PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 91 JANUARY 2009 79–88

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

Omar F. Zaidan

Abstract

We introduce **Z-MERT**, a software tool for minimum error rate training of machine translation systems (Och, 2003). In addition to being an open source tool that is extremely easy to compile and run, Z-MERT is also agnostic regarding the evaluation metric, fully configurable, and *requires no modification to work with any decoder*. We describe Z-MERT and review its features, and report the results of a series of experiments that examine the tool's runtime. We establish that Z-MERT is extremely efficient, making it well-suited for time-sensitive pipelines. The experiments also provide an insight into the tool's runtime in terms of several variables (size of the development set, size of produced N-best lists, etc).

1. Introduction

Many state-of-the-art machine translation (MT) systems over the past few years (Och and Ney, 2002, Koehn, Och, and Marcu, 2003, Chiang, 2007, Koehn et al., 2007) rely on several models to evaluate the "goodness" of a given candidate translation in the target language. The MT system proceeds by searching for the highest-scoring candidate translation, as scored by the different model components, and returns that candidate as the hypothesis translation. Each of these models need not be a probabilistic model, and instead corresponds 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. This is achieved by choosing the weights so as to maximize the performance of the MT system 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 has come to be known as the MERT phase (for **M**inimum **E**rror **R**ate **T**raining) in training pipelines of MT systems. The existence of a MERT module that can be integrated

Please cite this article as: Omar F. Zaidan, 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, 2009, 79–88.

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. We describe here one such tool, **Z-MERT**, 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[1] web of scripts, which is a situation that unfortunately exists in practice in current training pipelines.

We first review log-linear models in MT systems and Och's efficient method (Section 2) before introducing Z-MERT and its usage (Section 3). We also report experimental results that demonstrate Z-MERT's efficiency (Section 4). Finally, we provide details on how to take full advantage of Z-MERT's unique features (Section 5). Readers already familiar with MERT should feel free to skip to the last paragraph of Section 2.

2. Log-linear Models in Machine Translation

Given a sentence to translate f in the source (aka 'foreign') language, a MT system attempts to produce a hypothesis translation $\hat{e}$ in the target (aka 'English') language that it believes is the best translation candidate. This is done by choosing the target sentence with the highest probability conditioned on the given source sentence. That is, the chosen translation is:

$$\hat{e} = \underset{e}{\operatorname{argmax}} \Pr(e \mid f) \tag{1}$$

One could model the posterior probability $\Pr(e \mid f)$ using a *log-linear model*. Such a model associates a sentence pair (e, f) with a feature vector $\Phi(e, f) = \{\phi_1(e, f), ..., \phi_M(e, f)\}$, and assigns a score

$$s_\Lambda(e, f) \stackrel{\text{def}}{=} \Lambda \cdot \Phi(e, f) = \sum_{m=1}^{M} \lambda_m \phi_m(e, f) \tag{2}$$

for that sentence pair, where $\Lambda = \{\lambda_1, ..., \lambda_M\}$ is the weight vector for the M features. Now, the posterior is defined as:

$$\Pr(e \mid f) \stackrel{\text{def}}{=} \frac{\exp(s_\Lambda(e, f))}{\sum_{e'} \exp(s_\Lambda(e', f))} \tag{3}$$

and therefore, the MT system selects the translation:

$$\hat{e} = \underset{e}{\operatorname{argmax}} \Pr(e \mid f) = \underset{e}{\operatorname{argmax}} \frac{\exp(s_\Lambda(e, f))}{\sum_{e'} \exp(s_\Lambda(e', f))} = \underset{e}{\operatorname{argmax}}\, s_\Lambda(e, f). \tag{4}$$

2.1. Parameter Estimation Using Och's Method

How should one set the weight vector Λ? Och (2003) argues it should be chosen so as to maximize the system's performance on some development dataset as measured by the evaluation metric of interest. The error surface in this approach is not smooth, which means that

[1] *Elaborate*, as in complicated, hard to navigate, and headache-inducing.

gradient-based optimization techniques cannot be used. A grid search by repeated line optimizations is not a good option either, since the function is quite expensive to evaluate at a given point $p \in \Re^M$, as this would require rescoring the candidate set[2] for each sentence to find the 1-best translations at p. Och suggests an alternative efficient approach for this optimization, which we review here.

Assume we are performing a line optimization along the d^{th} dimension. That is, we have a weight vector $\Lambda = \{\lambda_1, ..., \lambda_d, ..., \lambda_M\}$, and we would like to find a new weight vector for which the d^{th} dimension is optimal, keeping the other dimensions fixed.

Consider a foreign sentence f, and let the candidate set[2] for f be $\{e_1, ..., e_K\}$. Recall from (4) that the 1-best candidate at a given Λ is the one with maximum $s_\Lambda(e_k, f)$, which (2) defines as $\sum_{m=1}^{M} \lambda_m \phi_m(e_k, f)$. We can rewrite that sum as $\lambda_d \phi_d(e_k, f) + \sum_{m \neq d} \lambda_m \phi_m(e_k, f)$. The second term is constant with respect to λ_d, and so is $\phi_d(e_k, f)$. If we rename those two quantities $\texttt{offest}_\Lambda(e_k)$ and $\texttt{slope}(e_k)$:

$$s_\Lambda(e_k, f) = \texttt{slope}(e_k)\lambda_d + \texttt{offset}_\Lambda(e_k). \tag{5}$$

This is the equation for a line, and so when we vary λ_d, the score of a candidate varies linearly. That is, if we plot the score for a candidate translation vs. λ_d, that candidate will be represented by a line. If we plot the lines for all candidates (Figure 1), then the upper envelope of these lines indicates the best candidate at any value for λ_d. This is basically a visualization of the decision process of (4).

Observe now that the non-smoothness of the error function surface is not arbitrary, but is in fact *piece-wise linear* along the λ_d dimension.[3] The reason is that the error is calculated based on the 1-best candidate translations, and a small change in λ_d usually does not change the top candidate. There is, however, a set of critical values along the λ_d dimension, corresponding to the intersection points that form the abovementioned upper envelope. These are the only points at which the error changes (due to a change in the set of 1-best candidates).

If we can determine these intersection points for each sentence, and then merge them all into one set of intersection points, we will then have an overall set of critical values along the λ_d dimension, with each value corresponding to a 1-best change for a single foreign sentence.[4]

This means that if we have already calculated the error's sufficient statistics for a λ_d value just before some critical value, the sufficient statistics for a λ_d value just after that critical value can be calculated quite easily: simply adjust the original sufficient statistics as dictated by the candidate change associated with that intersection point.

This way, there is no need to rescore the candidates, and we traverse the λ_d dimension by considering only intersection points to find the optimum value. Finding those critical values amounts to finding intersection points of the lines representing the candidates, which is an easy

[2]We have not yet indicated how this candidate set is obtained, but will do so shortly.

[3]Or, in fact, along any linear combination of the M dimensions.

[4]In theory, a single critical value might correspond to a 1-best change for more than one foreign sentence. Though infrequent, this does happen in practice and is accounted for in Z-MERT.

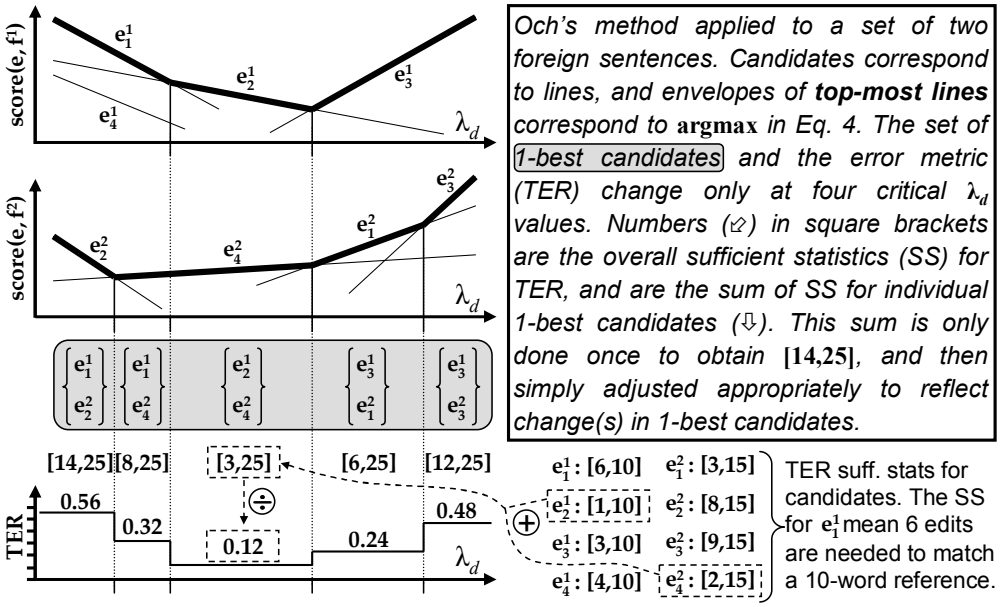

Figure 1. Och's method applied to a set of two foreign sentences.

process.[5]

One final piece of the puzzle is needed. We have been assuming that we have access to the set of candidate translations for each foreign sentence. How is this set actually obtained? One could try to enumerate *all* the possible candidates to cover the entire search space, but that may not be possible, and is likely quite costly anyway.

So we need an approximation to the candidate set. We could use the top, say, 300 candidates according the initial weight vector, but this set is quite concentrated, and is therefore not a good representative of the search space.[6] Instead, we alternate between optimizing the weight vector and producing the set of top candidates, each time merging the new candidate set with the existing candidates. The process is repeated until convergence, indicated by the candidate set not growing in size.

Och's method corresponds to line 17 in Algorithm 1, which is the pseudocode for Z-MERT's optimization process. Notice that Z-MERT repeatedly performs a line optimization (lines 14–27) along one of the M dimensions, greedily selecting the one that gives the most gain (lines 16–23). Notice also that each iteration optimizes several random "initial" points (line 9) in addition to the one surviving from the previous MERT iteration (line 8). This is used as an alternative to true multiple restarts.

[5]And it can be done efficiently: many of the lines need not be considered at all, such as the one for $\mathbf{e}_4^1$ in Figure 1.

[6]We are not referring to the small **number** of candidates here (which we already accept as a compromise to avoid

Algorithm 1 Z-MERT: Optimization of weight vector Λ to minimize error, using for line optimization (line 17) the efficient method of Och (2003).

Input: Initial weight vector $\Lambda^0 = \{\Lambda^0[1], ..., \Lambda^0[M]\}$; `numInter`, the number of initial points per iteration; and `N`, the size of the candidate list generated each iteration.
Return: Final weight vector $\Lambda^* = \{\Lambda^*[1], ..., \Lambda^*[M]\}$.

1. Initialize $\Lambda \leftarrow \Lambda^0$
2. Initialize `currError` $\leftarrow +\infty$
3. Intialize the cumulative candidate set for each sentence to the empty set.
4. **loop**
5. Using Λ, produce an `N`-best candidate list for each sentence, and merge it with the cumulative candidate set for that sentence.
6. **if** no candidate set grew **then** Return Λ // MERT convergence; we are done.
7.
8. Initialize $\Lambda_1 \leftarrow \Lambda$
9. **for** ($j = 2$ to `numInter`), initialize $\Lambda_j \leftarrow$ random weight vector
10.
11. Initialize $j_{best} \leftarrow 0$
12. **for** ($j = 1$ to `numInter`) **do**
13. Initialize $currError_j \leftarrow error(\Lambda_j)$ based on cumulative candidate sets
14. **repeat**
15. Initialize $m_{best} \leftarrow 0$
16. **for** ($m = 1$ to M) **do**
17. Set (λ,`err`) = value returned by efficient investigation of the m^{th} dimension and the error at that value (i.e. using Och's method)
18. **if** ($err < currError_j$) **then**
19. $m_{best} \leftarrow m$
20. $\lambda_{best} \leftarrow \lambda$
21. $currError_j \leftarrow err$
22. **end if**
23. **end for**
24. **if** ($m_{best} \neq 0$) **then**
25. Change $\Lambda_j[m_{best}]$ to λ_{best}
26. **end if**
27. **until** ($m_{best} == 0$)
28. **if** ($currError_j < currError$) **then**
29. $currError \leftarrow currError_j$
30. $j_{best} \leftarrow j$
31. $\Lambda \leftarrow \Lambda_j$
32. **end if**
33. **end for**
34. **if** ($j_{best} == 0$) **then** Return Λ // Could not improve any further; we are done.
35. **end loop**

3. Z-MERT

Z-MERT is part of a larger effort at Johns Hopkins University to develop **Joshua** (Li and Khudanpur, 2008) into an open source software package that includes a hierarchical phrase-based decoder (Chiang, 2007), as well as the components of a complete MT training pipeline. Two principles established by the developers were flexibility and ease of use, and were observed in the development of Z-MERT, Joshua's MERT module. Z-MERT also functions independently as a standalone application. That is, Z-MERT is also publicly available[7] separately from Joshua, which is still under development, since Z-MERT does not rely on any of Joshua's other components.

3.1. Existing MERT Implementations

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. (After all, there is no point in reinventing the wheel.) However, we found that existing implementations were not suitable for our needs, and did not meet our standards of flexibility and ease of use. We review here two such open source MERT implementations.

The first MERT implementation we examined is by Ashish Venugopal[8], which appears to have been first used by Venugopal and Vogel (2005). One immediate drawback of this implementation is that it is written in MATLAB®, which, like other interpreted languages, is quite slow.[9] Furthermore, MATLAB® is a proprietary product of The MathWorks, which limits use of Venugopal's implementation to those who have access to a licensed installation of MATLAB®.

Beyond that, the tool needs to be launched after every decoding step to perform the MERT optimization.[10] The user could certainly opt out of monitoring the MERT process to manually launch the decoder at the end of each MERT run (and vice versa) by writing a script capable of monitoring the two processes and launching them at appropriate times. But writing such a script seems like an unnecessary nuisance.

Z-MERT, on the other hand, is written in Java, making it orders of magnitude faster. This also makes it usable by practically everybody, since Java compilers are freely available for all common platforms, and users are likely to already have one installed and be familiar with it. Z-MERT also requires no monitoring from the user – all the user needs to do is specify the command that launches the decoder, and Z-MERT takes care of everything else.

---

enumerating all the candidates), but their limited **distribution**.

[7] Software and documentation available at: http://www.cs.jhu.edu/~ozaidan/zmert.html.

[8] Software and documentation available at: http://www.cs.cmu.edu/~ashishv/mer.html.

[9] It should be noted that the sufficient statistics for error are calculated outside MATLAB®, since it is a "costly process which is not well suited to MATLAB," according to the documentation. This implies an external script in some other language performs those calculations. It is not clear *which* language, since those scripts do not appear to be available for download on the software's page.

[10] It essentially performs a single iteration of the outermost loop of Algorithm 1.

Another implementation is the MERT module of Phramer[11], an open source MT system written by Marian Olteanu as an alternative to Pharaoh (Koehn, Och, and Marcu, 2003). The MERT module is written in Java, but a quick examination of the package's source code reveals that the `mert` folder contains a whopping 31 Java files! Granted, some of these are class definitions necessary for aspects like evaluation metrics, but the MERT "core" is still a large group of 15–20 files. Compare this to Z-MERT, which consists of only 2 Java files, one of which is a 20-line driver program. This makes compiling Z-MERT almost trivial and running it quite easy.

The biggest drawback with Olteanu's implementation, however, is that it is specifically geared towards Phramer and Pharaoh. It is not immediately clear how one would adapt it for use with other decoders.[12] Z-MERT, on the other hand, can be used immediately as a standalone application, without any modification to the code.

To summarize, Z-MERT is the first MERT implementation specifically meant for public release and for easy use with any decoder. Besides some unique features (Subsection 3.2), there are various advantages to using it: it is extremely easy to compile and run, it produces useful verbose output, it has the ability to resume stopped runs, it is highly optimized (Section 4), and its code is documented `javadoc`-style.

3.2. Z-MERT Usage and Features

Z-MERT is very easy to run. It expects a single parameter, a configuration file:

```
java ZMERT MERT_config.txt
```

The configuration file allows the user to specify any subset of MERT's 20 or so parameters, eight of which are shown in the sample file in Figure 2 (most parameters have default values and need not be specified). This high degree of configurability is Z-MERT's first feature. `-cmd` specifies a one-line file that contains the command that Z-MERT should use to produce an iteration's N-best list (line 5 in Algorithm 1). It is assumed that this command makes use of a decoder config file `-dcfg`, which Z-MERT updates just before producing the N-best list. Z-MERT knows how to update the file because it is informed of the parameter names in the file specified by `-p`.

The `-decOut` parameter indicates the file containing the newly created candidate translations. Z-MERT then proceeds by calculating the sufficient statistics for each candidate, as calculated against the reference translations in the `-r` file. Notice that Z-MERT is agnostic regarding the decoder, and treats it as a black box: Z-MERT prepares the configuration file, starts the decoder, and expects an output file once the decoder is done. The output file, which contains the candidate sentences and feature values, is expected to be in the familiar Moses-

[11] Software and FAQ for Phramer available at: http://www.utdallas.edu/~mgo031000/phramer.

[12] We are assuming here that it is indeed possible to adapt Phramer's MERT module to decoders other than Phramer and Pharaoh. This appears to be the case according to Lane Schwartz, who used it to tune parameters for a third decoder (personal communication). He mentions two Java classes that he needed to write to adapt Phramer's MERT module. We also imagine that, at a minimum, one would have to find and remove `import` and `package` statements referring to the Phramer package.

MERT_config.txt:

```
-cmd     dec_cmd.txt # decoder command file
-dcfg    dec_cfg.txt # decoder config file
-p       params.txt  # parameter file
-decOut  nbest.out   # decoder output file
-N       300         # size of N-best list
-r       refs.txt    # reference sentences
-ipi     20          # numInter (see Alg. 1)
-m       BLEU        # evaluation metric
```

params.txt:

```
lm                 |||  1.0  Fix  +0.5  +1.5
phrasemodel pt 0   |||  0.5  Opt  -1    +1
phrasemodel pt 1   |||  0.5  Opt  -1    +1
phrasemodel pt 2   |||  0.5  Opt  -1    +1
wordpenalty        ||| -2.5  Opt  -5    0
```

(Parameter name; Initial value; Optimizable?; Range for random values)

Figure 2. Sample MERT configuration file and parameter file.

and Joshua-like format.

The -m parameter illustrates another feature of Z-MERT: it is completely modular when it comes to the evaluation metric. Z-MERT can handle any evaluation metric, as long as its sufficient statistics are decomposable. This includes the most popular automatic evaluation metrics in the MT community, such as BLEU and TER . The public release already includes an implementation of BLEU (both IBM and NIST definitions), and implementing a new evaluation metric is quite easy (see Section 5).

4. Experiments

In Figure 3, we report the runtime of a number of experiments to demonstrate Z-MERT's efficiency by optimizing parameters for the Joshua decoder.[13] The MERT optimization was performed on a 2.0 GHz ThinkPad laptop. The development dataset is the text data of MT06, with 4 references per sentence. The evaluation metric being optimized is 4-BLEU (IBM definition). The left graph illustrates the effect of the development set size, for two different N-best sizes. The right graph illustrates the effect of `numInter` of Algorithm 1, for two different set sizes. The reported times are averaged over the first 4 iterations, and are for MERT only and do not include decoding times.

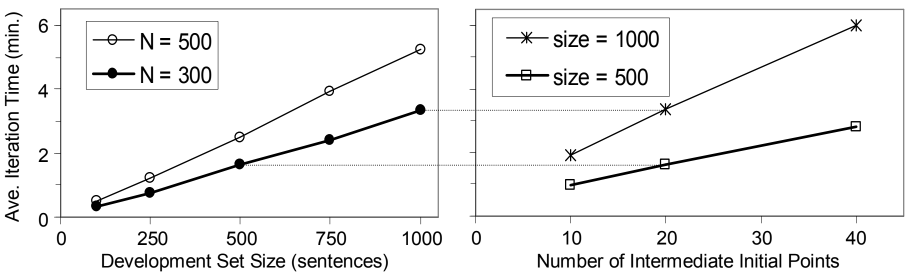

Figure 3. Average iteration time for MERT under different settings.

[13] We use the same parameter file as in Figure 2, except all parameters are optimizable. The language model is a 5-gram LM trained on the English side of the Gigaword corpus (about 130M words). The translation model has about 7.8 million rules.

5. Implementation Details

In addition to the MERT driver and the MERT core, Z-MERT has an `abstract EvaluationMetric` class. For an evaluation metric of interest, we need a corresponding class that `extends EvaluationMetric`. We need only two method definitions: one to calculate sufficient statistics for a candidate, and one that calculates the error given sufficient statistics. Consider the following metric. A candidate is compared against the reference translations. If the first word in the candidate matches the first in any reference, it gets +1, and similarly for the last word. So, a candidate translation can have a score between 0 and 2. Define the error to be the ratio of the sum of those scores divided by the maximum possible score. Here is the implementation:

```
public int[] suffStats(String cand_str, int i) {
  // Calculate the sufficient statistics for cand_str, compared
  // against the references for the ith source sentence.

  int[] retA = new int[suffStatsCount]; // array of SS to be returned
  int firstWordMatches = 0, lastWordMatches = 0;

  for (int r = 0; r < refsPerSen; ++r) {
    if (firstWord(cand_str).equals(firstWord(refSentences[i][r])))
      firstWordMatches = 1;
    if (lastWord(cand_str).equals(lastWord(refSentences[i][r])))
      lastWordMatches = 1;
  }

  retA[0] = firstWordMatches + lastWordMatches;
  retA[1] = 2;

  return retA;
}
public double score(int[] stats) {
  return stats[0]/(double)stats[1];
}
```

For clarity, we omit the trivial definitions for `firstWord(.)` and `lastWord(.)`. Data members such as `refSentences` and `refsPerSen` are already set by the parent class `EvaluationMetric`, which also defines appropriate methods to sum the sufficient statistics. And so, the complexity of the new code is a function of the complexity of the metric itself only; the user need not worry about any kind of bookkeeping, etc.

6. Conclusion and Acknowledgments

We presented Z-MERT, a flexible, fully configurable, easy to use, efficient MERT module for tuning the parameters of MT systems. Z-MERT is agnostic when it comes to the particular system, the parameters optimized, and the evaluation metric. Compiling and running Z-MERT is very easy, and no modification to the source code is needed. The user can easily perform MERT with a new evaluation metric by overriding only a small part of a generic `EvaluationMetric` class.

The author would like to thank the members of the Joshua development team at JHU. This research was supported in part by the Defense Advanced Research Projects Agency's GALE program under Contract No. HR0011-06-2-0001.

Bibliography

Chiang, David. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of ACL, Demo and Poster Sessions*, pages 177–180.

Koehn, Philipp, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proceedings of HLT-NAACL*, pages 127–133.

Li, Zhifei and Sanjeev Khudanpur. 2008. A scalable decoder for parsing-based machine translation with equivalent language model state maintenance. In *Proceedings of ACL, Second Workshop on Syntax and Structure in Statistical Translation*, pages 10–18.

Och, Franz Josef. 2003. Minimum error rate training in statistical machine translation. In *Proceedings of ACL*, pages 160–167.

Och, Franz Josef and Hermann Ney. 2002. Discriminative training and maximum entropy models for statistical machine translation. In *Proceedings of ACL*, pages 295–302.

Venugopal, Ashish and Stephan Vogel. 2005. Considerations in maximum mutual information and minimum classification error training for statistical machine translation. In *Proceedings of the European Association for Machine Translation (EAMT)*.