Z-MERT: A Fully Configurable Open Source Tool for Minimum Error Rate Training of Machine Translation Systems

Omar F. Zaidan <ozaidan@cs.jhu.edu>
Johns Hopkins University – *Department of Computer Science*
and *The Center for Language and Speech Processing*

Documentation for **v1.10** (released February 9th, 2009)

I. General Information

1. Overview

Z-MERT is a software tool for minimum error rate training of machine translation systems (Och, 2003). It is:

- open source, extremely easy to run, and platform-independent.
- fully modular regarding the evaluation metric, easily supporting any new evaluation metric (with decomposable sufficient statistics).
- fully modular regarding the decoder, requiring *no* modification to function with any decoder.
- fully configurable, allowing the user to specify any subset of its 20-some parameters.
- highly optimized, and demonstrably time- and space- efficient.
- bug-free ☺

2. Description

State-of-the-art machine translation (MT) systems rely on several models to evaluate the "goodness" of a given candidate translation in the target language. Each model would correspond to a feature that is a function of a <*candidate translation*,*foreign sentence*> pair. Treated as a log-linear model, we need to assign a weight for each of the features. Och (2003) provides empirical evidence that setting those weights should take into account the evaluation metric by which the MT system will eventually be judged (i.e. maximize performance on a development set, as measured by that evaluation metric). The other insight of Och's work is that there exists an efficient algorithm to find such weights. This process is known as the **MERT** phase, for **M**inimum **E**rror **R**ate **T**raining.

The existence of a MERT module that can be integrated with minimal effort with an existing MT system would be beneficial for the research community. For maximum benefit, this tool should be easy to set up and use and should have a demonstrably efficient implementation. **Z-MERT** (Zaidan, 2009) is a tool developed with these goals in mind. Great care has been taken to ensure that Z-MERT can be used with any MT system without modification to the code, and without the need for an elaborate web of scripts, which is a situation that unfortunately exists in practice in current training pipelines.

3. Why You Should Use Z-MERT

- Z-MERT is completely **modular regarding the decoder**.
- Z-MERT **supports *any* evaluation metric** (with decomposable sufficient statistics), and a minimal amount of code is needed to implement any new evaluation metric.
- Z-MERT is **written in Java**, making it usable by virtually everybody, since the Java compiler is freely available (and for all common platforms).
- Z-MERT is **highly optimized**, making it both time- and space-efficient, and orders of magnitude faster than implementations written in interpreted languages, and apparently even faster than Moses' C++ MERT implementation.
- Z-MERT **requires no monitoring** from the user, and launches the decoder and processes its output automatically.
- Z-MERT is **fully configurable**, allowing the user to specify any subset of 20-some MERT parameters.
- Z-MERT is **fully documented**, complete with usage instructions as well as a tutorial.
- Z-MERT produces human-readable, (optionally) verbose, and **useful output**.

4. Download, Licensing, and Citation

Z-MERT's source code, instructions, documentation, and a tutorial are all included in the distribution, which can be obtained from Z-MERT's webpage:

http://cs.jhu.edu/~ozaidan/zmert/

Z-MERT is an open-source tool, licensed under the terms of the GNU Lesser General Public License (LGPL). Therefore, it is free for personal and scientific use by individuals and/or research groups. It may not be modified or redistributed, publicly or privately, unless the licensing terms are observed. If in doubt, contact the author for clarification and/or an explicit permission.

If you use Z-MERT in your work, please cite the following paper:

Omar F. Zaidan. 2009. Z-MERT: A Fully Configurable Open Source Tool for Minimum Error Rate Training of Machine Translation Systems. The Prague Bulletin of Mathematical Linguistics, No. 91:79–88.

And here's a BibTeX entry:

```
@article{Zaidan09zmert,
  author  = {Omar F. Zaidan},
  title   = {{Z-MERT}: A Fully Configurable Open Source Tool for
             Minimum Error Rate Training of Machine Translation Systems},
  journal = {The Prague Bulletin of Mathematical Linguistics},
  volume  = {91},
  pages   = {79--88},
  year    = {2009}
}
```

II. Usage

1. Requirements

Z-MERT is written in Java, and so you will need to have a Java interpreter on your computer. Sun's Java interpreter can be obtained for free (`http://java.sun.com/`) for all common platforms. The code was developed and tested using Java SE 6, and it is recommended that you avoid using earlier versions of Sun's Java interpreter or other free interpreters such as GIJ, though there is no reason to believe Z-MERT would behave differently with those interpreters.

A simple decoder is included in Z-MERT's distribution, and so you can go through the instructions/tutorial in this documentation without any additional components on your machine. The documentation only assumes that you have a Java interpreter, and nothing else.

2. A Note to Windows OS Users

Since Z-MERT is written in Java, its behavior should be platform-independent. Indeed, it has been extensively tested both under Windows and Linux, and you should not run into any problems even if you are a PC user. However, there is one thing that will require you to tweak the instructions just a bit: wherever you see the filename `SDecoder_cmd`, change that to `SDecoder_cmd.bat`. You should also rename the actual file (which is in `zmert_ex_orig\`), and change the filename in `zmert_ex_orig\ZMERT_cfg.txt` for the `-cmd` argument from "`./SDecoder_cmd`" to "`SDecoder_cmd.bat`".

3. Overview

Z-MERT is quite easy to use. There is no need to compile or install any files. You simply edit Z-MERT's configuration files to suit whichever experimental setup you want. We'll get to the details in a little bit, but basically Z-MERT expects a configuration file as its main argument, and some limit on how much memory it can use: (don't type this command just yet!)

```
  somedir> java -cp zmert.jar ZMERT -maxMem 500 ZMERT_config.txt
```

The `-maxMem` argument tells Z-MERT that it should not persist to use up memory while the decoder is running, during which time Z-MERT would be idle. The 500 tells Z-MERT that it can only use a maximum of 500 MB when it *is* active. (Java's `-Xmx` option would not do that for you. The documentation later explains exactly what `-maxMem` is (see III.4). It's pretty simple really, so don't worry too much about it for now.)

How does Z-MERT interact with the decoder? The configuration file tells Z-MERT how the decoder is launched. Z-MERT uses that information to launch the decoder as an external process to produce translations, and then uses the resulting output file in its parameter tuning. In so doing, **Z-MERT treats the decoder as a black box**, and knows nothing about its internals.

4. An Example

Let's go through an example of how Z-MERT would be launched and how it interacts with the decoder. First, create a new directory in `zmert_v1.10/` called `zmert_ex`, and copy the contents of `zmert_v1.10/zmert_ex_orig/` to it. Now take a look at the files in the newly created directory. Later in the documentation, we give more details on what everything means, what the proper formats are, etc, but for now, examine the files just to get a sense of what's about to happen when Z-MERT is launched. You should find:

- `ZMERT_cfg.txt`, where you can set any subset of Z-MERT's 20-some parameters.
- `params.txt`, where you tell Z-MERT about the weights you want to optimize.
- `ref.[0-3]`, which are the reference translations used to score candidate translations.
- `SDecoder_cfg.txt`, which appears to be a decoder's configuration file.
- `SDecoder_cmd`, which appears to be a script that would launch such a decoder.

Whenever you run Z-MERT, you **must** have those 5 things. No exceptions. Clearly, the contents of the files will depend on the specific experiment you are running and the specific decoder, but this group of files is the bare minimum that Z-MERT needs to run.

The names and contents of the last two files in the list above would lead you to believe that some sort of "simple" decoder exists. You would be correct, you smart lad you. This simple decoder is implemented by the aptly named `SimpleDecoder.java`, which we include so that we could design a tutorial that we're confident everybody can go through.

So what does this simple decoder do? It reads in a "database" of candidate translations that contains many candidates along with their feature values, ranks them according to the current feature weights, and produces an output file with *N*-best lists. To see it in action, type:

```
zmert_v1.10/zmert_ex> java SimpleDecoder SDecoder_cfg.txt src.txt nbest.out
```

The decoder expects 3 arguments: a configuration file, a file of source sentences, and a filename for its output. If you run the command above, the decoder will "translate" the Chinese sentences in `src.txt`, and produce for each one an *N*-best candidate list (with $N = 10$, the value for `top_n` in `SDecoder_cfg.txt`), as can be seen by examining its output file, `nbest.out`.

How are the candidates ranked? If you examine the decoder's configuration file, you will see it includes weights for 3 imaginary features whose names are: "`LM`", "`first model`", and "`second model`". It is based on these weights that the decoder ranks the candidates, where a candidate's score is the dot product of the weight vector and the candidate's feature vector. This feature vector is read from `cand_database.txt`, along with the candidate string itself.

Now take a look at the file `SDecoder_cmd`, and notice that it consists of exactly the command we just used to launch the decoder. (Make sure you set this file's permissions so that it is executable.) This file will be Z-MERT's way of launching the decoder in each iteration.

OK, let's run Z-MERT and see what kind of things it does. First, copy the file `zmert.jar` from `zmert_v1.10/lib/` to `zmert_v1.10/zmert_ex/`, and then type:

```
zmert_v1.10/zmert_ex> java -cp zmert.jar ZMERT -maxMem 100 ZMERT_cfg.txt
```

Z-MERT will complete 3 iterations in about just as many seconds, giving off some useful output about its optimization. You might want to rerun the command above and redirect the output to a file, capturing all that Z-MERT goodness. Now look at the contents of the folder, and notice that there are some new files produced by Z-MERT: `nbest.out.ZMERT.it[1-3]`, `SDecoder_cfg.txt.ZMERT.it[1-3]`, and `SDecoder_cfg.txt.ZMERT.final`.

Here's exactly what happened:

(0a) Z-MERT processes `params.txt` and figures out the names of the parameters it cares about, as well as the initial weight vector, which will be optimized.

(0b) Z-MERT renames `SDecoder_cfg.txt` to `SDecoder_cfg.txt.ZMERT.orig`.

(1) Z-MERT recreates `SDecoder_cfg.txt` with the current weight vector, using as a template the file `SDecoder_cfg.txt.ZMERT.orig`.

(2) Z-MERT launches the decoder by executing `SDecoder_cmd` as an external process, which in turn uses `SDecoder_cfg.txt` to produce (or overwrite) the file `nbest.out`, containing *N*-best candidate lists.

(3) Z-MERT makes copies of `SDecoder_cfg.txt` and `nbest.out`, with an extra `.ZMERT.it[1-3]` tacked on to the filenames of the copies.

(4) Z-MERT attempts to optimize the weight vector, based on `nbest.out` and the reference translations in `ref.[0-3]`. If there were no room for improvement, it proceeds to step (5), but otherwise it goes to (1).

(5) Z-MERT creates a `.ZMERT.final` decoder configuration file with the final weight vector, and renames the `.ZMERT.orig` file to its original name.

The Z-MERT run looped through steps (1)-(4) three times, with steps (3) and (5) creating those new files you saw after Z-MERT was done. There were a few other files Z-MERT created while it was running (with a `temp` in their names), but you probably didn't notice them since the Z-MERT run was so short, and those files were deleted at the end.

In order to do all that, Z-MERT needs to know the decoder-related filenames, where to read information about the parameters, where to find the references, etc. **You specify such information in `ZMERT_cfg.txt`, Z-MERT's configuration file**. It's fairly easy to figure out the meaning of the arguments in that file, but you should also read the documentation in III.3.E for the details. Note that the filenames for `-decOut` and `-dcfg` match the ones specified in `SDecoder_cmd`, and that the value for `-N` matches the one for `top_n` in `SDecoder_cfg.txt`.

5. Paths and Z-MERT

Before you ran Z-MERT, you were asked to copy the file `zmert.jar` to the `zmert_ex` directory, but you don't actually need to do that. Indeed, if you remove `zmert.jar` and then run:

```
zmert_v1.10/zmert_ex> java -cp ../lib/zmert.jar ZMERT -maxMem 100 ZMERT_cfg.txt
```

…you should get the same behavior as before. Let the **jar path** refer to the JAR file location, `zmert_v1.10/lib/`. In contrast, call `zmert_v1.10/zmert_ex/` the **working path**. We just saw those two need not be equal (i.e. you won't need copies of `zmert.jar` all over the place).

There is a *third* relevant path: the location you're in when launching Z-MERT. So far, this path, which we will call the **launch path**, has been equal to the working path. Do those two need to be equal? The answer is no. To illustrate this, let's say we want the launch path to be `zmert_v1.10/`. That is, we'd like to change directories to `zmert_v1.10/` and type:

```
zmert_v1.10> java -cp lib/zmert.jar ZMERT -maxMem 100 zmert_ex/ZMERT_cfg.txt
```

If you try the above command, Z-MERT will complain that it can't find the file `params.txt`, and then quits. You may have already figured out why that happens: Z-MERT is searching for the files in the launch path, because it wasn't told to look elsewhere.

To remedy this, edit Z-MERT's configuration file (which is in the working path). The first argument, `-dir`, is commented out. Uncomment this line, and replace `???` with `zmert_ex`. This is your way of telling Z-MERT what the working path is, *relative to the launch path*. Now, if you try the above command once more, you will get some good news and some bad news.

The good news is that Z-MERT is able to find the relevant files, as evidenced by its output. Indeed, you will see that Z-MERT has changed the decoder's configuration filename to `SDecoder_cfg.txt.ZMERT.orig`, and then recreated it based on the initial values in `params.txt`, indicating it has already done steps (0a) and (0b) above. To reverse this, delete `SDecoder_cfg.txt` and restore the original name of `SDecoder_cfg.txt.ZMERT.orig`.

The bad news is that the decoder wasn't launched successfully, and for a familiar reason: `SDecoder_cmd`, the decoder launcher, is being executed from the launch path. So, edit the decoder launcher by adding "`cd zmert_ex`" as the first line.

Now, run the above command again and, *voila*, it works. Nice!

In summary, the **jar path**, the **working path**, and the **launch path** need not be equal. The jar path is the prefix of Java's classpath (`-cp`). If the working path and the launch path are different, use `-dir` to indicate what the working path is, *relative to the launch path* (if they're equal, comment out the `-dir` argument). Finally, the decoder launcher must assume that it is in the launch path, and *must ensure the decoder output ends up in the working path*.

III. Technical Details

1. The MERT Algorithm

For more details on MERT and its line optimization, see Och (2003). Zaidan (2009) also provides a detailed overview, as well as Z-MERT's pseudocode. In essence, Z-MERT alternates between producing *N*-best lists and optimizing weights. Each iteration starts by decoding the data set using the current weights (step (2) on p.5), followed by repeated greedy single weight changes until no weight change can improve the score on the (cumulative) candidate list (step (4) on p.5). To avoid local optima, several random weight vectors are generated *in each iteration*, each optimized individually *in addition to* the weight vector generating the latest *N*-best lists. The value for `-ipi` (see 3.E below) indicates the number of those intermediate "initial" vectors For example, `-ipi`'s default value, `20`, means that, in addition to optimizing the weight vector that generated the latest *N*-best lists, each iteration should create 19 additional random weight vectors, and optimize each of them individually. Of the 20 intermediate "final" vectors, the one giving the best score survives, and is the one used to redecode in the next iteration.

2. Z-MERT's Modularity Regarding the Evaluation Metrics

You can specify which evaluation metric Z-MERT should optimize using the `-m` argument (see 3.E below). As of **v1.10**, Z-MERT includes implementations for `BLEU` (Papineni et al., 2002), `BLEU_SBP` (BLEU with strict brevity penalty, in Chiang et al., 2008), and `01LOSS`. The BLEU implementation closely follows IBM's `bleu-1.04.pl` evaluation script. In addition to the metric **name**, you should specify any **metric options**, which are variations on the metric. In the case of `BLEU` (and `BLEU_SBP`), there are two such options: maximum gram length, indicated by an integer, and the method for computing effective reference length, indicated by one of `closest` and `shortest`, corresponding to the IBM and NIST definitions, respectively. (In the near future, case insensitivity will be added as a third option.) `01LOSS` has no options.

The code structure of Z-MERT makes it rather easy to *implement a new evaluation metric*. To do so, extract the contents of `zmert.jar` file by typing `jar xf zmert.jar`, and modify a copy of `NewMetric.java.template`, using the following videos for guidance:

Part 1: http://www.youtube.com/watch?v=Yr56pD8bTUc&fmt=18
Part 2: http://www.youtube.com/watch?v=oCZdUxhE7ZU&fmt=18

Then, recompile the java files by typing `javac *.java`, and recreate the JAR file by typing `jar cvfM zmert.jar *.java* *.class`. (You can then discard the extracted files.) You can now run Z-MERT with your new metric by providing its name (and any metric options) to the `-m` argument. We also provide `EvalTool`, a tool that you can use to score translations using your new metric instead of writing an evaluation script from scratch. The tool can also be used to easily debug your new metric, since it can be instructed to produce detailed useful calculations on the sentence level.

The contents of `zmert_v1.10/New metric demo/` contain more details on this.

3. File Formats

A. The decoder configuration file is where the decoder finds the weights for the decoder's features, most likely surrounded by other things irrelevant to Z-MERT. The format is pretty liberal, in that the file can contain anything you desire. The only relevant lines are those where the weights are specified, each of which should be of the form `featureName featureWeight`. That is, ensure that the name starts the line, and notice the space that separates the name from the weight. There are no real restrictions on the feature name; it is even allowed to be composed of multiple words. Those lines can be anywhere in the file (and in any order), and need not be consecutive. Please note that the initial weight vector is specified by the parameter file (see 3.B below), and so **the weights initially in the decoder configuration file are ignored, since this file is used by Z-MERT only as a template**.

B. The parameter file is where you tell Z-MERT about the features used by the decoder, **including the features you do not wish to optimize**. Assuming there are *m* features, this file should consist of *m* + 1 lines: one line per feature, and one additional line for the normalization method. Unlike in the decoder's configuration file, **the order of the features in this file does matter**, and must match the order in which they appear in the decoder's output (see 3.C below). Each of the first *m* lines should be of the form:

```
featureName ||| defValue optStr minVal maxVal minRandVal maxRandVal
```

Where: `featureName` should match that used in the decoder configuration file (see 3.A above).
`defValue` is the "default" value for this feature. This will be used as the initial weight for this feature. If `-rand` is set to `1` (see 3.E below), this value is ignored.
`optStr` is either `Opt` or `Fix`, signifying whether this weight should be optimized or fixed. If this string is `Fix`, the next four values on this line are ignored.
`min/maxVal` is the minimum/maximum weight this feature can take. `minVal` can be the string `-Inf` (to indicate $-\infty$), and `maxVal` can be the string `+Inf` (to indicate $+\infty$).
`min/maxRandVal` specify the range from which random values are generated for this feature and must be numerical values (i.e. don't use `-Inf` or `+Inf`). Random values are generated when creating intermediate "initial" points (see III.1 above).

At the end of each MERT iteration, the values in the surviving weight vector are scaled according to some normalization method, which is indicated in the $(m + 1)^{st}$ line of this file. As of **v1.10**, five normalization methods are implemented, illustrated by the following five examples:

`normalization = none`	no normalization
`normalization = absval 1 LM`	scale weights so that the weight for `LM` equals `1`
`normalization = maxabsval 1`	scale weights so the maximum absolute value is `1`
`normalization = minabsval 1`	scale weights so the minimum absolute value is `1`
`normalization = LNorm 2 1`	scale weights so that the L-`2` norm equals `1`

In the above, `LM` can be replaced by any other feature name, and the other values `in red` can also be replaced by other numerical values.

C. The decoder output contains *N*-best candidate lists for the source sentences, along with their feature values. Assuming there are *m* features, a line in this file should be of the form:

```
i ||| candidate translation ||| featVal_1 featVal_2 ... featVal_m
```

Where: `i` is the index of the source sentence translated.
`candidate translation` are the words of the translation, separated by spaces.
`featVal_c` is the c^{th} feature value associated with this translation.

Other things are allowed after the m^{th} feature value (e.g. it is possible that the decoder also prints the candidate's score); they would simply be ignored by Z-MERT. But the lines must start with the sentence index, followed by the candidate and the feature values as above.

It is important that the candidates for a given sentence appear on consecutive lines, and it is important that the value for `i` never increases by more than 1. Each sentence would usually have a number of candidates equal to the value provided to `-N` (see 3.E below), but it is not a problem if there are fewer candidates for some sentence (Z-MERT will recognize this when it encounters an increase in `i`).

D. The reference file contains the references against which candidate translations are evaluated. The file should contain the references in plain format (i.e. no XML markup, etc), one reference per line. If there are multiple references per sentence, they should appear consecutively. For instance, if there are *r* references per sentence, the first *r* lines should be the references for the first sentence, the next *r* lines should be the references for the second sentence, and so on.

In the case of multiple references, it is also possible to provide multiple reference files, (where each file contains a single reference per sentence), as long as all the filenames are formed by concatenating a common filename prefix and a number. The numbering can start with 0 or 1. For instance, `ref[0-2]`, `ref[1-3]`, `ref.[0-2]`, and `ref.[1-3]` are all acceptable. When multiple files are provided, Z-MERT will create a unified reference file with the `.all` suffix in its name, which can be used in later experiments if one desires to disregard the individual files.

E. Z-MERT's configuration file is used to specify any subset of its 20-some parameters (23, to be exact), one argument per line. Comments are allowed using the `#` character. Each line follows the pattern `-arg value`. In general, a parameter expects a single value, which is either a number or a filename. The exceptions are `-m`, which might need several values to specify the metric, `-dir`, which expects a path, `-r`, which expects a filename but accepts a filename prefix, and `-seed`, which expects a number but accepts the string `time`. See below for details.

All the parameters have default values, and many of them probably need not be specified by the user. We will highlight some parameters **`in red`**, indicating the "main" subset of Z-MERT's parameters that the user would probably set explicitly.

Relevant files:

`-dir` **`dirPrefix`**: the working path, *relative to the launch path* (see II.5).
Default: null string (i.e. the working path and the launch path are equal).

`-r` **`refFile`**: target sentences (reference translations) of the MERT dataset.
Default: `reference.txt`.
If there are multiple references per sentence, you will likely have multiple files. In that case, provide **`-r`** with the *filename prefix*. For instance, if the reference files are `ref.0`, `ref.1`, and `ref.2`, make sure **`-rps`** gets `3`, and provide **`-r`** with `ref.` or `ref` (Z-MERT will automatically detect if it should add `x` or `.x` to the prefix). The numbering could also start with `1` instead of `0` (i.e. `ref.1`, `ref.2`, and `ref.3`), which Z-MERT will detect automatically as well. If you would like to consolidate all the references in a single file, see III.3.D above for the proper format and for an easy way of creating such a file.

`-rps` **`refsPerSen`**: number of reference translations per sentence.
Default: `1`.
This is not the same as the number of files containing references. That is, even if the multiple references are consolidated in a single file (see III.3.D), you should not provide the number of files (i.e. `1`), but the number of references per sentence (e.g. `3`).

`-p` **`paramsFile`**: file containing parameter names, initial values, and ranges.
Default: `params.txt`. See III.3.B for more details on the contents.

`-fin` **`finalLambda`**: filename where the final weight vector (and nothing else) is written.
Default: null string (i.e. no such file is created).
The final weight vector can also be found in the `.ZMERT.final` decoder configuration file, which Z-MERT will always create, regardless of **`-fin`**'s value.

MERT specs:

`-m` **`metricName metric options`**: name of evaluation metric and its options.
Default: `BLEU 4 closest`.
This is the only parameter that might need more than one value. The first value is a string, which is the **name** of the evaluation metric, and the other values are the metric's **options**. Z-MERT comes with 3 evaluation metrics already implemented: `BLEU`, `BLEU_SBP`, each of which need 2 options, and `01LOSS`, which needs zero options. See III.2 for more details on the evaluation metrics, and for guidance on implementing a *new* metric.

`-maxIt` **`maxMERTIts`**: maximum number of MERT iterations.
Default: `20`.

`-prevIt` **`prevMERTIts`**: maximum number of previous MERT iterations to construct candidate sets from.
Default: `20`.
The MERT algorithm uses the most recent decoder output to update the cumulative candidate set. Usually, this cumulative set consists of the output files from all the previous iterations. If you would like to limit Z-MERT to considering the output of only the most recent iterations, you can use **`-prevIt`** to do just that. If you would like all the output files to be considered, make sure that **`-prevIt`** and **`-maxIt`** have the same value.

`-minIt minMERTIts`: number of iterations before considering an early exit.

Default: `5`.

Z-MERT terminates if one of the following two criteria is met: the maximum number of iterations (**`-maxIt`**) is performed, or the weight vector is unchanged during an iteration. However, there are other stopping criteria (only one supported as of **v1.10**) that could indicate that it is safe for Z-MERT to terminate early. The user can use **`-minIt`** to ensure that Z-MERT runs for at least `minMERTIts` before considering an early exit, basically overriding the early exit mechanism during those initial iterations. Keep in mind, however, that *this parameter does not override the two "true" convergence criteria*.

`-stopIt stopMinIts`: number of consecutive iterations that must satisfy some early stopping criterion to cause an early exit.

Default: `3`.

Z-MERT keeps track of how many previous consecutive iterations (if any) satisfied some early exit criterion. That is, this count is incremented if an iteration satisfied some early exit criterion, and is reset to zero if an iteration satisfies no early exit criterion. The value for **`-stopIt`** indicates the value this count must reach to cause an early exit.

`-stopSig sigValue`: early exit criterion: no weight changes by more than `sigValue`

Default: `-1` (i.e. this criterion is never investigated).

`-save saveInter`: save intermediate decoder configuration files (1) or decoder outputs (2) or both (3) or neither (0).

Default: `3`.

This value determines what happens in step (3) of the description in II.4 (page 5). Regardless of what this value is, Z-MERT will always create the `.ZMERT.final` decoder configuration file.

`-ipi initsPerIt`: number of intermediate initial points per iteration.

Default: `20`.

This value corresponds to `numInter` of Alg. 1 in Zaidan (2009). See III.1 for details.

`-opi oncePerIt`: modify a parameter only once per iteration (1) or not (0).

Default: `0`.

If this is set to 1, Z-MERT will launch the decoder after every single weight change. Unless you have an incredibly (unnaturally?) fast decoder, this would mean a huge increase on runtime. Only set this parameter to 1 if you have a good reason to do so.

`-rand randInit`: choose initial point randomly (1) or from `paramsFile` (0).

Default: `0`.

If this is set to 1, the feature weight values in `paramsFile` are ignored, and the initial weight vector is generated randomly (see III.3.B). Either way, **the values in the decoder configuration file are ignored, since Z-MERT uses this file only as a template**.

`-seed seed`: seed used to initialize the random number generator.

Default: `time`.

This parameter expects a numerical value (a `long`), but also accepts the string `time`. If a number is provided, it is used as the seed. If `time` is provided, the seed will be the value returned by a Java `System.currentTimeMillis()` call. Either way, Z-MERT will print out the seed as part of its output, for replicability purposes.

Decoder specs:

`-cmd` **`commandFile`**: name of file containing commands to run the decoder.

Default: `./dec_cmd`.

Notice that the provided value might not simply be the filename, but might include the familiar "`./`" prefix. You as the user must be able to execute this file from your command line by typing exactly the value provided for this argument. For instance, under Linux/Unix, ensure that the file's permissions are set to be executable. Under Windows, ensure that the file has an executable extension (e.g. `.bat`). Also, you must ensure that the decoder launcher can be executed by Z-MERT from the launch path, and that its output file ends up in the working path (see II.5).

`-decOut` **`decoderOutFile`**: name of the output file produced by the decoder.

Default: `output.nbest`. See III.3.C for the proper format.

`-decExit` **`validExit`**: value returned by decoder to indicate successful completion.

Default: `0`.

Z-MERT launches the decoder as an external process, and checks the value returned by that external process. If the returned value does not equal `validExit`, Z-MERT will print an error message indicating that and exit. If you are uncertain what the 'success' return value is, it's likely to be `0`.

`-dcfg` **`decConfigFile`**: name of decoder configuration file.

Default: `dec_cfg.txt`.

Z-MERT will rename the provided decoder configuration file (by adding `.ZMERT.orig`), and then use it as a template to recreate the file (under the original filename) at the start of each iteration before executing the decoder launcher. See III.3.A for the proper format (which is actually fairly liberal). Keep in mind that **the values in the decoder configuration file are ignored, since Z-MERT uses this file only as a template**. At the end of the Z-MERT run, the original filename is restored (by removing `.ZMERT.orig`).

`-N` **`N`**: size of *N*-best list (per sentence) generated in each MERT iteration.

Default: `100`.

If the decoder configuration file contains a parameter that corresponds to this value (e.g. `top_n`), you must ensure that it is does not exceed the value provided for `-N`.

Output specs:

`-v` **`verbosity`**: Z-MERT verbosity level (0-2; higher value → more verbose).

Default: `1`.

The 1-level is suitable for most users. The 0-level renders Z-MERT completely silent, except for warnings and error messages. The 2-level gives more detailed output about the critical values extracted during line optimizations. (Z-MERT will also accept the values 3 and 4, which are values occasionally used during the development and debugging of Z-MERT, and they correspond to print statements suppressed in the public release.)

`-decV` **`decVerbosity`**: should decoder output be printed (1) or ignored (0)?

Default: `0`.

If set to 1, Z-MERT will include the decoder's output (both error and standard streams) in its output. Otherwise, the decoder's output will be discarded.

4. Z-MERT's Memory Usage and `-maxMem`

During a MERT iteration, there are several large data structures needed by the optimization process that require a decent amount of memory. However, chances are that if you have enough memory to be running a respectable decoder, then memory should not be a problem for you, since Z-MERT probably won't need any more memory than the decoder would.

The problem is that you may not have enough memory to support *both* the decoder and Z-MERT at the same time. It might seem that this should not be an issue, since Z-MERT basically sits idle while the decoder is producing translations. So why should we be concerned about the prospect of Z-MERT needing any memory while the decoder is running?

The answer is that a Java process does not return the memory allocated to it until the process actually terminates, even if no data structures are allocated any memory "internally" by that process. In other words, even though no memory is *needed* by the end of a Z-MERT iteration, the Java process does not return any memory already allocated to it back to the OS. That is why, when the decoder is launched, it would be competing for memory with Z-MERT, since Z-MERT would be hogging up quite a bit of memory that it refuses to return.

But fear not! To get around that, <u>Z-MERT performs each iteration as a separate Java process</u>. In other words, the Z-MERT driver launches one external process per MERT iteration, and so the end of that iteration is indeed the end of that individual external process. This means that the memory required for that iteration will be returned to the OS just before the decoder is launched by the Z-MERT driver.

Therefore, the only Java process that persists across iterations is the Z-MERT driver itself (the `ZMERT` class), which requires very little memory. This way, when the decoder is launched, it will not have to compete with other processes for memory. The `-maxMem` argument provided to Z-MERT (as in the commands of II.4 and II.5) indicates the maximum amount of memory (in MB) it is allowed during any single iteration.

IV. Miscellanea

1. FAQ

Q: Why did you develop Z-MERT?

A: Z-MERT is part of a larger effort at JHU to develop Joshua (Li et al., 2009), an open source package that includes the components of a complete MT training pipeline, including a MERT module. At first, it seemed reasonable to use some existing MERT implementation as a starting point for Joshua's MERT module and adapt it as we see fit. However, we found that existing implementations were not suitable for our needs, and did not meet our standards of flexibility and ease of use. So we wrote our own implementation, and Z-MERT was born.

Q: Does that mean I need to have Joshua to use Z-MERT?

A: Not at all. Z-MERT functions completely independently from the decoder (in fact, that is one of its features), and so it does not even know that Joshua exists.

Q: Why is it called **Z**-MERT?

A: Our research group had been using an implementation by David Chiang called C-MERT, so I interpolated. Also, **Z**, being the last letter of the alphabet, signifies that this is the last implementation of MERT the world will ever need! Oh, and everybody knows that the letter **Z** is pretty awesome.

Q: I have some questions about the MERT algorithm itself. Can you help me?

A: Ideally, referring you to Och's paper (Och, 2003) would be enough. Unfortunately, key ideas of MERT are not explained well in Och's paper. My Z-MERT paper (Zaidan, 2009) contains a section that explains MERT's optimization algorithm, so it could be pretty helpful.

Q: What else is in your paper?

A: It contains Z-MERT's pseudocode, contrasts my implementation to two existing implementations, and discusses some of Z-MERT's features. It also reports on a number of experiments that illustrate Z-MERT's runtime efficiency.

Q: I have a feeling that you'd like to thank some people. Am I right or am I right?

A: Right you are. For starters, this research was supported in part by the Defense Advanced Research Projects Agency's GALE program under Contract No. HR0011-06-2-0001. More importantly, I would like to thank some members of the Joshua development team at JHU who offered continuing and helpful discussion, feedback, and ideas: Zhifei Li, Lane Schwartz, Wren Thornton, and our team leader Chris Callison-Burch.

Q: I have an unanswered question. What do I do now?

A: Ask it! Put that series of tubes to good use: ozaidan@cs.jhu.edu.

Q: Wow, you sure have a lot of patience to answer all those questions.

A: Is that a question or a qompliment?

2. History

Version changes in the first decimal place (e.g. **v1.05** to **v1.10**) reflect significant changes, such as changes in functionality or use. Changes in the second decimal place (e.g. **v1.23** to **v1.24**) reflect minor changes in the documentation, instructions, output, etc.

v1.10 (2/9/09)
Full documentation
.jar packaging
Video demonstration (implementing a new metric in Z-MERT)

v1.00 (1/20/09)
Initial release!

3. References

Chiang et al., 2008	D. **Chiang**, S. DeNeefe, Y.S. **Chan**, and H.T. **Ng**. 2008. Decomposability of Translation Metrics for Improved Evaluation and Efficient Algorithms. In Proceedings of EMNLP, pages 610–619.
Li et al., 2009	Z. **Li**, C. **Callison-Burch**, C. **Dyer**, J. **Ganitkevitch**, S. **Khudanpur**, L. **Schwartz**, W. **Thornton**, J. **Weese**, and O. **Zaidan**. 2009. Joshua: Open Source Toolkit for Parsing-based Machine Translation. EACL 2009 Fourth Workshop on Statistical Machine Translation. Athens, Greece. (*Forthcoming*)
Och, 2003	F. **Och**. 2003. Minimum Error Rate Training in Statistical Machine Translation. In Proceedings of ACL, pages 160–167.
Papineni et al., 2002	K. **Papineni**, S. **Roukos**, T. **Ward**, and W-J. **Zhu**. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In Proceedings of ACL, pages 311–318.
Zaidan, 2009	O. **Zaidan**. 2009. Z-MERT: A Fully Configurable Open Source Tool for Minimum Error Rate Training of Machine Translation Systems. The Prague Bulletin of Mathematical Linguistics, No. 91:79–88. (PDF included with Z-MERT distribution.)