MTFeatures

-Design and implementation –

Catalina Hallett
Modified by Kashif Shah

User guide

- 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

On running the application that performs the feature extraction (shef.mt.FeatureExtractorSimple) 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.

- The feature extractor

- Overview

The application that performs feature extraction is shef.mt.FeatureExtractorSimple. It extracts 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 FeatureExtractorSimple 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.

- 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)

All the resources are referred in the config_en-es.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 shef.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 shef.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 shef.mt.util includes an application that can be used to perform this translation.

Usage:

```
shef.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.

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

- N-best list

- One-best
- One-best.log

Running the Feature Extractor

Usage:

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

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*>
-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 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)
-log : enable logging
-config <*config file*>

Examples

Running black box features:

```
java -classpath build/classes:lib/commons-cli-1.2.jar:lib/stanford-postagger.jar:lib/BerkeleyParser-1.7.jar shef.mt.enes.FeatureExtractorSimple -lang english spanish -input input/source.en input/target.es -mode bb -config config/config_en-es.properties
```

Note: There are certain features based on standford-postagger and berkeleyParser. We need to include the jar files in lib directory and mention them in command.

Running glass box features:

```
Java -classpath build/classes:lib/commons-cli-1.2.jar:lib/stanford-postagger.jar shef.mt.enes. shef.mt.enes.FeatureExtractorSimple -lang english spanish -input input/source.en input/target.es -mode gb -config config/config_en-es.properties -gb gb_examples/nbest.txt gb_examples/onebest.txt gb_examples/onebest.txt.log
```

Running glass box features from XML output:

```
java -classpath build/classes:lib/commons-cli-1.2.jar:lib/BerkeleyParser-1.7.jar
shef.mt.enes.FeatureExtractorSimple -lang english spanish -input input/source.en
input/target.es -mode gb -config config/config_en-es.properties -gb
input/testGB/systems/cmu_source.en.tok.xml
```

The result of FeatureExtractorSimple is a file stored in the /output folder of the current working directory.

• Developer guide

• 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 **shef.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.

The interpretation of any tool's output is delegated to a subclass of **shef.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, shef.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 implementation 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 shef.mt.enes.FeatureExtractorSimple, which extends shef.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 FeatureExtractorSimple 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.

- **Adding new features**

In order to add a new feature, you have to implement a class that extends shef.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).
An entry into the XML configuration file looks like this:
< feature index="1022" description="percentage of nouns in the source sentence" class="shef.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 shef.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

- **Adapting for a new language**

One need to adapt configuration file to extract the features. i.e name of the source / target language should be changed to the new language wherever it occurs. e.g.
spanish.tokenizer should be replaced with french.tokenizer
spanish.ngramScript.path should be replaced with french.ngramScript.path
and so on.
Also one need to change the the run command accordingly according to desired language pair. i.e -lang english french

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 extend shef.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.

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.