PARAMETER",vp);         //3
```

A singular parameter is when a parameter is given one value (see example 1). This is used in most cases.

A range parameter is when a parameter can have any of a range of values. This range is inclusive between startvalue and endvalue. The minimum granularity is stepvalue. For example, if startvalue = 1, endvalue = 2, and stepvalue = .25 then the possible values are (1,1.25,1.5,1.75,2).

A set parameter is used when a parameter can take on a set of values.

For the latter two types the first value is used by default when a single value is expected. This default can explicitly be changed and is changed during looping and optimization automatically.

### 3.7. TdmComponentPath

Once the data connections between components are described there is still the issue of designing the execution of these components. This is currently designed using a concept known as a “path”. A path describes a sequential execution of a number of components. That is if I have a path A->B->C component A is processed, then component B is processed, and finally component C is processed. It is important to note that although some of the diagrams (especially of the data dependencies) are drawn in parallel at this time all execution is sequential.

With cursory inspection it becomes obvious that the serial execution of a set of components can itself be thought of as a component. There are a number of open input pins that need input and a number of available output pins. Thus, the input pins for the path could be thought of as the union of the open input pins of each of the components in the path. The same is true for output pins, uses dependencies, and products.
This is where `TdmComponentPath` becomes useful. `TdmComponentPath` is derived from `Component` and therefore is itself a component. The main purpose of `TdmComponentPath` is to manage the serial execution of other components in the system. It also provides the capability to loop and optimize over parameters on a particular component. This functionality is particularly useful since `TdmComponentPath` is a component, which allows one to create a rather complex recursive system.

### 3.7.1. Sequential Execution

The most basic of operations of `TdmComponentPath` is to add components for serial execution. This is done through the `addSequential()` method. An example is shown below.

```java
TdmPtr<A> a = new A("my first component");
TdmPtr<B> b = new B("my second component"); // A and B are derived from Component
b.usesAs("Buses", a->generic());
TdmPtr<Configuration> cfg = new Configuration("my configuration");
TdmPtr<TdmComponentPath> path = new TdmComponentPath("my path");

path->addSequential(a->component(),cfg);
path->addSequential(b->component(),cfg);

path->process();
```

When `path` is processed on the last line this will process `Component a` using configuration `cfg` and then process `Component b` using `Configuration cfg`. The configuration is optional but in this case both components use a `Configuration`. Notice the use of the method `component()` to get a `TdmPtr` generalized to `TdmPtr<Component>`. Also note that now the code

```java
path->addSequential(path->component());
```

would be perfectly valid though it would of course cause problems at runtime. The path above could be visualized as follows.
Notice the TdmComponentPath is represented by an external box and that dotted lines are used to show the path of execution. Also, notice that there can still be data relationships between the components in the TdmComponentPath. For ease of coding it is strongly suggested their not be relationships that range outside of the TdmComponentPath.

As well, the name of the TdmComponentPath is provided in the upper left hand corner. This is so that in later diagrams the TdmComponentPath can be used as a stand alone component. That is it could be referred to as:

![Path diagram]

### 3.7.2. Configuring Input & Output pins

It was discussed above how a path could be considered to have as input pins the union of all the internal Component’s input pins. While this is true in theory, a better solution is to have the programmer manually create connections between inputs for the path and inputs for components within. This circumvents a number of issues especially that of similarly named component input pins. It also makes the data flow more explicit. TdmComponentPath has two methods to allow exactly this.

Going back to the example above lets assume that component ‘A’ has input pins “AI1”, “AI2”, “AI3” and output pins “AO1”, “AO2”. It also has uses dependencies “AU1”, “AU2” and product “AP1”. Lets also assume that component ‘B’ has three input pins “BI1”, “BI2”, “BI3” and output pins “BO1”, “BO2”. It also has uses dependencies “BU1”, “BU2” and a product “BP1”. Component A is visualized below and component B would look similar.
The following would allow path to have two input pins “AI1”, “BI1” and two output pins “AO1”, “BO1” as well as a uses dependency “AU1” and a product “BP1”. Note that all the input/output pins are of integer type in this example. All products/uses are Repositories.

```cpp
path->passInput(Primitive_Desc("AI1","int",P_INT),cfg, "AI1");
path->passInput(Primitive_Desc("BI1","int",P_INT),cfg, "BI1");

path->passOutput(Primitive_Desc("AO1","int",P_INT),a->component(), “AO1”);
path->passOutput(Primitive_Desc("BO1","int",P_INT),b->component(), “BO1”);

path->passUses(TdmObj_Desc("AU1","Repository",""),a->component(), “AU1”);
path->passProduct(TdmObj_Desc("BP1","Repository","GenRep"),b->component(),"BP1");
```

Notice that Inputs are passed to an existing configuration going into the component. This standardized the input received.

This would change the visualization of path to the following:

3.7.3. Looping

TdmComponentPaths also have the ability to loop over a parameter or a uses dependency for a particular component. Only one component can be in the loop or optimization but
because a TdmComponentPath can present a single component interface to a number of components this does not present a problem. It is possible to loop either over the values of a Configuration parameter (only if the parameter is of set or range type). It is also possible to loop over a uses dependency. This means it is just as easy to loop over a parameter value from 1 to 100 with a step of .5 and loop over different FeatureExtractors.

To implement looping using a TdmComponentPath a LoopConfiguration class is used. How the LoopConfiguration is configured depends on which type of looping is going to occur. This is explained in detail in the two following sections “Looping Over a Parameter” and “Looping Over Uses Dependencies”.

A loop is added in the sequential processing order of a TdmComponentPath. Therefore, there can be components before and after a loop. Below is another version of the code example provided above that demonstrates looping.

```java
... see above ...

path->addSequential(a->component(),cfg);
path->addSequentialLoop(b->component(),cfg, new LoopConfiguration(…) );
path->process();
```

This example will first process component a and then loop over some parameter or uses dependency in component b.

### 3.7.3.1. Looping Over a Parameter

To loop over a set of parameter values the LoopConfiguration object is constructed with the following constructor:

```
LoopConfiguration(TdmPtr<Configuration> cnf, const String &var);
```

In this constructor cnf represents the configuration where the parameter is and var represents the name of the parameter. The parameter must be of the range or set type. So, for example, to loop over “BI1” the code from above would look like this:

```java
... see above ...

cfg->setParameterRange("BI1",1.0,100.0,0.5);
path->addSequential(a->component(),cfg);
path->addSequentialLoop(b->component(),cfg, new LoopConfiguration(cfg,"BI1") );
path->process();
```

This code would first process component a and then process component b with BI1 equal to all values from 1.0, 1.5, 2.0, … , 100.0 .

©2000-2004 Parallel and Distributed Text Mining Lab, CSE Department, Lehigh University
3.7.3.2. Looping Over Uses Dependencies

To loop over a set of uses dependencies the LoopConfiguration object is constructed with the following constructor:

```cpp
LoopConfiguration(const String &var);
```

In this constructor var represents the name of the uses dependency. It is important to note that the String only constructor defaults to uses dependency mode. An example of this method of looping is provided below.

```cpp
... see above ...

vector<TdmPtr<TdmObj>> vo;
vo.push_back(new Rep1);
vo.push_back(new Rep2);
vo.push_back(new Rep3); //If Rep1, Rep2, and Rep3 were all Repositories
b->usesAs("BU2",vo);

path->addSequential(a->component(),cfg);
path->addSequentialLoop(b->component(),cfg, new LoopConfiguration("BU2") );

path->process();
```

This code would first process component “a” and then process component ‘b’ with BU2 using Rep1, Rep2, and Rep3. Notice that this is only effective if a vector of uses is provided. If only one option is provided it will execute exactly the same as the equivalent addSequential.

3.7.4. Optimization

Many of the algorithms applied in TDM have a number of parameters whose optimal values are either unknown or vary by application. Therefore, it is crucial that one be able to optimize these parameters in an efficient way. TdmComponentPath has the ability to use an OptMethod derived class to perform such an optimization over a number of parameters exposed by a component.

The definition of an optimization process for TMI is as follows. There is some component A which has a set of parameters $I_1, I_2, ..., I_N$. It also has Primitive outputs $O_1, O_2, ..., O_N$. To create an optimization problem one can view the component as a set of abstract functions. $A_I (I_1, I_2, ..., I_N) = O_1$, etc. Only one of these functions can be optimized at this time. Since these functions correlate with a target output only one of the $O$-set can be selected to be optimized. As well, a subset of the $I$-set can be selected to limit the dimensions of the optimization.
To define the optimization problem one must create an OptConfiguration. An OptConfiguration is significantly more complex than a LoopConfiguration because of the additional concerns. All of the inputs must be range parameters that appear in the Configuration that is passed to the Component being optimized. At this time set parameters are not supported. OptConfiguration has the following constructor:

```cpp
OptConfiguration(vector<String> &var, TdmPtr<Configuration> cnf,
                  TdmPtr<OptMethod> opt, const String &evalvar, bool max=true)
```

Descriptions of the parameters are as follows:

- `var` - a vector of the input parameters to use (correlates with the subset of the I-set)
- `cnf` - the configuration in which the inputs appear
- `opt` - a class derived from OptMethod that will perform the optimization
- `evalvar` - the name of the target variable
- `max` - should the value be maximized or minimized

So for the code sample above Optimization would look like this:

```cpp
... see above ...

cfg->setParameterRange("BI1",1.0,100.0,0.5);
cfg->setParameterRange("BI2",1.0,100.0,0.5);
cfg->setParameterRange("BI3",1.0,100.0,0.5);

vector<String> useVars;
useVars.push_back("BI1");
useVars.push_back("BI2");
useVars.push_back("BI3");

    path->addSequential(a ->component(),cfg);
    path->addSequentialLoop(b ->component(),cfg,
      new OptConfiguration(useVars,cfg, new QNewton(), "BO1") );

    path->process();
```

This code would first process component a and then optimize over the parameters BI1, BI2, and BI3 using component b’s output “BO1” as the result. This optimization uses the QNewton OptMethod which is a bound-constrained quasi-Newton optimization approach.
4. An Example System

4.1 Before You Begin

If, in reading this section of the tutorial, you come across terms that are unfamiliar to you it is suggested you read Appendix 2 which has a brief description of the interfaces used by TMI. These descriptions will help to explain some of the more common terminology used.

4.2 System Description

The example system that will be described here is that implemented in the ClusterTest driver that is supplied with TMI. The ClusterTest driver performs the following operation:

1. Load a set of files
2. Extract words from the files
3. Create a graph from these words using a co-occurrence metric for arc weight
4. Partition this graph to create semantic clusters of the words

This process has a number of known applications including emerging trend detection and document clustering.

4.3 Relevant Components

A number of components will be used to build this system. A brief description is given about these components here for more information see Appendix 1.

FileRepBuilder - Builds a Repository from a list of files
CCTagger - Brill’s part of speech Tagger
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 Tarjan’s algorithm.
ModelBuilder - Coordinates the unsupervised feature clustering process
4.4 Assembling the Components

The following diagram represents the dataflow for the system.

The following diagram represents the execution flow for the system.

Notice that TarModel and CoNetImp are not in the execution flow. This is because they are implicitly run by ModelBuilder. This is an anomaly and in almost all cases it is required that a component occurs earlier in the execution flow than all items that are dependant on it. That is, an item must be processed before it is used.

4.5 Writing the Code

Once a data flow and execution flow have been designed the next step is to actually implement the design using TMI. The issues of creating a new driver have been omitted here and can be found in section 6. This will only focus on the code which can be found in ClusterTest.cpp.
4.5.1 Before

The first step of writing a TMI component is to add the appropriate header and comment information. This is found in the following code segment.

```cpp
/**
 * \example ClusterTest.cpp
 * \brief An example of term clustering
 *
 \verbatim
 ClusterTest.cpp
 TMI Lehigh University HDDI
 Copyrighted by Lehigh University

 THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

 INITIAL REVISION: Lars Holzman 1.0 ( Summer 2003 )
 LAST REVISION: $Id: ClusterTest.cpp,v 1.7 2003/07/30 21:54:49 leh7 Exp $ 1.0 ( July 19, 2002 )
 \endverbatim
 */

#include "tdm/tdm_config.h"
#include <iostream>
#include <fstream>
#include "tdm/components/execution/TdmComponentPath.h"
#include "tdm/repository_builder/FileRepBuilder.h"
#include "tdm/tagger/CCTagger.h"
#include "tdm/feature_extractor/Word.h"
#include "tdm/feature_generator/GenFg.h"
#include "tdm/ifs/GenIfs.h"
#include "tdm/semantic_network/CoNet.h"
#include "tdm/semantic_model/TarjanModel.h"
#include "tdm/ml_driver/ModelBuilder.h"

using namespace std;
using namespace tdm;
```

Notice that the initial documentation block provides necessary copyright and version information. The backslash commands are doxygen directives to ensure proper automatic documentation.

After the comment block all necessary headers are included. All TMI files should have the header “tdm/tdm_config.h” first which sets various options for TMI. Also, notice that the headers included correspond with the components described above and TdmComponentPath which will be used for execution flow.

Next, the proper namespaces must be defined. Using namespaces is generally considered risky practice but for our purpose is adequately safe.
Two namespaces are used std and tdm. The namespace std is used to allow access to the STL classes. All TMI classes are in namespace tdm and so that namespace was used as well.

In the main method, some IO must occur before the system is designed. This data will represent the input to the open input pin “LOCATION”(input).

```cpp
int main() {
    String input, output;
ofstream outf;

    cout << "Cluster Generator" << endl;
    cout << "Written using TMI\n" << endl;
    cout << "Enter name of input (list) file: ";
    cin >> input;

    TdmPtr<RepositoryBuilder> rm(new FileRepBuilder("File Repository Builder"));
    TdmPtr<Configuration> cRm = new Configuration("File Repository Builder Configuration");
    cRm->setParameter("LOCATION", input);

    TdmPtr<Tagger> cc(new CCTagger("POS tagger"));
    TdmPtr<FeatureExtractor> fe(new WordExtractor("Word Extractor"));
    TdmPtr<FeatureGenerator> fg(new GenFg("Generic Feature Generator"));
    fg->usesAs("TAGGER", cc.generic());
    fg->usesAs("FEATURE_EXTRACTOR", fe.generic());

    . . .
```

4.5.2 Data Flow Description

The next step is to describe the data flow from the diagram above using TMI components. This can be done in any order as long as all components exist before they are declared a uses dependency. Configurations are also declared here to account for the open pins. The first Component in the diagram is the FileRepBuilder.

Notice that as well as creating the FileRepBuilder the input received above is placed in a configuration that it will later be processed with.

Next, the CCTagger, WordExtractor, and GenFG components are defined.
Notice the use of usesAs as described above. Also, it is important to notice that this description is the exact parallel of the figure above. One simply describes the diagram piece by piece using the component architecture.

Next, the GenIfs is defined.

```cpp
TdmPtr<ItemFeatureSet> ifs(new GenIfs("Generic IFS"));
ifs->usesProductAs("REPOSITORY","rm","REPOSITORY");
ifs->usesAs("FEATURE_GENERATOR",fg.generic());
```

Notice the use of usesProductAs to connect the product of the FileRepBuilder with the uses dependency of the ifs.

Finally, the model building components are defined.

```cpp
TdmPtr<SemanticNetwork> sn(new CoNetImp("Co-occurrence semantic network"));
sn->usesAs("IFS",ifs.generic());

TdmPtr<SemanticModel> sm(new TarModel("Tarjan's algorithm clustered model"));
sm->usesAs("IFS",ifs.generic());
sm->usesAs("NET",sn.generic());

TdmPtr<Configuration> mbConf = new Configuration("Semantic Network/Model Configuration");
mbConf->setParameter("THRESHOLD", 0.005);
mbConf->setParameter("ALPHA", 1.65);

TdmPtr<ModelBuilder> mb(new ModelBuilder("ModelBuilder"));
mb->usesAs("SEMANTIC_NETWORK",sn.generic());
mb->usesAs("SEMANTIC_MODEL",sm.generic());
```

This follows the same method of representing the diagram with the component architecture. It is also interesting to notice how the configuration simply provides specified values for the two remaining input pins.

After these the data definitions are complete.

### 4.5.3 Execution Flow Description

The execution flow for this system is very simple. It is simply a sequential ordering of the modules without any looping or optimization. Thus, to create this path all of the relevant components are added to the TdmComponentPath in the order they occur in the execution path and then the TdmComponentPath is processed.
The execution path is placed in a try catch block because it is possible that one of the
STL classes may throw an exception or the TMI classes may throw a TdmException or
ConfigurationException all of which should end execution and report the error.

4.5.4 After

After the normal flow of execution of this system has ended some other code is present to
output the clusters that are generated. This code will be ignored in this tutorial because it
is irrelevant to designing a system. It is important to remember that any of the interface
functions of any of the components are still valid before or after execution and can be
used to further customize the systems.
5. Creating Your Own Driver

5.1. Adding a Driver/Test

Drivers are added to the (TMI)\tests directory. The first step to creating a driver is to make a directory for it. This directory will look something like (TMI)\tests\MyDriver.

5.2. Using the Wizard for Visual Studio

Go to File -> New Project in Visual Studio .NET 2003. If you want to add the project to an existing solution, right click on the solution and choose New Project.

The Wizard screen will come up. Choose Use JNI if you installed the Java SDK per the instructions in section 2. Choose to include the default main.cpp file if you want to start with a template that will compile immediately, or choose not to include it if you just want a project with the proper settings for using TMI. The latter would be more appropriate if you are, for example, importing a project from Linux.
The resulting project will have both Debug and Release modes set to link to the TMI library and have all settings set properly to have that link succeed. If you chose to include the main.cpp file, the project will compile immediately and is ready to be modified.

5.3. Updating Makefiles

There are a few parts to adding your driver to the makefile system. This is because there are mechanisms in place to ensure your driver is only compiled if the proper libraries are installed.

The first step is to add a Makefile.am to the directory you have created. This is a template that will be used by the automake system to correctly configure your driver. The template of this file is as follows (this can also be found at (TMI)/test/Makefile.tmpl).

```
bin_PROGRAMS= [DriverName]
[DriverName]_SOURCES= [FileList]
[DriverName]_LDADD = @top_srcdir@/lib/tdm/libtdm-1.0.la @top_srcdir@/lib/OPT++2.1/lib/libopt-linux.a @top_srcdir@/lib/OPT++2.1/lib/libnewmat-linux.a

[DriverName]_LDFLAGS = $(CUSTLIBS)
LIBS =
INCLUDES=-I@top_srcdir@/lib
```
Obviously, each of the [DriverName]’s above should be replaced by the name of your driver and [FileList] should be replaced by the list of files used by your driver. Drivers that have subdirectories are not covered by the tutorial at this time.

The next step is to add this Makefile.am to the configuration script so that the Makefile is automatically generated. There are a few other changes that must occur in this script as well. To make these changes open the file ‘(TMI)/configure.in”. The first section you want to modify is called the “driver checks” section. This section verifies if all of the proper libraries exist before compiling a driver. A portion of this section follows.

\begin{verbatim}
dnl-----------------------
dnl Perform driver checks
dnl-----------------------
AM_CONDITIONAL(CLUSTER_TEST_OPT, [test x = x])
AM_CONDITIONAL(GENIFS_TEST_OPT, [test "$ac_cv_mysql" = yes])
AM_CONDITIONAL(SHOW_FILES_TEST_OPT, [test "$ac_cv_curl" = yes -a "$ac_cv_mysql" = yes])
\ldots
\end{verbatim}

Notice that this section can be found by searching for “driver checks”. You must add a line like the three examples above for your driver. The examples above were chosen because they show some different tests. ClusterTest requires no special libraries so a test is chosen that always returns true. GenIfsTest requires MySQL so the variable $ac_cv_mysql is tested for. ShowFiles requires both MySQL and Curl so both variables are tested using the and ( -a ) operator. The following variables are available:

\begin{itemize}
  \item \$ac_cv_mysql \quad \text{Is MySQL available}
  \item \$ac_cv_curl \quad \text{Is Curl available}
  \item \$ac_cv_jni \quad \text{Is JNI available}
\end{itemize}

Make sure the identifier you choose for the first argument is distinct and closely mirrors your driver name. It should follow the convention seen in the file which is all upper case letters and underscores and ending with OPT.

In the next step you add your Makefile.am to the list mentioned above. To do this find the section beginning with “AC_OUTPUT(“ which clarifies what Makefiles should be produced. Going to the bottom of this list you should notice that the Makefiles begin to have names in the test directory. Add the location of your Makefile.am with the “.am” removed.

Finally, you must modify the “test/Makefile.am” so that the Makefile system knows to recursively descend into your directory. It should only do so if the test designed above passes. Thus, you must add a conditional block to “test/Makefile.am”

\begin{verbatim}
CLUSTER_TEST =
if CLUSTER_TEST_OPT
  CLUSTER_TEST = ClusterTest
endif
\end{verbatim}
Notice that this code defines a new variable named the same as the test variable without
the “_OPT” at the end. This variable is first initialized to nothing and then conditionally
set to the directory which your driver is in. You should be able to write a similar block
for your driver by analogy to the example above.

Finally, add the name of the variable created above to the end of the “SUBDIRS”
variable list. Let’s assume the name of your variable is MY_VAR. The “SUBDIRS” list
looks like this:

```
SUBDIRS= $(CLUSTER_TEST) $(EBT) $(FFE) $(GENIFS_TEST) $(ETD_TEST) $(ETDLSI_TEST)
 $(COLLOCATION_TEST) $(SHOW_FILES_TEST) $(BIBLEMINE_TEST) $(OPT_METHOD_TEST)
 $(DB_TEST) $(ME_SENT_EXTRACTOR) $(ME_SENT_DRIVER) $(MY_VAR)
```

After you add your variable it would look like this:

```
SUBDIRS= $(CLUSTER_TEST) $(EBT) $(FFE) $(GENIFS_TEST) $(ETD_TEST) $(ETDLSI_TEST)
 $(COLLOCATION_TEST) $(SHOW_FILES_TEST) $(BIBLEMINE_TEST) $(OPT_METHOD_TEST)
 $(DB_TEST) $(ME_SENT_EXTRACTOR) $(ME_SENT_DRIVER) $(MY_VAR)
```

Finally, return to the root directory and execute “./autogen.sh” and “make” as explained
in the Linux installation directions.

5.4. Writing the Code

There is no special trick to writing the code for your project. The best method is probably
to follow the process illustrated above. First, map your system onto the components
provided by TMI. If there is one or more components missing see section 6 for
information about creating them. Once all components have been created draw a data
flow diagram describing your system. Then, draw the execution flow diagram. It should
then be trivial to design the code for the system. The system may also have pre or post
processing steps which are not reflected in the diagrams.
6. Creating Your Own Component(s)

6.1. Choose Your Interface

The most important part about creating a component is choosing the correct mapping to the infrastructure. Before creating a component it is strongly recommended that you read the first two Appendices very carefully and spend some time examining this tutorial, the doxygen, and the TMI itself. After this you may have some idea where your component fits into the infrastructure. If you are still having problems it is possible that you will need to divide the functionality of your component into a number of components that better map to the TMI. It is also possible you will have to combine more functionality into your component. Before proceeding make sure you have a clear picture of what interfaces your components will implement. Also make sure you know what method you must implement to satisfy these interfaces and how these methods map to your component(s).

6.2. Adding a Component

The next step is to make space for your component. From now on the interface your component will be implementing will be referred to as [interface]. Components are always stored in the directory (TMI)/lib/tdm/[interface]/[component name]. The only exceptions are when components fit into subclasses of interfaces (ex. collocations). So, you should create this directory. You should now have a directory that looks something like (TMI)/lib/tdm/[interface]/MyComponent. All source code for your component should go in this directory. The one exception is the header file which defines the subclass of the interface this will be named [component name].h and should go in the (TMI)/lib/tdm/[interface] directory.

6.3. Adding to Your Visual Studio Solution

This step assumes you already have files containing the code for your component. If you do not create empty files representing the files you expect to have.

To add the component to your Visual Studio workspace is a fairly trivial process. Open the solution file (TMI)/vcpp/libtdm/libtdm.sln. There will be a project in that solution named libtdm. This project has folders for each interface. Find the interface you have chosen for your component. Right click the folder for this interface and choose add existing item. Select your header file ([component name].h). Next, you must add a folder for your component. Right click the folder for the interface again and choose “New Folder”. This will create a new folder. Rename this folder to the name of your
component. Finally, right click the folder you have created and choose “Add existing item” and add all relevant files.

6.4. Updating Makefiles

To update the Makefile system to compile your new component you must first add a Makefile.am file to the component directory. This file can be built using the template provided at (TMI)/lib/tdm/Makefile.tmpl. The contents of this file are also displayed below.

```latex
noinst_LTLIBRARIES= lib[ComponentName].la
lib[ComponentName]_la_SOURCES= [FileList]
INCLUDES=-I@top_srcdir@/lib
```

In this template [ComponentName] should be replaced by the name of your component and [FileList] should be replaced by the names of the files you will use to code your component. The value that you set the first line to (i.e. lib[ComponentName].la) is the library that will be generated from your component.

The next step is to modify the (TMI)/lib/tdm/[interface]/Makefile.am. This file will look similar to the example file (for the interface MIAdapter) shown below.

```latex
SUBDIRS= NFoldSplit ResampleAdapter
h_sources= NFoldSplit.h ResampleAdapter.h
library_includedir=$(includedir)/$(GENERIC_LIBRARY_NAME) -
$(GENERIC_API_VERSION)/$(GENERIC_LIBRARY_NAME)/ml_adapter
library_include_HEADERS=$(h_sources)
INCLUDES=-I$(top_srcdir)/lib
```

The two variables of interest are SUBDIRS and h_sources. You must add the directory of your component (which should be the component name) to SUBDIRS and the header file for your component to h_sources. If your component was named MyComponent and was an MIAdapter, the new lib/tdm/ml_adapter/Makefile.am would appear as follows after you updated it.

```latex
SUBDIRS= NFoldSplit ResampleAdapter MyComponent
h_sources= NFoldSplit.h ResampleAdapter.h MyComponent.h
library_includedir=$(includedir)/$(GENERIC_LIBRARY_NAME) -
$(GENERIC_API_VERSION)/$(GENERIC_LIBRARY_NAME)/ml_adapter
library_include_HEADERS=$(h_sources)
INCLUDES=-I$(top_srcdir)/lib
```

Next, you must modify the lib/tdm/Makefile.am to have it add the library that will be created to libtdm. To do this open lib/tdm/Makefile.am. You will notice that there is a variable “libtdm_1_0_la_LIBADD” which is followed by a list of library names that mirror the components in the system (and thus also the directory structure).

```latex
libtdm_1_0_la_LIBADD= attribute/AttString/libAttString.la \ attribute/ClusterSize/libClusterSize.la \ ...
ml_adapter/NFoldSplit/libNFoldSplit.la 
ml_adapter/ResampleAdapter/libResampleAdapter.la 
ml_algorithm/WekaDT/libWekaDT.la 
```
Find the proper location for your library and add it to the list followed by a backslash (which indicates to continue to the next line. Continuing our example from above the library for MyComponent which is an MIAdapter has been added below.

```
libtdm_1_0_la_LIBADD= attribute/AttString/libAttString.la \attribute/ClusterSize/libClusterSize.la \...
  ml_adapter/NFoldSplit/libNFoldSplit.la 
ml_adapter/ResampleAdapter/libResampleAdapter.la \ml_adapter/MyComponent/libMyComponent.la \ml_algorithm/WekaDT/libWekaDT.la \...
```

Finally, you must add your Makefile to the list of makefiles that will be created. To do this you must edit (TMI)/configure.in. Find the section beginning with “AC_OUTPUT(['“. This section clarifies what Makefiles should be produced. This file again mimics the directory structure. Find the proper location for your Makefile (where your Makefile.am currently resides) and add the (TMI)/lib/tdm/[interface]/[component name]/Makefile. This will be the location of the Makefile.am you created without the “.am” on the end.

Finally, return to the (TMI) directory. Run “./autogen.sh” and “make” as described in the Linux installation instructions.

### 6.5. Writing the Code

When writing code for a component you have the obligation to implement all of the virtual abstract methods of both the interface that you have chosen and the generic interface for a component. Some of the methods required by the component interface will be nearly identical across all derived classes. This is why a utility class TdmComponent has been added.

### 6.6. Using TdmComponent

While TdmComponent does implement some of the Component interface methods there are a number of functions that are still left to implement. These methods are:
Essentially it leaves the method for processing the component, methods for getting information about the component and methods to get the results of the component. The input methods are taken care of in TdmComponent.

To access these inputs a few utility functions are provided. These utility functions are:

```cpp
void processInput(const String&, const Configuration&, bool& ) const throw (ConfigurationException)
void processInput(const String&, const Configuration&, int& ) const throw (ConfigurationException)
void processInput(const String&, const Configuration&, double& ) const throw (ConfigurationException)
void processInput(const String&, const Configuration&, String& ) const throw (ConfigurationException)

bool hasInput(const String &name, const Configuration &con)

TdmPtr<TdmObj> processUsed(const String &name) throw (ConfigurationException)
```

As you can see 3 main methods are implemented: processInput, hasInput, and processUsed. These are the only methods that should be used to access the input given to a TdmComponent.

The method processInput is used to get a Primitive value from an input pin. The type used must match the type provided to the input pin. This method is useful because it helps coordinate checking the configuration provided as well as the possible connection with another component. If the input is not found anywhere a ConfigurationException is thrown.

The method hasInput is used to detect if the input is available. Any input declared optional should use hasInput to detect if it is there. If it is it should be processed if it isn’t a default value should be set. A code example of this is provided below. The method hasInput is also useful for custom error handling of missing inputs.

```cpp
if (hasInput("SEED",config)) {
    processInput("SEED",config,seed);
} else {
    seed = 42;
}
```

The method processUsed is similar to processInput except due to the fact it only ever gets one type that type is returned and the only exception that is thrown is if the used object is not available. No hasUsed command is available because uses can not be optional.
Appendix A - Component List

This list is valid as of TMI release 1.0 (July 31, 2003).
See http://hddi.cse.lehigh.edu/CompDoc/toc.html for more recent documentation.

A.1 Special

TdmComponentPath
This component is used to implement a serial execution path of components. It is the key component responsible for handling looping and optimization. It allows a debug mode to view execution. It has additional capability to pass through inputs and outputs.

Inputs: None
Outputs: None
Uses: None
Products: None

StreamWriter
This component is used to write a primitive to cout.

Inputs: None
Outputs: None
Uses: None
Products: None

SchemaConvertor
Inputs: None
Outputs: None
Uses: REPOSITORY (Repository::)
SCHEMA (Schema::)
Products: REPOSITORY(Repository::)

DBWriter
This writes a repository to a MySQL database.

Inputs: SERVER (string)
DATABASE (string)
USERNAME (string)
PASSWORD (string)
TABLE (string)
Outputs: None
Uses: REPOSITORY (Repository::)
A.1 Special (cont.)

XmlWriter
This writes a repository to an XML database form

Inputs:   DOCTAG (string)
         ITEMTAG (string)
         FILENAME (string)

Outputs: None
Uses:     REPOSITORY (Repository::)
Products: None
A.2 FeatureExtractor

**NounPhraser**
Extracts noun phrases using POS tags. This uses flex to find noun phrases using a RegEx. This should be used with CCTagger.

Inputs: None  
Outputs: None  
Uses: None  
Products: None

**NullExtractor**
This is a placeholder extractor. It does not find features.

Inputs: None  
Outputs: None  
Uses: None  
Products: None

**StemWordExtractor**
This is a word extractor with two additions. First, it uses Porter's stemmer to stem the words. Second, it removes stop words.

Inputs: None  
Outputs: None  
Uses: None  
Products: None

**WordExtractor**
This is a simple word extractor.

Inputs: None  
Outputs: None  
Uses: None  
Products: None

**HypTestColExt**
This extracts Hypothesis Tested collocations. The collocations are generated from the supplied repository and then found in the text.

Inputs: TEST (string) [optional]  
         THRESHOLD (double) [optional]  
Outputs: None  
Uses: REPOSITORY (Repository::)  
Products: None
A.2 FeatureExtractor (cont.)

MeanVarColExt
This extracts Mean/Variance collocations. The collocations are generated from the supplied repository and then found in the text.

Inputs:
   WINDOW_SIZE (int) [optional]
   THRESHOLD (double) [optional]
   COVERAGE (int) [optional]

Outputs: None
Uses:    REPOSITORY (Repository::)
Products: None

POSColExt
This extracts Part of Speech collocations using a number of regular expressions.

Inputs: None
Outputs: None
Uses: None
Products: None

NaiveSentenceExt
Inputs: MERGE_SIZE (int) [optional]
Outputs: None
Uses: None
Products: None
A.3 FeatureGenerator

GenFg
This is a general feature generator to coordinate tagging and feature extraction.

Inputs: None
Outputs: None
Uses: TAGGER (Tagger::)
      FEATURE_EXTRACTOR (FeatureExtractor::)
Products: None
A.4 ItemFeatureSet

FileIfs
This loads an IFS from a file.

Inputs: ITEM_COUNT (int) [optional]
FILENAME (string)
Outputs: ITEM_COUNT(int)
FEATURE_COUNT(int)
Uses: None
Products: None

GenIfs
This is a general purpose IFS.

Inputs: None
Outputs: ITEM_COUNT(int)
FEATURE_COUNT(int)
Uses: REPOSITORY (Repository::)
FEATURE_GENERATOR (FeatureGenerator::)
Products: None
A.5  MIAdapter

NFoldSplit
This adapter splits the supplied 'TRAINING_SET' into a training and test fold using a stratified N-fold cross-validation.

Inputs:  NUM_FOLD (int)
         FOLD (int) [optional]
         STRATIFIED (bool) [optional]
         SEED (int) [optional]

Outputs: None

Uses:   TRAINING_SET (TrainingSet::)

Products: TRAIN_FOLD(TrainingSet::BasicTrainingSet)
          TEST_FOLD(TrainingSet::BasicTrainingSet)

ResampleAdapter
This adapter resamples a particular class in the training set provided.

Inputs:  TIMES (int) [optional]

Outputs: None

Uses:   TRAINING_SET (TrainingSet::)
         CLASS (Attribute::)

Products: TRAINING_SET(TrainingSet::BasicTrainingSet)
A.6 MIAlgorithm

**BinGis**
This is a machine learning algorithm to apply binary general iterative scaling. It is used in the Maximum-Entropy Modeling approach.

Inputs: ALPHAFILE (string)
Outputs: None
Uses: TRAIN_SET (TrainingSet::)
Products: None

**WekaDT**
This wraps the Weka j4.8 decision tree class.

Inputs: UNPRUNED (bool) [optional]
CONFIDENCE (double) [optional]
SUPPORT (int) [optional]
FOLDS (int) [optional]
R_E_PRUNE (bool) [optional]
BINARY_SPLITS (bool) [optional]
SUBTREE_RAISING (bool) [optional]
LAPLACE_SMOOTHING (bool) [optional]
Outputs: METHOD(string)
Uses: None
Products: None
A.7 MlDriver

ModelBuilder
This is a machine learning driver to build a clustering of features from an IFS

Inputs: None
Outputs: None
Uses: ALGORITHM (MlAlgorithm::)
      TRAINING_SET (TrainingSet::)
      TEST_SET (TrainingSet::)
Products: SEMANTIC_MAP(SemanticMap::GenMap)

WekaDriver
This is a machine learning driver to coordinate machine learning using the Weka library

Inputs: None
Outputs: PRECISION(double)
         RECALL(double)
         FMEASURE(double)
Uses: ALGORITHM (MlAlgorithm::)
      TRAINING_SET (TrainingSet::)
      TEST_SET (TrainingSet::)
Products: PRECISION(MIEvaluation::PrecisionEvaluator)
         RECALL(MIEvaluation::RecallEvaluator)
         FMEASURE(MIEvaluation::FMeasure)
A.8 MIEvaluation

**MeanEvaluator**
This uses a number of results and supplies as its result the mean (or average) of them.

Inputs: RESULT (double)
RESET (bool)
Outputs: RESULT(double)
Uses: None
Products: None

**FMeasure**
This is a metric finding the trade off between precision and recall. (Beta = 1)

Inputs: TP (int)
FP (int)
FN (int)
TN (int)
Outputs: RESULT(double)
MATRIX(string)
Uses: None
Products: None

**PrecisionEvaluator**
This is the standard precision metric (TP/TP+FP).

Inputs: TP (int)
FP (int)
FN (int)
TN (int)
Outputs: RESULT(double)
MATRIX(string)
Uses: None
Products: None

**RecallEvaluator**
This is the standard recall metric (TP/TP+FN).

Inputs: TP (int)
FP (int)
FN (int)
TN (int)
Outputs: RESULT(double)
MATRIX(string)
Uses: None
Products: None
A.9 RepositoryBuilder

XmlRepBuilder
This builds a repository from a set of XML files.

Inputs:
LOCATION (string)
ITEMTAG (string)
SUBTAGS (bool)

Outputs: None
Uses: None
Products: REPOSITORY(Repository::GenRep)

BadXmlRepBuilder
This builds a repository from a set of (bad) XML files. Previous versions of our corpus builder tool produced non-standard XML. This component is used to read their output.

Inputs:
LOCATION (string)

Outputs: None
Uses: None
Products: REPOSITORY(Repository::GenRep)

FileRepBuilder
This builds a repository from a set of flat files.

Inputs:
LOCATION (string)

Outputs: None
Uses: None
Products: REPOSITORY(Repository::GenRep)

DBRepBuilder
This builds a repository from a MySQL database.

Inputs:
SERVER (string)
DATABASE (string)
USERNAME (string)
PASSWORD (string)
TABLE (string)
DATERANGE_FIELD (string)
DATERANGE_BEGIN (string)
DATERANGE_END (string)

Outputs: None
Uses: None
Products: REPOSITORY(Repository::GenRep)
A.9  RepositoryBuilder (cont.)

AxiomRepBuilder
This builds a repository from the Axiom web repository. It uses the Corpus Builder technology.

Inputs:    TERM (string)
          DBASE (string)
          XMLOUTPUT (string)
          APPEND (bool)
          PROXY (bool)
          PARSEALL (bool)
          INSPEC (bool)
          COMPENDEX (bool)
          PAGEONE (bool)
          TITLE (bool)
          AUTHOR (bool)
          SUBJECT (bool)
          ABSTRACT (bool)
          DATE (int)
          YEARBEGIN (int)
          YEAREND (int)
          PHRASEMATCHING (int)

Outputs: None
Uses: None
Products:  REPOSITORY(Repository::GenRep)

DelphionRepBuilder
This builds a repository from the Delphion web repository. It uses the Corpus Builder technology.

Inputs:    TERM (string)
          DBASE (string)
          XMLOUTPUT (string)
          APPEND (bool)
          PROXY (bool)
          PARSEALL (bool)
          COLLECTION (string)

Outputs: None
Uses: None
Products:  REPOSITORY(Repository::GenRep)
A.10 SemanticModel

FileLsiModel
This loads an LSI semantic model from a file format. It was used for the PhD thesis experiments of April Kontostathis.

Inputs: FILENAME (string)
Outputs: None
Uses: None
Products: SEMANTIC_MAP(SemanticMap::)

TarModel
This creates a semantic model (clustering) from a supplied semantic network (graph). Tarjan's algorithm is used which is optimal showing a run time of O(N).

Inputs: ALPHA (double)
Outputs: None
Uses: IFS (ItemFeatureSet::)
      NET (SemanticNetwork::)
Products: SEMANTIC_MAP (SemanticMap::GenMap)
A.11 SemanticNetwork

CoNetImp
This creates an asymmetric graph of weighted arcs between the features of an IFS. A cooccurrence metric is used.

Inputs: THRESHOLD (double)
Outputs: MAX_WEIGHT(double)
          MIN_WEIGHT(double)
          MEAN_WEIGHT(double)
          STD_WEIGHT(double)
Uses: IFS (ItemFeatureSet::)
Products: None
A.12 TruthSet

HotTpcSet
This represents the truth set for the Emerging Trend Detection applications. The TffParser is used to read the truths from a tff file.

Inputs: None
Outputs: None
Uses: IFS (ItemFeatureSet::)
Products: None
A.13 TsGenerator

HotTopicTsGen
This represents the training set for the Emerging Trend Detection applications.

Inputs: None
Outputs: None
Uses:  IFS_CURRENT (ItemFeatureSet::)
        IFS_PREVIOUS (ItemFeatureSet::)
        IFS_PREVIOUS_2 (ItemFeatureSet::)
        IFS_PREVIOUS_ALL (ItemFeatureSet::)
        MB_CURRENT (MIDriver::ModelBuilder)
        MB_PREVIOUS (MIDriver::ModelBuilder)
        TRUTH_SET (TruthSet::)
Products: TRAINING_SET(TrainingSet::BasicTrainingSet)

ArffReader
This creates a training set from the Weka Library's arff format. Sparsified data sets are not yet supported.

Inputs:  FILENAME (string)
Outputs: None
Uses: None
Products: TRAINING_SET(TrainingSet::BasicTrainingSet)

MESentExtractTsGen
This generates a training set for the Maximum Entropy Sentence Extractor learning stage. This training set is utilized in the BinGIS learning process.

Inputs:  ABBRVS (string)
Outputs: None
Uses: REPOSITORY (Repository::)
Products: TRAINING_SET(TrainingSet::BasicTrainingSet)

MaxEntSentenceExtractor - FeatureExtractor
Maximum Entropy Sentence Extractor. A very efficient and useful method to remove sentences from text. This should be used with SimpleTagger.

Inputs: ALPHA_FILE (string)
        ABBREV_FILE (string)
Outputs: None
Uses: None
Products: None
Appendix B - Interface List

Attribute
An attribute represents a fundamental machine learning concept. This concept is that of a value selected from a set of possible though not necessarily finite and/or enumerable values. These values should be known and enforced by each derived class.

AttributeSelector
This represents a class that will be used to select a subset of the given attributes using some algorithm. Examples can be seen in both the Weka and MLC++ libraries of such algorithms.

Cluster
This represents the result of an unsupervised machine learning algorithm that generates clusters which are classes of features. An example of this is the strongly connected clusters that can be found in a semantic network using Tarjan's Algorithm.

Feature
A feature is a fundamental unit of TMI. It represents a basic textual entity and is sometimes referred to as a concept or phrase. Some examples of features are: noun phrases, N-grams, words, sentences, etc. A feature may represent a number of literal representations in a repository. That is, "the cat", "The cat", and "THE CAT" should all be the same feature.

FeatureExtractor
This represents a class that will be used to take a set of tagged data that includes a list of words (in order), a String of comma separated integer tags (to find out the meaning of these integers consult the Tagger) and offsets. It then uses this data to group the words in some fashion.

FeatureGenerator
The FeatureGenerator class is responsible for finding features in an input stream. It then stores these features in a GendFeatures object. This is often used in coordination with Tagger and FeatureExtractor. It is also used by Items for their FeatureGeneration process.

Item
An item is a coherent unit of text that will be analyzed. This includes but is not limited to documents. Some things which would not traditionally be thought of as documents such as web pages, chat conversations, etc. may be Item's. The key identifying feature of an Item is one coherent unit of text in a collection of such units.

ItemFeatureRelation
An ItemFeatureRelation is used by an ItemFeatureSet to store the relationship between an Item and a Feature. These relations can then later be queried to find information about the use of features in items, etc.

ItemFeatureSet
Specifically, an ItemFeatureSet is often generated from a Repository. This allows the statistics affecting the occurrence and co-occurrence of Features across the Items of the Repository to be better understood and leveraged for later processing steps.

MIAdapter
The change affected by an MIAdapter could be a simple filtering of the TrainingSet or could be something more complex such as breaking the data into training and test folds.

MIAlgorithm
This represents a class that will be used to attempt to discover classification patterns in training or truth sets that are provided. Examples of MIAlgorithms are decision tree induction algorithms, neural network optimizers, etc.

MIDriver
This coordinates the various steps of a Machine Learning process. In the componentized design some of these steps may occur external to the driver. The most important function of such a driver is to produce models and evaluate them. That is, an MIDriver should be able to take a train and test set and produce standard metrics.

MIEvaluation
MIEvaluations are often produced by MIDrivers though they need not be. Specifically they should give some objective and suitable measure of the performance of a machine learning algorithm. These metrics are all standardized to doubles. The usual examples are Precision, Recall, F-Measure, etc.

OptMethod
A suitable optimization method should have the following properties (or be as close as possible):

1. It should search for the minimum point in an N-dimensional space
2. It should work effectively in low dimensions
3. It should have control over the granularity of gradient estimates and steps
4. It should make as few f(x) calculations as possible
5. It should have a cutoff at which a suitably low f(x) value is obtained

Repository
An item is a coherent unit of text that will be analyzed. The Repository holds all the items in the data set, and they are retrieved using the getNext() function.

RepositoryBuilder
A RepositoryBuilder is responsible for generating, from some specified source, a Repository which is a set of documents. A RepositoryBuilder can make many repositories from the same source, given different parameters.

SemanticMap
This represents a partition or set of clusters for the features in an IFS. It is the natural result of an unsupervised clustering process on these features. This class is the result of a SemanticModel.

SemanticModel
A SemanticModel represents the model learning stage of a clustering process upon features. The result is a SemanticMap which is a list of the clusters generated.

SemanticNetwork
A SemanticNetwork represents a graph where the nodes are features. Arcs represent a measure of the relation of the features. Arcs may be asymmetric, positive, negative, etc. Very little is determined about them except by implementation. Examples of arc weighting metrics are TFIDF, LSI, etc.

Tagger
The Tagger is both responsible for splitting the item up into units and tagging these units. A common unit is the word and a common tag is the Part of Speech.

TrainingSet
A training set is defined as a set of instances. This class represents a training set by allowing access to these instances (TsInstance abstract class) through an iterator. As well, to enhance compatibility it allows the ability to write itself to an ARFF file or an MLC++ format file.

TruthSet
This class was designed to represent a truth set which is distinct from a training set. A truth set is defined as: An ordered set of pairs from data to data classification. Order MUST be maintained during all operations. As well the mapping may not be one-to-one. This allows this interface once implemented to represent:

1. Ground Truths
2. Gold Standards
3. Raw input to a machine learning library (must be processed through a training set generator first)

TsGenerator
The TrainingSetGenerator is responsible for creating uniform training sets thus each implementation will create a particular type of training set and for most cases a particular training set generator will be used for a particular instance of a machine learning driver (i.e. if you train on one type of training set the same type should be used for classification).

TsInstance
This class represents an abstract instance which consists of a set of attributes with an optional class. As well, one can set a comment about an instance which will explain where the instance is derived from or other give necessary background information */
Appendix C - Driver List

**ClusterTest**
This tests the unsupervised term clustering abilities of CoNetImp and TarModel. To do this it builds a Repository from a set of files and clusters the terms in these files. The clusters are written to a file.

**CollocationTester**
This demonstrates the Mean-Variance and Hypothesis-Testing methods of collocation generation. It does this by reading a repository from a set of XML files and extracting all the collocations of each type in the files. Other information about the collocation generation is displayed.

**ComponentDoc**
This is the component documentation tool. It uses an extra comments file (comments.txt) to get information about components. It combines this information with the query information from the components to generate a set of web pages and a plain text file documenting the components.

**DBTest**
This tests the database repository building capabilities. To test this it reads a Repository in from a database converts to a Repository with a new schema and then writes this Repository to an XML file.

**EmergingTrendDetection**
This demonstrates the Emerging Trend Detection as described in the TDMAPI paper. Please consult that paper for further information. This extends that by optimizing over the parameter ALPHA.

**EnglishBrillTagger**
This driver is a demonstration of the POS tagger.

**FlexFeatureExtraction**
This driver is a demonstration of the Noun Phraser.

**GenIfsTest**
This driver tests the Generic ItemFeatureSet from a Database Repository. It then writes this IFS to a file. This will soon write the IFS back to a database.
**HotTopicLsiTest**
The driver used for the PhD work of April Kontostathis. This driver has two parts. The first part reads Repositories in from a bad xml format. It then creates an IFS and uses this to write a feature list, feature x item matrix, and IFS to file. The second part reads the IFS file back in along with clusters generated from an LSI clustering system. It then allows EmergingTrendDetection experiments to be run on these clusters.

**MaxEntSentenceDriver**
This driver is used to train a Maximum Entropy model for sentence extraction. The model is stored for later use. Performance metrics are displayed.

**MaxEntSentExtractorDriver**
This driver is used to demonstrate the sentence feature extraction capabilities of the training Maximum Entropy model.

**OptMethodTester**
This driver demonstrates the integration of the bound-constrained quasi-newton optimization method in the infrastructure.

**OptML**
This demonstrates the use of an optimization method to optimize the f-measure performance of the WEKA decision-tree algorithm using a standard UCI training set.

**ShowFiles**
The name of this driver is a bit misleading. This driver is actually a combination of unit tests for various RepositoryBuilders.