MTI ML tool

Introduction

This package provides machine learning algorithms optimized for large text categorization tasks and means to combine several text categorization solutions. The advantages of this package compared to existing approaches are: its speed, able to work with a large number of categorization problems and the means to compare several text categorization tools.

This document presents the installation procedure and then goes on with the description of the different main components and examples of use. In addition to this document, the API of this package provides more detail of the available Java classes and their functionality.

Installation

This tool requires Java 1.6 or higher. If you do not have Java install in your system, you can obtain it from: http://java.com

This tool has been written entirely in Java and requires three JAR (Java Archive) files which can be downloaded from http://ii.nlm.nih.gov/MTI_ML/index.shtml:

- mti_production.jar: core package with the machine learning and meta-learning tools.
- util.jar: support tools including stop words processing and Java collection sorting.
- monq-1.1.1.jar: support for regular expression processing and client/server version of the annotator.

These jar files have to be added to the CLASSPATH variable. For instance, in bash shell:

export CLASSPATH=$CLASSPATH:mti.jar:util.jar:monq-1.1.1.jar

an alternative way of setting the CLASSPATH is with the –cp argument of the Java Virtual Machine (JVM).

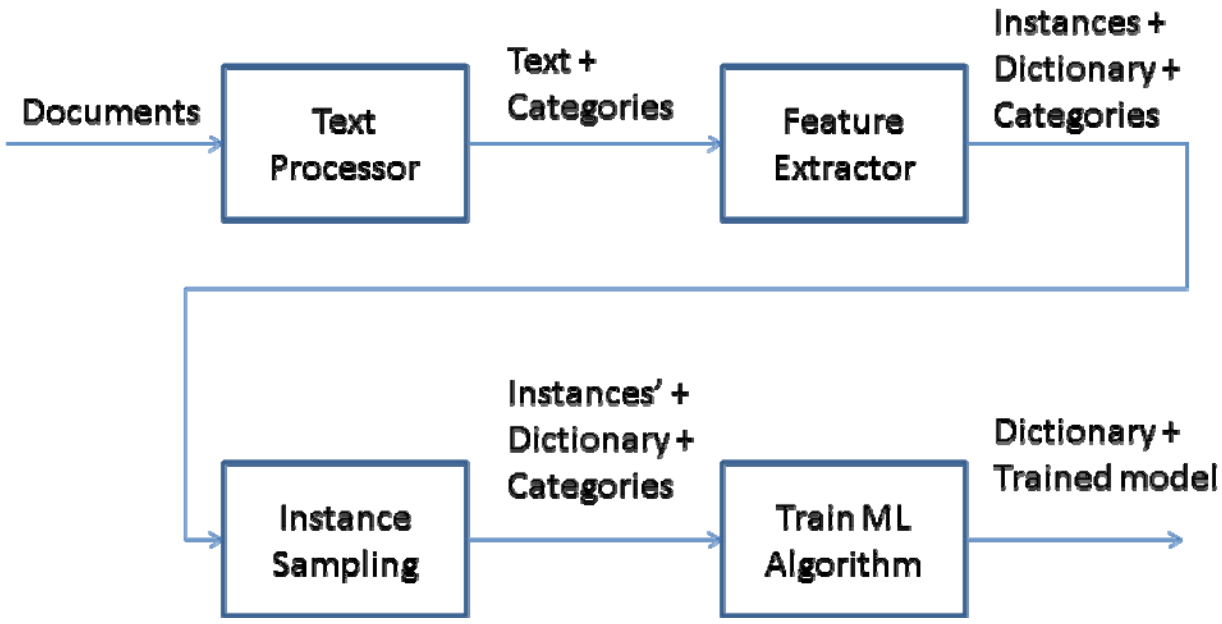
Note: trained machine learning algorithms might be contained in large objects. The JVM might exceed the maximum memory size allocated or the stack size per thread. Use the –Xmx and –Xms JVM options accordingly, currently a value of 10G is used for training and 1G is used for annotation. Consider as well the "-ss" option from the JVM to increase the maximum stack size, currently the value 6000k is used during training and annotation.

Text categorization

We define two scenarios: training and annotation. During training, a model is learned for each category (e.g. MeSH heading). During annotation, the models are used to annotate new documents (e.g. MEDLINE citations). Results from the annotation scenario might be used to evaluate the classifiers.

Training

The following graph shows the training pipeline. The input is documents that already contain the category they belong to. The documents are processed by the *Text Processor* to extract text and meta-data to be consider for training. In the case of MEDLINE records, this means to extract the title, abstract text, and MeSH heading (MH) annotations. The MHs are the categories used to index the citations. Then, the text is processed by the *Feature Extractor* to extract the features to be used during training. To speed up the matching of tokens, a dictionary stored in a Trie structure[1] is generated. Then a Sampler is used to select and sample the instances before training. An example of sampler is OverSampling, which resamples from the category with less examples till positives and negatives have the same number of examples.



Training parameters

[1] http://en.wikipedia.org/wiki/Trie

Global parameters are specified when calling the training program. If any category needs a different preprocessing of the documents, e.g. a different processing of the citations to consider the affiliation of the first author, it has to be specified in a different training scenario.

- Text processor: extract textual pieces from the documents, (XML MEDLINE processor, ASCII MEDLINE or WEKA ARFF processor) (TE_parameter). Check the javadoc for each TextProcessor class options
- Feature processor: specifies the processing of the text. Each processor class will have its own set of parameters. More than one parameter can be specified. For non-specified parameters the default value of the processor is considered. Check the javadoc for each FeatureProcessor class options. Examples of common feature processing to be considered (FP_parameters):
    - Unigram, bigram, n-gram
    - Lowercase tokens or leave them as they appear in text
    - Stem features (e.g. Porter Stemmer) or leave the features as they appear in text
    - Remove features which appear less than a defined number of times
- File with details about the MeSH headings learning specificities (TrainingConfigurationFile). The details are specified below.
- Output files
    - Dictionary file: used to store the dictionary once the collection is processed.
    - Map file which stores the trained models. In this map, the key is the MH and the value is the trained model for that MH.

Example of use:

```
zcat train.xml.gz | java -ss6000k -cp $CLASSPATH
gov.nih.nlm.nls.mti.trainer.OVATrainer
gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor ""
gov.nih.nlm.nls.mti.featuresextractors.BinaryFeatureExtractor "-l -n -c -f1"
configuration.txt trie_new.gz classifiers_new.gz
```

For each category, the configuration file TrainingConfigurationFile specifies the learning configuration parameters:

- Category: the category (e.g. MeSH heading) for which the model will be learned. If there is more than one category with the same configuration, they can appear in the same configuration line separated by semi-colon.

- Data set filters: how documents are processed before training. These filters include feature selection and sampling. The filters are specified by the name of the class and separated by pipe the pipe character. The filters can be placed in a pipeline, e.g. perform feature selection2 and then perform oversampling. Implementations of filters are available under the package gov.nih.nlm.nls.mti.datasetfilters and implement the DataSetFilter interface. In the case of sampling, a seed for the generation of random numbers must be specified if required by the sampling algorithm. As we find in the example for Cats and Dogs, it is possible to do training without applying any filter.

- Learning algorithm: during training this should be the training algorithm. Parameters will be specified for each of the algorithms as required or default values will be considered.

Sample TrainingConfigurationFile file

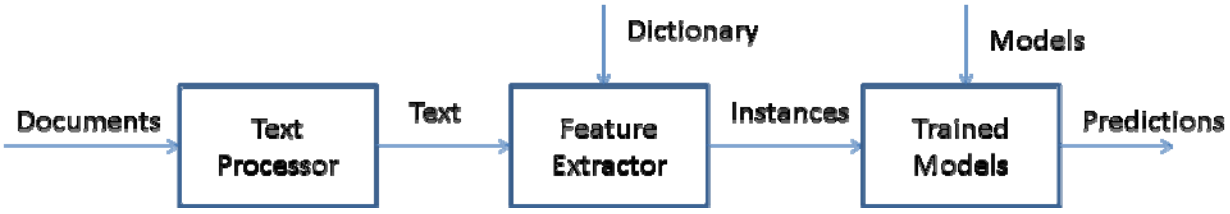
```
Cats;Dogs|gov.nih.nlm.nls.mti.ml.AdaBoostM1|base=”C45” basePar=”-M 5”

United States|
gov.nih.nlm.nls.mti.datasetfilters.samplers.Oversampling|seed=1|gov.nih.nlm.n
ls.mti.ml.AdaBoostM1|base=”C45” basePar=”-M 5”
```

Annotation

The following graph shows the pipeline of the process used to annotate new documents. The input is new documents and the output is predictions by the selected trained model. Again, the Text Processor extracts the textual part of the document, and then the Feature Extractor turns the text of the document into features to be used by the machine learning algorithm. The dictionary built during training is used by the feature extractor to ensure that the features are the same during training and annotation. Finally, the document instance is annotated by the selected trained model from the set of models selected above. The output prediction is a pipe separated list

[2] http://en.wikipedia.org/wiki/Feature_selection



Annotation Parameters

During annotation time, pointers to the generated files during training and instructions for text processing are required. The text processing instructions should be the same as the ones specified during training. The training configuration specified in TrainingConfigurationFile is not required at this point. There are two classes that can be used to perform annotation: OVAAnnotator and OVAAnnotatorConfidence. The latter returns the score (e.g. prediction) of the classifier in addition to the prediction.

The annotation parameters can be specified when calling the application as shown in the example below.

- Text processor: extract textual pieces from the documents, e.g. MEDLINE processor, WEKA ARFF processor.
- Feature processor: specifies the processing of the text (e.g. stemming, lower case ...). Each processor class will have its own set of parameters.
- Dictionary file: dictionary from the training data (DictionaryFile).
- Trained models: this is a map in which the key is the MH and the value is the trained model for that MH (MapFile).

Example of use:

```
zcat test.xml.gz |java -ss6000k -cp $CLASSPATH
gov.nih.nlm.nls.mti.annotator.OVAAnnotator
gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor ""
gov.nih.nlm.nls.mti.featuresextractors.BinaryFeatureExtractor "-l -n -c"
trie.gz classifiers.gz
```

Using the annotation tool as a server

The annotator relies on serialized trained algorithms. Loading these serialized models takes time and might use a large quantity of memory depending on the model. We propose using a client/server configuration. On the server side, the models are loaded in main memory and are ready for the annotation. On the client side, the citations to index are sent to the server that sends back the annotated set.

As we can see in the following example, compared to the previous annotation example, the only difference is the port in which the server is running (65000) that is the last parameter of the annotator class.

```
java -ss6000k -cp $CLASSPATH gov.nih.nlm.nls.mti.annotator.OVAAnnotator
gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor ""
gov.nih.nlm.nls.mti.featuresextractors.BinaryFeatureExtractor "-l -n -c"
trie.gz classifiers.gz 65000
```

As shown in the example below, the client requires a stream of citations to annotate from the standard input and attributes specifying the configuration of the server, this is, the server and port number. The current implementation supports up to a 100 clients. If there more than 100 simultaneous requests they will be queued until there is a free slot to process the request.

```
zcat test.xml.gz | java –cp $CLASSPATH
gov.nih.nlm.nls.mti.annotator.AnnotatorClient ii-server8 port
```

The output is the same as the one obtained with the regular annotator.

Building a training/annotation set

The training set should contain enough examples for the machine learning to build a model. Current research has shown that the more documents the better the models. On the other hand, the computational time and memory define an upper limit on the number of examples to consider. Current experiments have been done on a training set of 200k citations. For the test, it should be large enough to include examples of the MeSH headings to make the results significant. There are only a few MeSH headings with a large number of positive examples. We are currently using a test set of 100k citations. Experience has shown that the larger the set, the better the results. The current data set, with 200k for training and 100k for test, has provided good results for the Check Tags.

To make the task easier to deal with MeSH indexing, the training and annotation sets can be built easily from current databases. Two of the current formats are the ASCII and the XML version of MEDLINE. This makes it very easy to use the MEDLINE Baseline Repository[3] or EUtils[4] to compile a training/annotation

[3] http://mbr.nlm.nih.gov/

[4] http://www.ncbi.nlm.nih.gov/entrez/query/static/esoap_help.html

set. MEDLINE citations have already been indexed with MeSH and can be used as training and evaluation set. If the annotation set has already been indexed, this information will be ignored during annotation time.

The java classes to process the ASCII and the XML versions of MEDLINE are gov.nih.nlm.nls.mti.textprocessors.MEDLINEASCIITextProcessor and gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor respectively, these two classes extend gov.nih.nlm.nls.mti.textprocessors.TextProcessor. Further text processors can be added to cover a broader range of formats. To do so, the new classes have to extend the TextProcessor class. Example codes are available in the code of the MEDLINEASCIITextProcessor and MEDLINEXMLTextProcessor.

Evaluation

To evaluate the outcome of the trained machine learning algorithms, we require the output of the machine learning algorithms and a Gold Standard to validate the indexing. The output of the learning algorithm follows the pipe separated format presented above. The reference set follows a pipe separated format, each line of the form (PMID|Category). There is currently and implementation to extract the annotation a reference XML MEDLINE set:

```
cat reference_set | java –cp $CLASSPATH
gov.nih.nlm.nls.mti.evaluator.GetBenchmark
gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor "" > reference_set
```

Once we have the annotated file and the reference set, we can run the following tool that shows the algorithm performance for each MeSH heading in the reference set the performance of the algorithms under evaluation:

```
cat annotation | java –cp $CLASSPATH gov.nih.nlm.nls.mti.evaluator.Evaluator
reference_set_file_name
```

The lines in the output look like:

```
Male|34463|24664|7107|0.7763054357747632|0.7156660766619273|0.7447534498897849
```

The fields are:

1. Category
2. Number of positives in the dataset

3. Number of positives identified (true positives)
4. Number of times the algorithm under evaluation predicted the category but the prediction was not correct (false positives)
5. Precision
6. Recall
7. F-measure

Training, test and annotation example based on the *Humans* MeSH heading

This section details the steps from collecting the datasets, to the evaluation of the trained classifier for the *Humans* MeSH heading.

Collect the training and test sets

Data can be collected either from a MEDLINE distribution or online based on EUtils. Once the dataset is prepared, we can split it into 2/3 for training and 1/3 for test. We call the training set *train.xml.gz* and the test set *test.xml.gz*. In this example, both the training and test set are in XML MEDLINE format.

Training the algorithm

To train the machine learning algorithm, we need to prepare a configuration file which explicitly says which learning algorithm is going to be used and how the examples from the training set are going to be sampled during training. The example below shows the configuration line for the *Humans* category.

Humans|gov.nih.nlm.nls.mti.samplers.None||gov.nih.nlm.nls.mti.classifiers.ova.bf.AdaBoostM1|t=10

The first field is the name of the category. The second field specifies the sampling algorithm to be used to select citations from the training set (None, just consider all documents). In this case, all the citations are considered for training, so no parameter is specified in the following field. This is an example, but empirical evidence is required to select the best sampling to be used according to the problem. Finally, the learning algorithm is specified. In this case is AdaBoostM1 with 10 iterations (t=10) set in the

following field. In order to evaluate several algorithms on the same MeSH heading, we have to train these algorithms with a different configuration file and repeat the training and testing steps.

Once the configuration file is ready, we can train the AdaBoostM1 for the *Humans* category using the command below:

```
zcat train.xml.gz | java -ss6000k -cp $CLASSPATH
gov.nih.nlm.nls.mti.trainer.OVATrainer
gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor ""
gov.nih.nlm.nls.mti.featuresextractors.BinaryFeatureExtractor "-l -n -c -f1"
configuration.txt trie.gz classifiers.gz
```

The outcome of this process is two files: trie.gz contains the dictionary and classifiers.gz contains the trained AdaBoostM1 model to annotate citations with the *Humans* category.

Testing the trained model

Given the dictionary file and trained model, we can test the model on new citations. First, we need to annotate the new set, in this case the test set in test.xml.gz. We can do that with the following command. We can see that the process to extract and deal with the features from citations is the same as during the training step. We can use a different TextProcessor, e.g. to use ASCII MEDLINE documents, but we have to ensure that the extracted text would have been the same in both cases. Special care should be considered for Unicode characters. We use the trained model produced above. The predictions of the model are collected from the standard output into the Humans_output.txt file.

```
zcat test.xml.gz |java -ss6000k -cp $CLASSPATH
gov.nih.nlm.nls.mti.annotator.OVAAnnotator
gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor ""
gov.nih.nlm.nls.mti.featuresextractors.BinaryFeatureExtractor "-l -n -c"
trie.gz classifiers.gz > Humans_output.txt
```

As shown above, we have to extract the reference set from the evaluation set, in this case test.xml.gz. This produces a pipe separated file with the document identifier (PMID) and the MeSH headings. The output is placed in the file *benchmark.txt*.

```
zcat test.xml.gz | java –cp $CLASSPATH
gov.nih.nlm.nls.mti.evaluator.GetBenchmark > benchmark.txt
```

Finally, we evaluate the predictions in Humans_output.txt against the reference set annotations in benchmark.txt.

```
cat Humans_output.txt | java –cp $CLASSPATH
gov.nih.nlm.nls.mti.evaluator.Evaluator benchmark.txt
```

Annotating new citations

Given the dictionary file and trained model, we can annotate new citations with the following command. We can see that the process to extract and deal with the features from citations is the same as during the training step. We can use a different TextProcessor, e.g. to use ASCII MEDLINE documents, but we have to ensure that the extracted text would have been the same in both cases. Special care should be considered for Unicode characters. We use the trained model produced above. The predictions of the model are collected from the standard output into the Humans_output.txt file.

```
zcat test.xml.gz |java -ss6000k -cp $CLASSPATH
gov.nih.nlm.nls.mti.annotator.OVAAnnotator
gov.nih.nlm.nls.mti.textprocessors.MEDLINEXMLTextProcessor ""
gov.nih.nlm.nls.mti.featuresextractors.BinaryFeatureExtractor "-l -n -c"
trie.gz classifiers.gz > Humans_output.txt
```

Meta-learning

Meta-learning applies automatic learning to machine learning experiments. This means, the experimental data are indexing algorithm results, which are used to select the most appropriate algorithm. Indexing methods have different performance depending on the MeSH heading.

The input is the expected results in text categorization and the results of each method. The results file follows the same format as specified above for the output of machine learning algorithms. The benchmark file is just a pipe separated file with the format (Document Id|Category). This indicates that the document (Document Id) is annotated with Category.

The annotator result file is a pipe separated file with the format (Document Id|Category|Prediction). In the case of MeSH indexing, the Document Id is the PubMed Identifier, the Category is a MeSH heading and the Prediction is either 0 or 1 to indicate whether the document is recommended to be annotated with the MeSH heading (1) or not (0).

In the current implementation, the outcome of the meta-learner is a mapping table that for each MeSH heading indicates the method to be used. Even though in the future, a more complex output might include a more advance combination of methods.

In addition, in order to obtain more reliable results, the method selection by the meta-learning algorithm depends on the measurements obtained from more than one validation set. In order to do so, the training set should be split into several training validation folders, then obtain results for each folder of the data on the validation. The meta-learning tool incorporates this knowledge as follows:

cat methods_names | gov.nih.nlm.nls.mti.metalearning.MetaLearning benchmark categories number_of_folders

Meta-learner parameters:

method_names – enumeration of methods in a text file. One method per line, the name is used to identify the result file.

benchmark – pipe separated file

categories – name of the categories to be considered

number_of_folders – number of result files, one per cross validation folder. The result file for method1 for folder 5 is method1.5

Appendix

Results on a subset of the Check Tag MeSH headings

The following table shows the categorization results based on a subset of the Check Tag MeSH headings on 100k evaluation citations. AdaBoostM1 with C45 as the base learner have been used. 200k documents were used during the training.

MH	Positive	True Pos	False Pos	Precision	Recall	F-measure
Adolescent	8,156	3,232	3,676	0.4679	0.3963	0.4291
Adult	19,362	12,187	7,686	0.6132	0.6294	0.6212
Aged	13,389	8,198	5,828	0.5845	0.6123	0.5981
Aged, 80 and over	5,205	1,695	3,045	0.3576	0.3256	0.3409
Child, Preschool	3,302	1,681	1,682	0.4999	0.5091	0.5044
Female	35,501	26,021	6,269	0.8059	0.7330	0.7677
Humans	71,484	67,144	6,417	0.9128	0.9393	0.9258
Infant	2,569	1,278	1,508	0.4587	0.4975	0.4773
Male	34,463	24,820	6,689	0.7877	0.7202	0.7524
Middle Aged	18,709	12,493	6,359	0.6627	0.6678	0.6652
Swine	767	618	239	0.7211	0.8057	0.7611
Young Adult	8,527	2,914	4,472	0.3945	0.3417	0.3662