REMEDI

Robust and Efficient Machine Translation in a Distributed Infrastructure

Bi-Annual Report: Month 6

Principal Investigator:

Dr. Christof Monz
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

Phone: +39 (0)20 525 8676
Fax: +39 (0)20 525 7940
E-mail: c.monz@uva.nl

Scientific Programmers:

Hamidreza Ghader and Praveen Dakwale
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

E-mail: h.ghader@uva.nl
E-mail: p.dakwale@uva.nl

Month	Deliverable	Progress
6	Implementation of language model data structure; implementation of multi-threading with large shared data structures; month 6 report	✓
12	Implementation of translation and reordering model data structures; implementation of decoder search and pruning strategies; month 12 report	✓
18	Implementation of search lattice data structure and feature passing; implementation of server front end; implementation of robustness preserving load balancing and restarting; complete software deliverable of decoder infrastructure; month 18 report	✓
24	Implementation of language model server in distributed environment; implementation of distributed translation model service; month 24 report	✓
36	Implementation of a suite of model building tools, covering translation, reordering, and language models; documentation model building tools; continued support of REMEDI translation infrastructure. Implementation of a suite of text preprocessing tools, covering parallel and monolingual data; documentation text preprocessing tools; continued support of REMEDI translation infrastructure; month 36 report.	✓

1 Introduction

In this report we describe the tools developed and provided for creating the required models for a phrase-based statistical machine translation (SMT) system. This set of tools can be downloaded from the MBT (Model Building Tool) Github repository[1] as follows:

```
git clone https://github.com/hamidreza-ghader/MBT.git
```

`git` should be already installed on most standard Unix distributions.

Modern SMT systems like the REMEDI system typically depend on three kinds of models: a language model, a translation model (phrase table), and a lexicalized reordering model (lexicalized distortion model). As the latter two are build from a parallel corpus (or bitext) they are referred to as bitext models when addressed together. We explain how to create bitext models in Section 2 and later on we move to language models in Section 3.

2 Bitext Models

Bitext models require a parallel corpus (bitext) to be built. Building them involves a number of steps:

1. Tokenization: A preprocessing action during which the text in a corpus is broken up into units of text called tokens.
2. Word Alignment: Here, word-to-word (but not necessarily one-to-one) links are learned for each parallel sentence pair in the the bitext.
3. Word Translation Models: Here, conditional word translation models of the form $p(w_f|w_e)$ and $p(w_e|w_f)$ are learned, where w_e and w_f are words.
4. Lexicalized Reordering Models: These are models of the form $p(Orientation|(p_f, p_e))$ where Orientation=monotonic,swap,discontinuous, where p_f and p_e are phrases.
5. Translation Models: These are models of the form $p(p_f|p_e)$ and $p(p_e|p_f)$, where p_f and p_e are phrases.

2.1 Tokenization

Tokenization is the process of breaking up the given text into units called tokens. The tokens may be words or numbers or punctuations. Tokenization does this task by locating word boundaries. In order to build robust models for machine translation, it is important to first tokenize the text. Our tool provides the utility to easily tokenize text in different languages.

[1] https://github.com/hamidreza-ghader/MBT

2.1.1 Dependencies

Since optimal tokenization depends on the orthography and vocabulary of languages, various language-dependent tools have been built for respective languages. Our tool uses the following two tools for four different languages.

- Stanford segmenter (Version 3.2)[2] : For tokenizing Arabic and Chinese text. Download and extract the segmenter from the given link. Ensure that the extracted directory contains a *.jar* file named *seg.jar*. As this is Java code, this tool does not need to be compiled, but does need to be installed. It is required to provide the path to this tool when tokenizing Arabic and Chinese.

- Moses decoder[3] : For tokenization of French and Italian, our tool relies on utilities provided by moses decoder scripts. When tokenizing these languages, it will be required to provide path of moses home directory.

For English and any other language, our tool uses a default tokenization scheme mostly based on punctuation.

Important Note that one should always use same tokenizer of the source side during model building and as well as during decoding. To do so with the REMEDI decoder, we provide a script that links the same tokenizer that is used in model building to the decoder. This is explained in more detail in Section 2.1.3.

2.1.2 Usage

Input texts can be tokenized by running perl script `tokenizer/tokenizer.pl`. The standard way of calling the tokenizer is as follows :

```
./[PATH-TO-MODEL-BUILDING-TOOL]/tokenizer/tokenizer.pl \
--input-file=input.txt \
--output-file=input.txt.tok \
--language=english
```

Detailed arguments (options) are explained as follows :

- `--input-file=string`

  Specifies the path to the input text file to be tokenized

- `--output-file=string`

  Specifies the path to the output tokenized file

[2] https://nlp.stanford.edu/software/stanford-segmenter-2013-06-20.zip
[3] http://www.statmt.org/moses/?n=Development.GetStarted

- `--language=string`

  Specifies the language of the input text. Where *string* can be either of the values : english, french, italian, arabic, chinese. if any other value is provided for this option, a default tokenizer is used. In case of other languages, this option can also be left unspecified.

- `--external-path=string`

  Specifies the path to external segmenters as explained in subsection 2.1.1. For any of the following languages, this option is mandatory.

  – For Arabic or Chinese input text, provide path to top level directory of Stanford segmenter.

    `--external-path=path_to_stanford_segmenter`

  – For French or Italian input text, provide path to top level directory of Moses decoder.

    `--external-path=path_to_Moses_decoder`

  For English or other languages, this option can be left unspecified

- `--num-parallel=int`

  Specifies degree of parallelization. If the text to be tokenized is very large, it maybe efficient to process the text in parallel threads. This option specifies the number of threads that should run in parallel. (default=1)

- `--num-batches=int`

  Specifies the number of batches the data should be split into for parallel processing. (default=`num-parallel`)

- `--keep-files`

  By default, the temporary files are deleted after successful tokenization. If you require the temporary files to be kept for debugging purposes, provide this option.

2.1.3 Linking Tokenizer to REMEDI Decoder

As mentioned above, it is necessary to use the same source side tokenizer during model building and decoding. To facilitate this, we provide a script to link the same tokenizer that was used during model building to the decoder. This script is called `tokenizerInjection.sh` and resides in the following path:

```
[PATH-TO-MODEL-BUILDING-TOOL-HOME]/tokenizer\
/tokenizerInjection.sh
```

Before use, the value of the variable `EXTERNAL_PATH` inside the file must change to your installation of Moses or Stanford segmenter depending on the source language (see Section 2.1.1).

Now follow the instructions in the preprocessor section of REMEDI documentation[4]. The only thing that needs to be changed is the path in the preprocessor executable in the corresponding configuration file of `bpbd-processor` and point to the `tokenizerInjection.sh` script. To be more specific, one has to change the `pre_call_templ` value in the configuration file of `bpbd-processor` to be as follows:

```
pre_call_templ=\
[PATH-TO-MODEL-BUILDING-TOOL-HOME]/tokenizer\
/tokenizerInjection.sh <WORK_DIR> <JOB_UID> <LANGUAGE>
```

2.2 Word Alignment

Having the source and the target side of our bitext tokenized, we are ready to compute word alignments (we are assuming that our bitext is a sentence-aligned bitext). This is done by calling the following script. The details of the call is covered in the following sections.

```
wordAlignment.pl
```

2.2.1 Installation

The word alignment tool works right after being cloning the repository. However, the following dependencies should be addressed beforehand.

Dependencies Our word alignment tool has dependencies to some executable files from the Moses[5] and MGIZA installation[6]. These files include:

1. mgiza
2. mkcls
3. snt2cooc
4. snt2coocrmp
5. merge_alignment.py

from the MGIZA installation and

6. symal

from the Moses installation.

In order to meet these dependencies, first of all Moses and the MGIZA tool should be installed following the Moses installation instructions[7] and the MGIZA installation instructions[8]. You may have already installed Moses following Section 2.1. After installations, the first 5 files should be copied from the MGIZA installation into the following path:

[4] https://github.com/ivan-zapreev/Distributed-Translation-Infrastructure#text-processor-bpbd-processor
[5] http://www.statmt.org/moses/?n=Moses.Releases
[6] https://github.com/moses-smt/mgiza/blob/master/mgizapp/INSTALL
[7] http://www.statmt.org/moses/?n=Development.GetStarted
[8] https://github.com/moses-smt/mgiza/blob/master/mgizapp/INSTALL

```
[PATH-TO-MODEL-BUILDING-TOOL-HOME]/dependencies/external_binaries
```

and the “symal” file should also be copied from the Moses installation (**important**: not from the MGIZA installation, although it also includes the same file) into the following:

```
[PATH-TO-MODEL-BUILDING-TOOL-HOME]/dependencies/moses/bin/
```

Note that all these files except `merge_alignment.py` are binary files which are generated during installation, including the compilation of the source code of Moses and the MGIZA tool.

Now the tool can be used by calling it using the command given in the example in the next section.

2.2.2 How to Use

In this section, we describe how to call the `wordAlignment.pl` script to compute the word alignments between the source and the target side of the bitext. The standard way of calling the script is as follows:

```
[PATH-TO-MODEL-BUILDING-TOOL-HOME]/wordAlignment.pl \
--dependencies=[PATH-TO-MODEL-BUILDING-TOOL-HOME]/dependencies \
--corpus=bitext --f=de --e=en --no-batches=2 \
--no-parallel=2 >& err.log
```

- `--corpus=string`

  specifies the shared prefix of the names of source and target files. This means that these two input files should have shared prefix in their names. For example `bitext.de` and `bitext.en`.

- `--f=string`

  specifies the suffix of the input source file. For example, `--f=de` if the source file name is like `[FILENAME].de`

- `--e=string`

  specifies the suffix of the input target file. For example, `--e=en` if the target file name is like `[FILENAME].en`

- `--dependencies=string`

  specifies the absolute path to the dependencies folder where other scripts are located.

  Note: The absolute path to dependencies directory must be provided. Providing a relative path may result in an error.

- `--no-batches=integer`

  specifies the number of batches that the data is split into. Alignments are computed independently for each batch. In order to obtain good-quality alignments, each batch should consist of at least 200K sentences.

- `--no-parallel=integer`

  specifies the maximum number of parallel runs. This number should always be a multiple of 2.

Running this command will create several directories including a ”models” directory in the directory from which you have run the script. If everything is went fine, there will be three files created under the following path:

`[PATH-TO-RUNNING_DIR]/models/model`

These files include:

- Aligned source corpus
- Aligned target corpus
- Alignment file

2.3 Translation and Reordering Models

With our model building tool perform steps 3 to 5 from Section 2, assuming that there is already a word-aligned parallel corpus. All these steps can be performed by the following script:

`build-models-from-wordAligned-bitext.pl`

This call is covered in more detail in the following sub-sections.

2.3.1 Installation

There is no need to install the model building tool itself. It works right after cloning from the repository. However, the following dependencies should be addressed beforehand.

Dependencies: This model building tool has dependencies to three binary files from Moses[9] translation system. These files include:

- consolidate
- extract
- score

In order to meet the dependencies, first install Moses translation tool following the installation instructions[10].

After installation, copy the aforementioned binary files from the Moses installation directory to the following path:

`[PATH-TO-MODEL-BUILDING-TOOL-HOME]/dependencies/moses/bin/`

Now the tool can be used by calling it using the command given in the example below.

[9] http://www.statmt.org/moses/?n=Moses.Releases
[10] http://www.statmt.org/moses/?n=Moses.Releases

2.3.2 How to Use

In this section, we describe how to call the `build-models-from-wordAligned-bitext.pl` script to create bitext models. The standard way of calling the script is as follows:

```
./build-models-from-wordAligned-bitext.pl \
--input-files-prefix=aligned --experiment-dir=./expdir \
--dependencies=[PATH-TO-MODEL-BUILDING-TOOL-HOME]/dependencies \
--f=chinese --e=english \
--a=grow-diag-final --build-distortion-model \
--use-dlr --moses-orientation --build-phrase-table >& err.log
```

- `--input-files-prefix=string`

  specifies the shared prefix of the names of source, target and alignment files. This means that these three input files should have shared prefix in their names. For examples `aligned.chinese`, `aligned.english` and `aligned.grow-diag-final`.

- `--experiment-dir=string`

  specifies the path in which the input files are placed and the final and intermediate output files of the model building process will be created.

- `--f=string`

  specifies the suffix of the input source file. For example, `--f=chinese` if the source file name is like `[FILENAME].chinese`

- `--e=string`

  specifies the suffix of the input target file. For example, `--e=english` if the target file name is like `[FILENAME].english`

- `--a=string`

  specifies the suffix of the input alignment file. For example, `--a=grow-diag-final` if the alignment file name is like `[FILENAME].grow-diag-final`

- `--build-distortion-model`

  flag to build lexicalized reordering model.

- `--build-phrase-table`

  flag to create phrase table (translation model).

- `--dependencies=string`

  specifies the absolute path to the dependencies folder where other scripts are located.

  Note: An absolute path to the dependencies directory must be provided. Providing a relative path may result in an error.

This will take some time, depending on the size of the bitext files. If they are small ($< 2,000$ lines, for debugging purposes), it will take a few minutes. If they are larger ($> 200,000$ lines) it can take several hours. After it has finished, you should see a number of new directories under the path to `--experiment-dir`. The most important one is `models/model` which contains the following files:

- `dm_fe_0.75.gz`
  the lexicalized reordering model
- `lex.e2f and lex.f2e`
  the word translation models
- `phrase-table.gz`
  the translation model

This completes the documentation of the tools used to generate the translation and reordering models. Next we proceed with describing the generation of the language model.

3 Language Models

In this section we describe the standard steps to build a language model from monolingual data provided in XML format. The following steps are carried out to extract, preprocess the text and to build a language model.

1. Plain text extraction from XML file. (optional if plain text is already available)
2. Tokenization
3. Language model building
4. Language model interpolation (optional)

In the following subsections we describe each of the steps, their utility, and usage.

3.1 Text extraction from XML documents

Standard datasets used for machine translation are made available in a standard XML format file with multiple documents along with their meta-information. Before, building a model, the plain text (one sentence per line) needs to be extracted from the XML file. This can be done by calling the perl script `xml-collection2plain.pl`. If training data is already available in plain text format, this step should be skipped. The standard way of calling the script is:

```
./[PATH-TO-MODEL-BUILDING-TOOL-HOME]/xml-extract\
/xml-collection2plain.pl \
--xml-input=input.xml \
--output-file=output.txt \
--exclude-dates=2004-07-01
```

We first describe the standard XML format that this script expects as an input. Note that the input file must follow the standard format in order to get the correct output. Following is a snippet showing the expected XML format.

```
<doc url="AFP_ENG_20040701.0001">
<date value="2004-07-01"/>
<language value="english"/>
<headline>
ISS crew successfully completes spacewalk at second attempt
</headline>
<body>
Astronaut Michael Fincke and Gennady Padalk returned to the ISS.
The pair opened the hatch at 1:19 a.m. Moscow time.
They worked so much so that the mission control had to rein them in.
</body>
</doc>
<doc url="AFP_ENG_20040702.0002">
<date value="2004-07-02"/>
<language value="english"/>
<headline>
Islanders plotting 'something special' against Wallabies
</headline>
<body>
The realisation of a rugby dream to pull together.
</body>
</doc>
```

The above XML snippet shows an input file with two documents. The meta-tags are described as follows:

1. <doc url="AFP_ENG_20040701.0001"> </doc>

   Each document should start with a *doc* tag and end with the corresponding close tag. It also requires a value for the attribute *url* which can either be the url of the web document from which the document is extracted or a filename in a text collection.

2. <date value="2004-07-01"/>

   The value of the attribute *value* can be the date on which the document was published or created. The required format is *YYYY-MM-DD*

3. <language value="english"/>

   The current version of the script only supports English documents. Hence, document should have the language tag with *value="english"*

4. <headline>Title_of_the_document </headline>

   Headline tag is used to provide the title of the web document from which it is extracted

5. `<body>Description of the article</body>`

   The main text of the document should be enclosed between the *body* tag. The text may have line breaks.

Next we describe the command line arguments for calling the script.

- `--xml-input=string`

  specifies path to the input XML file as described above

- `--output-file=string`

  Output text can be obtained in two formats, either as a single file or separate file for each document described in the XML file. If a single output file is required, provide `path/name` of the output file.

- `--output-dir=string`

  Provide the `path/name` of the directory if the output is required as a separate file for each document. The output files will be named with the value of the *url* field and stored in the given directory. Only one of the options `--output-file=string` or `--output-dir=string` should be provided. The script will complain if both options are provided.

- `--no-sentence-splitting`

  By default, the script splits the sentences and outputs one sentence per line. If sentence splitting is not required, this option is provided.

- `--exclude-dates=yyyy-mm-dd,yyyy-mm,yyyy,...`

  In some instances, we need to avoid extraction of text for some specific dates, for example, when our test sets are from a specific date range, we may want to avoid extracting training texts for those dates. To exclude text published/generated on specific dates, provide a comma separated list of strings representing excluded dates in *yyyy-mm-dd* format.

- `--constrain-dates=yyyy-mm-dd,yyyy-mm,yyyy,...`

  If documents published/created only on specific dates are required, provide a comma separated list of strings representing required dates in *yyyy-mm-dd* format.

3.2 Tokenization

See Section 2.1.

3.3 Building Large Language Models

With our model building tool, you can easily build large language models from plain text. This tool supports easy integration with the well-known SRILM language modeling toolkit.

3.3.1 Dependencies

This tool makes calls to executables from the SRILM language modelling toolkit, hence it is required to install SRILM before using this tool. Download the toolkit from SRILM[11] and follow the instruction for installation in INSTALL[12] file.

3.3.2 Usage

Large language models can be built by simply running the perl script `build-large-lm.pl`. Here is the standard way of calling the script:

```
./[PATH-TO-MODEL-BUILDING-TOOL]/build_language_model/\
build-large-lm.pl \
--text=input.txt \
--lm=output.lm \
--srilm-path=absolute-path-to-srilm-directory \
--order=5
```

Detailed arguments (options) are explained as follows :

- `--text=string`

  specifies the input text file. The input file should contain one sentences per line.

- `--lm=string`

  specifies the name of the output file containing the language model.

- `--srilm-path=string`

  specifies the absolute path to srilm top level installation directory.
  Note : The absolute path to SRILM should be provided as the script searches for configuration files inside the SRILM directory. Providing a relative path may result in an error.

- `--order=integer`

  specifies the n-gram order of the language model required. For example, `--order=5` specifies a 5-gram language model. (default=5)

- `--working-dir=string`

  This tools creates multiple temporary files while building the language model. This option specifies the path to a directory where these temporary files should be stored. The tool deletes the files after the language model has been built. (default=`working-dir`. If no value is provided, a temporary directory is built where the script is called)

[11] http://www.speech.sri.com/projects/srilm/download.html
[12] http://www.speech.sri.com/projects/srilm/docs/INSTALL

- `--keep-files`

  specifies temporary log files not to be deleted. By default, this tool deletes temporary log files. If they are needed for debugging purposes, provide this flag.

- `--smoothing=string`

  Smoothing or discounting techniques are used while building language model to account for rare n-grams. This tool provides two smoothing/discounting techniques.

  1. Kneser-Ney smoothing : specified as `--smoothing=kneser-ney` or `--smoothing=kndiscount`
  2. Witten-bell smoothing : specified as `--smoothing=wbdiscount` or `--smoothing=-wbdiscount`.

  (default=`kndiscount`)

- `--no-interpolation`

  specifies that no interpolation of the discounted n-gram probability estimates with lower-order estimates should be used. By default, this tool provides for such an interpolation (This sometimes yields better models with some smoothing methods).

- `--min-counts=integer-integer-integer-.....`

  Sets the minimal count of n-grams of order n that will be included in the LM. All n-grams with frequency less than that will effectively be discounted to 0. For example, `--min-counts=2-2-2-2-2`, specifies that for a LM of order 5, the minimum frequency of all n-grams of size 5 or lesser should be 2. Any n-gram with value less that 2 will be considered non-existent. (default=`1-1-1-2-2`)

- `--batch_size=integer`

  Specifies number of sentences per batch to be processed. This tool splits the input text in batches and finally combines output of each batch in a single language model. A greater batch size results in higher parallelization and hence builds the LM faster. However, it increases the memory requirement. A smaller batch size requires less memory, however, splits data into fewer batches and hence results in slower speed. (default=1000000). Reduce the number of batches if you have memory limitations.

- `--pre-processing=string`

  Here `string` is a comma separated list of one or more of the following :
  `{lc,numsub,dedupl,sent_tags}`
  This option specifies different pre-processing steps to be applied to the input text before building the language model. For example, `--pre-processing=lc` specifies that text should be converted to lowercase. This tool by default provides 4 pre-processing operations :

1. `--pre-processing=lc`
   all the letters in the text to be lowercased
2. `--pre-processing=numsub`
   all number should be nomalized. For example '27.6' and '15.3' will both be converted to '55.5'
3. `--pre-processing=dedupl`
   sort and remove duplicate entries from the files.
4. `--pre-processing=sent_tags`
   add tags `<s>` and `</s>` to start and end of each sentence, if they are not already present

Multiple preprocessing options can be provided for example by

```
--pre-processing=lc,numsub
```

The recommended value is `--preprocessing=lc,numsub,dedupl,sent_tags`

3.4 Language Model Interpolation

In Statistical Machine Translation, it may be beneficial to interpolate two or more different language models into one single Language model. For example, when building a Machine Translation system for specific domains such as healthcare/medical, we would like to build a language model only from the training data of medical domain. However, using additional data from other domains may result in better translation outputs. In such cases, one may want to give different weights to data or language models from different domains. For example, in this case higher weights for the medical domain LM and lower for general or other domain data. Our tool provides support for a such an interpolation.
Here we provide two sample calls to the interpolation script.
If you want to build an interpolated language model from multiple sets of plain texts, then the interpolation script should be called as follows by providing the `--input-corpora` option:

```
./[PATH-TO-MODEL-BUILDING-TOOL-HOME]/build_language_model/\
build-interpolated-lm.pl \
--input-corpora=1-1-1-2-2:kndiscount:input1.txt,\
1-1-1-1-1:wbdiscount:input2.txt \
--lm=interpolated.lm \
--ppl=dev-ref.txt \
--srilm-path=/absolute-path-to-srilm/\
--order=5
```

or if you want to interpolate pre-trained language models, then the interpolation script should be called as follows by providing the `--input-lms` option:

```
./[PATH-TO-MODEL-BUILDING-TOOL-HOME]/build_language_model/\
build-interpolated-lm.pl \
--input-lms=pretrained-1.lm,pretrained-2.lm,pretrained-3.lm \
--lm=interpolated.lm \
--ppl=dev-ref.txt \
--srilm-path=/absolute-path-to-srilm/\
--order=5
```

Further details on calling the interpolation script are explained below:

3.4.1 Usage:

- `--input-corpora=<tuple1>,<tuple2>,...`

  specifies a comma-separated list of input files format:
  `--input-corpora=<tuple1>,<tuple2>,...` where
  `tupleN = <min-counts>:<smoothing>:textfile`
  Here `textfile` is a plain text input file (one sentence per line) and `min-counts` and `smoothing` are the corresponding minimum frequency count and smoothing options to be used for the language model built from `textfile`.

  For example, in the example call above, we want to interpolate data from two different text files `input1.txt` and `input2.txt`. For the former we want to use a minimum frequency count of `1-1-1-2-2` and smoothing method *kndiscount* (Kneser-Ney) and for the later, we want to use a minimum frequency count of `1-1-1-1-1` and smoothing method `wbdiscount` (Witten-bell). If you do not require to build a language model form plain text, and only use pre trained language models, do not specify this option.

- `--input-lms=string1,string2,.......`

  specifies a comma-separated list of already built (pre-trained) LMs. If we want to simply interpolate pre-trained LMs, a comma separated list of the names/paths of the all pre-trained model is provided to this option. For example, in above sample call, we want to interpolate 3 language models. `pretrained-1.lm`, `pretrained-2.lm`, `pretrained-3.lm`.
  Do not specify this option, if you do not want to use any pre-trained LM. Only one of the two options, `--input-corpora` or `--input-lms` should be provided.
  Note : For interpolation, all the pre-trained language models must have the **same order**

- `--ppl=string`

  specifies name of the text file used for perplexity minimization.
  As described in the introduction of this section, during LM interpolation, we want to assign different weights to the data from different sources. These weights are calculated by minimizing perplexities of input language models on a single sample development text. Ideally, this text should be a text similar to the test domain. In the sample call above, we provide text file `dev-ref.txt` for perplexity minimization.

- `--num-parallel=integer`

  specifies the number of parallel builds (default=1). This depends on the size of the corpora. It's recommended to keep this value to less than 5.

- `--delete-builds`

  specifies if the individual language models built from plain text to be deleted or retained. (default=`true`)

The following options for the interpolated output language model are the same as described in Section 3.3.2.

- `--lm=string`

  Path to output language model file

- `--srilm-path=string`
- `--batch-size=integer`
- `--pre-processing=string`
- `--working-dir=string`
- `--order=integer`

  Note: The order of the interpolated output language model should be same as the orders of the pre-trained input language models

- `--min-counts=integer`
- `--no-interpolation`
- `--keep-files`

4 Software deliverables

This deliverable consists of this report as well as software deliverables. Both are distributed on MBT (Model Building Tool) Github repository[13]. If you clone the repository or download it as an archive you will see the following structure:

- `MBT/buildmodelsfromwordAlignedbitext.pl`

  The main bitext model building script.

- `MBT/build_language_model/...`

  This directory includes all language model building scripts.

- `MBT/dependencies/...`

  This directory is designated to include the scripts and the external binary files that our tools depend on. The external binaries should be compiled on your machine and copied to this directory based on the guidelines provided in this document.

- `MBT/readme.pdf`

  The PDF format readme file of the repository (this file).

- `MBT/tokenizer/...`

  This directory includes all the scripts required for tokenization.

- `MBT/wordAlignment.pl`

  The main script for computing word alignments on the bilingual corpus (bitext).

- `MBT/xmlextract/...`

  This directory includes scripts required for extracting plain text from standard XML formats used to distribute standard datasets.

The software component is accompanied by a short README file that explains how to run it.

5 Conclusions

In this deliverable we provide the software tools used to generate all models required by the REMEDI decoder, specifically this includes the generation of the translation model, the reordering model, and the language model.

The software is available as a Github repository and this PDF manual documents how to use the model building tools.

[13]https://github.com/hamidreza-ghader/MBT