MTFeatures

-Design and implementation –

Catalina Hallett

1. User guide

1.1 Setting up

All resources required for running the application are included in the MTFeatures folder.
The folder contains the following resources:
/src – java source files
/lib – jar files, including the external jars required by MTFeatures and MTFeatures.jar
/doc – Javadoc documentation

In order to run the application or any of the tools included in the project, you have to include in the classpath MTFeatures.jar as well as the other jar files in the lib folder.

On running the application that performs the feature extraction (wlv.mt.FeatureExtractor) or the testing and training tools, they will create a folder structure at the location where they are run for. Therefore, you have to ensure that you run the application from a location where you have writing privileges and where you want your results stored.

1.2 The feature extractor

1.2.1 Overview

The application that performs feature extraction is wlv.mt.FeatureExtractor. It extracts Glassbox and/or Blackbox features from a pair of source-target input files and a set of additional resources specified as input parameters.

Whilst the command line parameters are instance-specific (i.e., they relate to the current set of input files), the FeatureExtractor also relies on a set of project-specific parameters, such as the location of resources. These are defined in a properties file in which resources are listed as pairs of *key=value* entries. By default, if no configuration file is specified in the input, the application will search for a config.properties file in the current working folder (i.e., the folder where the application is launched from). Failing to find it at that location, the application will look at the location of MTFeatures.jar. If no config.properties is found, the application will fail to initialize.

Another required resource is the XML feature configuration file, which have to be present at the location specified in the configuration file. For the current project, they are called

featureConfigBB.xml, featureConfigGB.xml and featureConfigAll.xml. Unless a feature is present in the respective feature configuration file it will not be picked up by the system.

1.2.2 Preparing the input

The basic resources required by the standard set of black-box features are:

- Source language:
  - Language model
  - Reference corpus (tokenised)
  - Ngram counts
- Target language:
  - Language model
  - Language model for the parts-of-speech
- Giza translation file(s)

These resources have to be referenced in the config.properties file.

NGram counts

In order to obtain an ngram counts file that is accepted by the application, one needs to process it first using the script wlv.mt.util.NGramSorter. This script processes an original ngram counts file and produces a similar file with the following modifications:

- Ngrams with frequencies lower than a given threshold are removed

Since the list of ngrams is very large, removing those with low frequency can significantly reduce both the amount of memory required by the application and the speed of access this list.

- The output file will start with a list of cut-off frequency values in various slices of the language model, such as:

```
1-gram  5     14    68    2289834
2-gram  4     6     16    1099927
3-gram  3     5     10    51465
```

This is interpreted as total number of 0-grams is 2289834, 1-grams with a frequency lower than 5 are in the first quartile of the language model, those with frequencies lower than 14 are in the second quartile, lower than 68 are in the third quartile. The remainder will be in the fourth quartile. This information is required by black-box features and although it can be computed on-the-fly this would require loading the whole list of ngrams in memory and put unnecessary strain on the application resources. Since the list of ngrams is a static resource, it makes much more sense to computhe the cut-off frequencies only once.

To use the NGramSorter:

```
java wlv.mt.util.NGramSorter <ngram_file> <sliceNo> <ngram_size> <minFreq> <output>
```

where:

- Ngram_file is the original file containing ngram counts
- sliceNo is an integer representing how many slices the corpus should be split into
- ngram_size is the size of ngrams

- minFreq is the lowest frequency value for ngrams in order to be included in the output
- output is the resulting ngrams file

Giza files

Giza files are lists of source word to target word translations. Entries in the Giza file are in the format

```
Source_word <whitespace> target_word <whitespace>
translation_probability
```

If a giza file contains indeces in a vocabulary rather than words, it has to be translated into the correct format before using it in the application. The package wlv.mt.util includes an application that can be used to perform this translation.

Usage:

```
wlv.mt.util.GizaMerger <giza_file> <source_vocabulary>
<target_vocabulary> <output> <probThresh>
```

The last parameter can be used to filter the Giza translation file if you want to restrict its size, by only selecting those translations that have a probability larger than probThresh. Set this parameter to 0 or ignore it if you want to keep all translations.

1.3 Running the Feature Extractor

Usage:

```
FeatureExtractor -input <source><target> -lang <source
lang><target lang> -feat [list of features] -mode [gb|bb|all] -gb
[list of GB resources] -rebuild -log
```

The valid arguments are:

-help : print project help information
-input <*source file*> <*target file*> ***(required)*** : the input source and target files
-lang <*source language*> <*target language*> : source and target language. If not present, the default values will be taken from the configuration file
-mode <*gb*|*bb*|*all*>
-feat : the list of features. The features can be specified as a coma-separated list (i.e., -feat [1,2,3,20]) or as an interval (–feat [1-20]). If this option is not present, all features corresponding to the selected mode will be included.
-gb [list of files]: input files required for computing the glassbox features
The arguments sent to the gb option depend on the MT system
For CMU: <nbest file> <onebest file> <onebest log file>
For IBM: <path to the word lattices folder>
For any system: the xml file containing the output of the MT system (see the section *Adding a new MT system* in the developer guide for details)

-rebuild : run all preprocessing tools
-log : enable logging
-config <config file>
-align <file> : file containing word alignment
-ner <list of files containing named entities>

Caveat! By default, the rebuild option is set to false, which means that a pre-processing tool will only run if it's output doesn't already exist. This greatly reduces the time required for pre-processing and is particularly useful when running experiments over the same set of files.
However, if a previous run of the FeatureExtractor has been prematurely interrupted or some of the pre-processing has failed for whatever reason, it is possible for the resulting pre-processing output to have been corrupt, which means that any future use of that output will produce wrong results or cause the FeatureExtractor to fail. There is no CRC check on the output and no built-in way of identifying whether things have gone wrong. If you suspect the output of the pre-processing is wrong, it is safer to run with the –rebuild option enabled. If the rebuild option is present, there is no choice of specifying which pre-processing tools will be run, therefore *all* pre-processing will be performed.

Example:

Running bb features:

```
java wlv.mt.aren.FeatureExtractor -input GALE09.sgm.Ar cmu-GALE09.txt -lang
arabic english -gb cmu/GALE-09/nw/nbest.txt cmu/GALE-09/nw/onebest.txt
cmu/GALE-09/nw/onebest.txt.log -mode gb
```

Running Glassbox features:

- **CMU system**

```
java wlv.mt.aren.FeatureExtractor -input GALE09.sgm.Ar cmu-GALE09.txt -lang
arabic english -gb cmu/GALE-09/nw/nbest.txt cmu/GALE-09/nw/onebest.txt
cmu/GALE-09/nw/onebest.txt.log -mode gb
```

- **IBM system**

```
java wlv.mt.aren.FeatureExtractor -input MT08.sgm.Ar ibm-MT08.txt -lang arabic
english -mode gb -gb ibm/MT08
```

- **CMU system from the XML output**

```
java wlv.mt.aren.FeatureExtractor -input GALE09.sgm.Ar cmu-GALE09.txt -lang
arabic english -gb cmu/GALE-09/GALE09.xml -mode gb
```

The result of FeatureExtractor is a file stored in the /output folder of the MTFeatures location. The file will be named [bb|gb|all]<source>_to_<target>.out

1.4 Training and testing

Constructing a training and testing dataset

The application wlv.mt.test.TestDataBuilder can be used to split a series of input files into testing and training files.

1) To create a dataset from scratch:

```
java wlv.mt.test.TestDataBuilder <test name> <percentage> <list
of files>
```

<test name> is an arbitrary name for a test - a folder with this name will be created at your current location and all files produced will be created in this folder
<percentage> a number representing the percentage of lines you want to use for testing (the closest integer will be selected)
<list of files> any number of files with the same number of lines. Typically one may supply a bb file, a gb file and a file with human scores, but you can also have, for example, multiple variations of the bb and gb files (with different number of features)

Result:
A folder named <test name> will be created containing:
- for each input file, a .train and a .test file
- an indeces.txt file containing the line indeces selected for testing and training
- a testSet.nfo file containing information about the dataset

2) To create a dataset from a list of indeces

If a test set has been previously created through a call to TestDataBuilder, the lines selected can be reused in further experiment. To create a dataset from a list of indeces, use:

```
java wlv.mt.test.TestDataBuilder <test name> <indeces file> <list
of files>
```

where <test name> and <list of files> have the same meaning as above, and <indeces file> is

Running the training and testing scripts

The application wlv.mt.test.TrainTest performs the operations necessary for running the training and testing scripts referenced in the config.properties file. The operations it performs, in sequence, are:

- Joins the training and testing files
- Converts the merged file to svm format
- Calls svm-scale on the converted file
- Splits the scaled data again
- Calls the training script with the scaled training data
- Calls the testing script with the scaled testing data

Usage:

```
java wlv.mt.test.TrainTest bbTrainFile gbTrainFile
humanScoreTrainFile bbTestFile gbTestFile humanScoreTestFile
```

The input files may have been produced by running TestDataBuilder, for example

2. Developer guide

2.1 Implementation overview

There are two principles that underpin the design choice: pre-processing must be separated from feature computation and one must be able to add features without resorting to re-compiling the whole project.

A typical application will contain a set of tools or resources (for pre-processing), with associated classes for processing the output of these tools. A Resource is usually a wrapper around an external process (such as, for example, a part-of-speech tagger or parser), but it can also be a brand new fully implemented pre-processing tool. The only requirement for a tool is to extend the abstract class **wlv.mt.tools.Resource**. The implementation of a tool/resource wrapper depends on the specific requirements of that particular tool and on the developer's preferences. Typically, it will take as input a file and a path to the external process it needs to run, as well as any additional parameters the external process requires, will call the external process, capture its output and write it to a file.
For examples of tool wrappers, please check the wlv.mt.tools package, which contains implementations for all tool wrappers required by the Arabic-to-English translation project.

The interpretation of any tool's output is delegated to a subclass of **wlv.mt.tools.ResourceProcessor** associated with that particular Resource. A ResourceProcessor typically reads in the output of a tool sentence by sentence and retrieves some information related to that sentence and stores it in a Sentence object. The processing of a sentence is done in the processNextSentence(Sentence sentence) function which all ResourceProcessor-derived classes must implement. The information it retrieves depends on the requirements of the application. For example, wlv.mt.tools.POSProcessor, which analyses the output of the TreeTagger, retrieves the number of nouns, verbs, pronouns and content words, since these are required by BB features in the current project, but it can be easily extended to retrieve, for example, adjectives, or full lists of nouns instead of counts.
Each ResourceProcessor must also register itself with the ResourceManager in order to signal the fact that it has successfully managed to initialise itself and it can pass information to be used by features. This registration should be done by calling `ResourceManager.registerResource(String resourceName)`. The resourceName is an arbitrary string, but if a feature requires this particular Resource for its computation, it needs to specify it as a requirement (see section *Adding new features*).

A **Sentence** is an intermediate object that is, on one, hand, used by ResourceProcessors to store information and, on the other hand, by Features to access this information. The i,mplementation of the Sentence class already contains access methods to some of the most commonly used sentence features, such as the text it spans, its tokens, its phrases and nbest translations (for gb features), its ngrams. For a full list of fields and methods, see the associated Javadoc. Any other sentence information is stored in a HashMap with keys of type String and values of generic type Object. A pre-processing tool can store any value in the HashMap by calling setValue(String key, Object value) on the currently processed Sentence object. This allows tools to store both simple values (integer, float) as well as more complex ones (for example, the ResourceProcessor

associated to the Stanford Parser resource associates full parses and lists of dependencies to a sentence via this method).
Features access these values through the method Sentence.getValue(String key), which will require a type cast to the appropriate type of the return value.
The Sentence class also contains access methods to n-best translations and translations phrases (only valid for source sentences). A list of ordered N-best translations can be retrieved by calling getTranslation(), which returns an object of type TreeSet<Translation>. A short-cut method for retrieving the best translation is getBest(), which returns a Translation object.
Similarly, phrases can be retrieved by calls to getPhrases(). For any other methods, see the Javadoc documentation for the Sentence class.

The main class of the project is wlv.mt.aren.FeatureExtractor, which extends wlv.mt.AbstractFeatureExtractor.
This class firstly assembles the input data from command line-parameters, and instantiates a ParameterManager which is in charge of accessing the application-specific parameters from the config.properties file. Secondly, the FeatureExtractor uses a FeatureLoader to instantiate those features required by the user and registers them with a FeatureManager. Thirdly, all pre-processing tools are run and the corresponding ResourceProcessors are instantiated.
The FeatureExtractor parses both the source and the target input files line by line and creates a Sentence object from each line. The each ResourceProcessor in turn is run over the pair of source and target Sentences and the FeatureManager is called to run the features over the Sentences.

2.2 Adding new features

In order to add a new feature, you have to implement a class that extends wlv.mt.features.impl.Feature. A Feature will typically have an index and a description which should be set in the constructor. The description is optional, whilst the index is used in selecting and ordering the features at runtime, therefore it should be set.

The only function a new Feature class has to implement is run(Sentence source, Sentence target). This will perform some computation over the source and/or target sentence and set the return value of the feature by calling setValue(float value).

If the computation of the feature value relies on some pre-processing tools or resources, then the constructor can add this resource in order to ensure that the feature will not run if the required resource is not present. This is done by a call to addResource(String resource_name), where resource_name has to match the resource name registered by the particular tool this feature depends on.

Example

The following Feature computes the percentage of nouns in the source. It is dependent on a part of speech tagger having run on the source, which registers the “sourcePosTarget” resource name. The feature accesses sentence properties, computes the value and sets it.

```
public class Feature1088 extends Feature {

  public Feature1088(){
  setIndex(1088);
  setDescription("percentage of nouns in the source");
  addResource("sourcePosTagger");
  }
  @Override
  public void run(Sentence source, Sentence target) {
        // get the number of tokens in the source
        int noWords = source.getNoTokens();
        //get the number of nouns which is set as a value on the
source Sentence
        float noNouns = (Integer)source.getValue("nouns");
        //compute and set the feature value
        setValue(noNouns/noWords);
  }
}
```

Features have to be added to the XML feature configuration files referenced in the application config file (by default, featureConfigBB.xml and featureConfigGB.xml).

An entry into the XML configuration file looks like this:

< feature index="1022" description="percentage of nouns in the source sentence" class="wlv.mt.features.impl.bb.Feature1022">

Whilst writing feature entries manually is preferred when just one feature, if adding multiple features one can use a tool included in MTFeatures.jar for automatically generating a feature configuration file or adding entries to an existing one. To do this, use:

```
java wlv.mt.features.util.FeatureSerializer <package name>
<config xml file name> <mode>
```

where

- <package name> is the name of the java package containing the features you want to serialize
- <config xml file name> is the full path of the xml file containing the feature configuration
- <mode> is either 0 or 1 depending on whether you want to add to or replace the configuration file

2.3 Adding a new MT system

Glassbox features rely on information retrieved from MT systems. The exact source of this information can vary widely from one MT system to another, therefore it is not possible to have an implementation that covers all possible source and formats of this information. For example, in the Arabic-to-English project we use the IBM system which produces word lattices, and the

CMU system, which produces a list of nbest translations in a proprietary format, a list of translation phrases and an execution log.

In order to allow for a consistent processing of any source of information we use an intermediate representation of MT data in XML format. Adding a new MT system means writing a wrapper around the information sources provided by that MT system, which stores the information in a pre-defined format.

The XML file will store the following information:

```
<text>

<Sentence attribute=value>

   <phrase start=start_value end=end_value text=text_value/>

   <translation rank=rank_no text=text_value attribute=value/>

</Sentence>

</text>
```

Each *Sentence* element can have any number of attributes with float values. There is no restriction imposed on the name or number of attributes. A Sentence also has a number of translation phrases which have to specify a start and end position in the text and the text they span. The *phrase* elements may be missing if the MT system doesn't provide them. If a list of nbest translations is available, they will be represented by *translation* elements, which can have any number of attributes with float values. Translations have a *rank* attribute which specifies their ordering in the nbest list. There is no other naming constraint imposed on the other attributes.

For an example of how to construct the XML output, see the XML wrappers written for the CMU and IBM systems: wlv.mt.xmlwrap.CMU_XMLWrapper and wlv.mt.xmlwrap.IBM_XMLWrapper.

The intermediate XML file is automatically processed by the ResourceProcessor-derived class wlv.mt.tools.MTOutputProcessor. Therefore an instance of MTOutputProcessor has to be added to the list of ResourceProcessors that are called from the main application.

When implementing glassbox features, one will need to implement one Feature class for each attribute one want to retrieve from this representation. The attributes will be stored in Sentence objects with the key named after the attribute name. For example, the CMU system will produce a sentence attribute called "discarded" (meaning the number of discarded search trees). From within a feature, this value can be accessed in the method run(Sentence source, Sentence target) by calling source.getValue("word penalty feature").

N-best translations are also stored as members of Sentence objects and their attributes can be accessed in a similar way. For example, if a feature wants to retrieve the value of the "word_penalty_feature" attribute of the best translation, the code will be:

```
public void run(Sentence source, Sentence target){

   Translation best = source.getBest();
```

```
//equivalent to source.getTranslations().first();

     Float value = best.getValue("word_penalty_feature");

setValue(value);

}
```

The MTOutputProcessor registers one resource for each sentence and translation attribute. It is therefore advisable to add these resources to the list of pre-requisites for the corresponding Feature class, to ensure that the Feature will not attempt to retrieve an attribute that is not present.

For example, the Feature class that retrieves the word_penalty_feature will call in its constructor `addResource("word_penalty_feature").`

2.4 Adapting for a new language

The first step in creating a new project for a different pair of languages is to create a new main class derived from wlv.mt.AbstractFeatureExtractor. This abstract class already provides implementations for some methods, which can be overridden to extend or adapt their behaviour to a new pair of languages.

For an example of a FeatureExtractor class, see wlv.mt.aren.FeatureExtractor for the Arabic-English project, or FeatureExtractorSimple in the English-Spanish project.

Secondly, the application will have to implement wrappers (classes extending wlv.mt.features.Resource) around pre-processing tools, if these are not already present, and the associated ResourceProcessors.

If a similar tool or processor already exists, the preferred method of extending its functionality to implement a class that extends an existing tool/processor.

For example, if a new application uses the TreeTagger but wants to count adjectives, it will need to implement a new processor that extends wlv.mt.tools.POSProcessor and override processNextSentence(Sentence sent) to count adjectives.

However, if the only change required is to have a different tag set for the TreeTagger, this can be done by setting the relevant tag sets programmatically.

For example:

```
PosTreeTagger myTagger = new PosTreeTagger();

myTagger.setNounTags(new String[]{"NN","NS"});

myTagger.setVerbTags(new String[]{"VB","VV","VBD"});
```

An alternative to implementing both a tool wrapper and a resource processor is to implement just a tool wrapper and have a post-processing step which converts its output to a format recognised by an already implemented processor. For example, in the Arabic-to-English project, we

implement ArabicPosTagger which calls the Arabic tagger and then converts its output to the TreeTagger format. This means we can use just one ResourceProcessor (wlv.mt.tools.POSProcessor) for both English and Arabic. This design choice was made because the output of the Arabic pos tagger was relatively similar to that of the TreeTagger.

The choice of implementation belongs to the developer and will have to take into account the coding effort involved in converting output formats as opposed to implementing ResourceProcessors fomr scratch.