Joshua 4.0: Packing, PRO, and Paraphrases

Juri Ganitkevitch[1], Yuan Cao[1], Jonathan Weese[1], Matt Post[2], and Chris Callison-Burch[1]
[1]Center for Language and Speech Processing
[2]Human Language Technology Center of Excellence
Johns Hopkins University

Abstract

We present Joshua 4.0, the newest version of our open-source decoder for parsing-based statistical machine translation. The main contributions in this release are the introduction of a compact grammar representation based on packed tries, and the integration of our implementation of pairwise ranking optimization, J-PRO. We further present the extension of the Thrax SCFG grammar extractor to pivot-based extraction of syntactically informed sentential paraphrases.

1 Introduction

Joshua is an open-source[1] toolkit for parsing-based statistical machine translation of human languages. The original version of Joshua (Li et al., 2009) was a reimplementation of the Python-based Hiero machine translation system (Chiang, 2007). It was later extended to support grammars with rich syntactic labels (Li et al., 2010a). More recent efforts introduced the Thrax module, an extensible Hadoop-based extraction toolkit for synchronous context-free grammars (Weese et al., 2011).

In this paper we describe a set of recent extensions to the Joshua system. We present a new compact grammar representation format that leverages sparse features, quantization, and data redundancies to store grammars in a dense binary format. This allows for both near-instantaneous start-up times and decoding with extremely large grammars. In Section 2 we outline our packed grammar format and present experimental results regarding its impact on decoding speed, memory use and translation quality.

Additionally, we present Joshua's implementation of the pairwise ranking optimization (Hopkins and May, 2011) approach to translation model tuning. J-PRO, like Z-MERT, makes it easy to implement new metrics and comes with both a built-in perceptron classifier and out-of-the-box support for widely used binary classifiers such as MegaM and MaxEnt (Daumé III and Marcu, 2006; Manning and Klein, 2003). We describe our implementation in Section 3, presenting experimental results on performance, classifier convergence, and tuning speed.

Finally, we introduce the inclusion of bilingual pivoting-based paraphrase extraction into Thrax, Joshua's grammar extractor. Thrax's paraphrase extraction mode is simple to use, and yields state-of-the-art syntactically informed sentential paraphrases (Ganitkevitch et al., 2011). The full feature set of Thrax (Weese et al., 2011) is supported for paraphrase grammars. An easily configured feature-level pruning mechanism allows to keep the paraphrase grammar size manageable. Section 4 presents details on our paraphrase extraction module.

2 Compact Grammar Representation

Statistical machine translation systems tend to perform better when trained on larger amounts of bilingual parallel data. Using tools such as Thrax, translation models and their parameters are extracted and estimated from the data. In Joshua, translation models are represented as synchronous context-free grammars (SCFGs). An SCFG is a collection of

[1] joshua-decoder.org

rules $\{\mathbf{r}_i\}$ that take the form:

$$\mathbf{r}_i = C_i \rightarrow \langle \alpha_i, \gamma_i, \sim_i, \vec{\varphi}_i \rangle, \quad (1)$$

where *left-hand side* C_i is a nonterminal symbol, the *source side* α_i and the *target side* γ_i are sequences of both nonterminal and terminal symbols. Further, $\sim_i$ is a one-to-one correspondence between the nonterminal symbols of α_i and γ_i, and $\vec{\varphi}_i$ is a vector of features quantifying the probability of α_i translating to γ_i, as well as other characteristics of the rule (Weese et al., 2011). At decoding time, Joshua loads the grammar rules into memory in their entirety, and stores them in a trie data structure indexed by the rules' source side. This allows the decoder to efficiently look up rules that are applicable to a particular span of the (partially translated) input.

As the size of the training corpus grows, so does the resulting translation grammar. Using more diverse sets of nonterminal labels – which can significantly improve translation performance – further aggravates this problem. As a consequence, the space requirements for storing the grammar in memory during decoding quickly outgrow both practicality in a multi-user cluster environment. In some cases grammars may become too large to fit into the memory available on any one machine accessible to the user.

As an alternative to the commonly used trie structures based on hash maps, we propose a packed trie representation for SCFGs. The approach we take is similar to work on efficiently storing large phrase tables by Zens and Ney (2007) and language models by Heafield (2011) and Pauls and Klein (2011) – both language model implementations are now integrated with Joshua.

2.1 Packed Synchronous Tries

For our grammar representation, we break the SCFG up into three distinct structures. As Figure 1 indicates, we store the grammar rules' source sides $\{\alpha_i\}$, target sides $\{\gamma_i\}$, and feature data $\{\vec{\varphi}_i\}$ in separate formats of their own. Each of the structures is packed into a flat array, and can thus be quickly read into memory. All terminal and nonterminal symbols in the grammar are mapped to integer symbol id's using a globally accessible vocabulary map. We will now describe the implementation details for each representation and their interactions in turn.

2.1.1 Source-Side Trie

The source-side trie (or source trie) is designed to facilitate efficient lookup of grammar rules by source side, and to allow us to completely specify a matching set of rule with a single integer index into the trie. We store the source sides $\{\alpha_i\}$ of a grammar in a downward-linking trie, i.e. each trie node maintains a record of its children. The trie is packed into an array of 32-bit integers. Figure 1 illustrates the composition of a node in the source-side trie. All information regarding the node is stored in a contiguous block of integers, and decomposes into two parts: a *linking block* and a *rule block*.

The linking block stores the links to the child trie nodes. It consists of an integer n, the number of children, and n blocks of two integers each, containing the symbol id a_j leading to the child and the child node's address s_j (as an index into the source-side array). The children in the link block are sorted by symbol id, allowing for a lookup via binary or interpolation search.

The rule block stores all information necessary to reconstruct the rules that share the source side that led to the current source trie node. It stores the number of rules, m, and then a tuple of three integers for each of the m rules: we store the symbol id of the left-hand side, an index into the target-side trie and a *data block id*. The rules in the data block are initially in an arbitrary order, but are sorted by application cost upon loading.

2.1.2 Target-Side Trie

The target-side trie (or target trie) is designed to enable us to uniquely identify a target side γ_i with a single pointer into the trie, as well as to exploit redundancies in the target side string. Like the source trie, it is stored as an array of integers. However, the target trie is a *reversed*, or upward-linking trie: a trie node retains a link to its parent, as well as the symbol id labeling said link.

As illustrated in Figure 1, the target trie is accessed by reading an array index from the source trie, pointing to a trie node at depth d. We then follow the parent links to the trie root, accumulating target side symbols g_j into a target side string g_1^d as we go along. In order to match this traversal, the target strings are entered into the trie in reverse order, i.e. last word first. In order to determine d from a

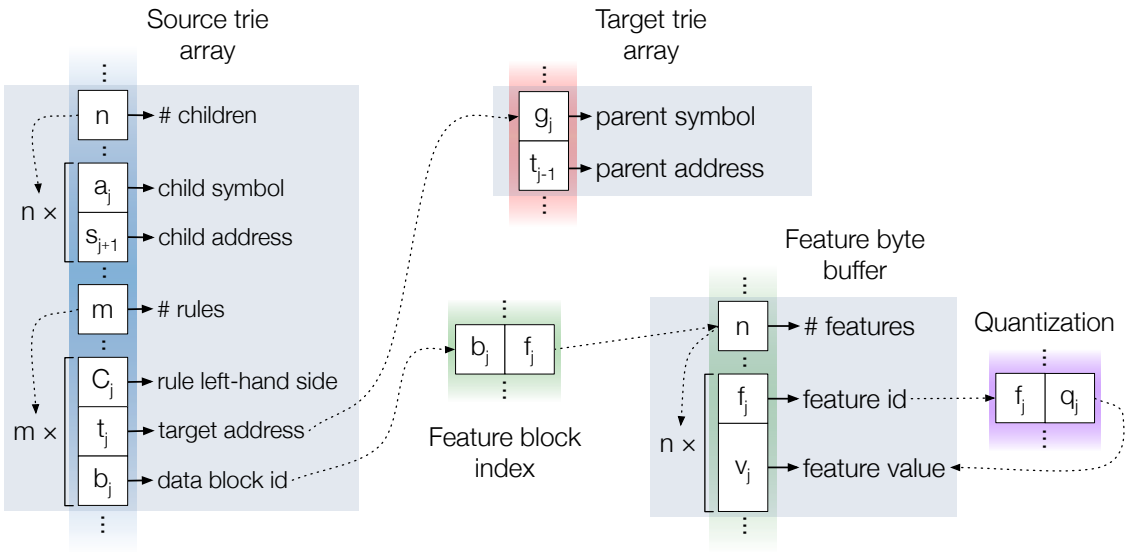

Figure 1: An illustration of our packed grammar data structures.

pointer into the target trie, we maintain an offset table in which we keep track of where each new trie level begins in the array. By first searching the offset table, we can determine d, and thus know how much space to allocate for the complete target side string.

To further benefit from the overlap there may be among the target sides in the grammar, we drop the nonterminal labels from the target string prior to inserting them into the trie. For richly labeled grammars, this collapses all lexically identical target sides that share the same nonterminal reordering behavior, but vary in nonterminal labels into a single path in the trie. Since the nonterminal labels are retained in the rules' source sides, we do not lose any information by doing this.

2.1.3 Features and Other Data

We designed the data format for the grammar rules' feature values to be easily extended to include other information that we may want to attach to a rule, such as word alignments, or locations of occurrences in the training data. In order to that, each rule $\mathbf{r}_i$ has a unique block id b_i associated with it. This block id identifies the information associated with the rule in every attached data store. All data stores are implemented as memory-mapped byte buffers that are only loaded into memory when actually requested by the decoder. The format for the feature data is detailed in the following.

The rules' feature values are stored as sparse features in contiguous blocks of variable length in a byte buffer. As shown in Figure 1, a lookup table is used to map the b_i to the index of the block in the buffer. Each block is structured as follows: a single integer, n, for the number of features, followed by n feature entries. Each feature entry is led by an integer for the feature id f_j, and followed by a field of variable length for the feature value v_j. The size of the value is determined by the type of the feature. Joshua maintains a quantization configuration which maps each feature id to a type handler or *quantizer*. After reading a feature id from the byte buffer, we retrieve the responsible quantizer and use it to read the value from the byte buffer.

Joshua's packed grammar format supports Java's standard primitive types, as well as an 8-bit quantizer. We chose 8 bit as a compromise between compression, value decoding speed and translation performance (Federico and Bertoldi, 2006). Our quantization approach follows Federico and Bertoldi (2006) and Heafield (2011) in partitioning the value histogram into 256 equal-sized buckets. We quantize by mapping each feature value onto the weighted average of its bucket. Quantizers can be easily configured on a per-feature level.

Grammar	Rules	Memory	Startup
Hash-based	43M	9.5G	1455 + 70s
Packed		3.6G	4 + 86s
+ Quantized			
Hash-based			
Packed			
+ Quantized			

Table 1: Memory consumption and loading times for the packed grammar versus the standard grammar format. The startup times are given as: time to load grammar + time to sort grammar.

2.2 Experiments

Table 1 shows a comparison of decoder startup times and memory consumption for our WMT12 French-English grammar. We can observe a dramatic decrease in grammar loading time, as well as a substantial decrease in memory consumption. The sorting time for the packed grammar is slightly longer due to the traversal of the packed feature data store taking longer.

Note to the reviewers: At the time of submission, we have not yet completed optimizing the packed grammar integration into the decoder. Table 1 as well as a more comprehensive set of experiments detailing memory use and decoding speed with packed grammars, as well as translation performance with various quantizer settings and very large grammars will be included in the camera-ready version of the paper.

3 J-PRO: Pairwise Ranking Optimization in Joshua

Pairwise ranking optimization (PRO) proposed by (Hopkins and May, 2011) is a new method for discriminative parameter tuning in statistical machine translation. It is reported to be more stable than the popular MERT algorithm (Och, 2003) and is more scalable with regard to the number of features. PRO treats parameter tuning as an n-best list reranking problem, and the idea is similar to other pairwise ranking techniques like ranking SVM and IR SVMs (Li, 2011). The algorithm can be described thusly:

Let $h(c) = \langle \mathbf{w}, \mathbf{\Phi}(c) \rangle$ be the linear model score of a candidate translation c, in which $\mathbf{\Phi}(c)$ is the feature vector of c and $\mathbf{w}$ is the parameter vector. Also let $g(c)$ be the metric score of c (without loss of generality, we assume a higher score indicates a better translation). We aim to find a parameter vector $\mathbf{w}$ such that for a pair of candidates $\{c_i, c_j\}$ in an n-best list,

$$(h(c_i) - h(c_j))(g(c_i) - g(c_j)) = \\ \langle \mathbf{w}, \mathbf{\Phi}(c_i) - \mathbf{\Phi}(c_j) \rangle (g(c_i) - g(c_j)) > 0,$$

namely the order of the model score is consistent with that of the metric score. This can be turned into a binary classification problem, by adding instance

$$\Delta\mathbf{\Phi}_{ij} = \mathbf{\Phi}(c_i) - \mathbf{\Phi}(c_j)$$

with class label $sign(g(c_i) - g(c_j))$ to the training data (and symmetrically add instance

$$\Delta\mathbf{\Phi}_{ji} = \mathbf{\Phi}(c_j) - \mathbf{\Phi}(c_i)$$

with class label $sign(g(c_j) - g(c_i))$ at the same time), then using any binary classifier to find the $\mathbf{w}$ which determines a hyperplane separating the two classes (therefore the performance of PRO depends on the choice of classifier to a large extent). Given a training set with T sentences, there are $O(Tn^2)$ pairs of candidates that can be added to the training set, this number is usually much too large for efficient training. To make the task more tractable, PRO samples a subset of the candidate pairs so that only those pairs whose metric score difference is large enough are qualified as training instances. This follows the intuition that high score differential makes it easier to separate good translations from bad ones.

3.1 Implementation

PRO is implemented in Joshua 4.0 named J-PRO. In order to ensure compatibility with the decoder and the parameter tuning module Z-MERT (Zaidan, 2009) included in all versions of Joshua, J-PRO is built upon the architecture of Z-MERT with similar usage and configuration files(with a few extra lines specifying PRO-related parameters). J-PRO inherits Z-MERT's ability to easily plug in new metrics. Since PRO allows using any off-the-shelf binary classifiers, J-PRO provides a Java interface that enables easy plug-in of any classifier. Currently, J-PRO supports three classifiers:

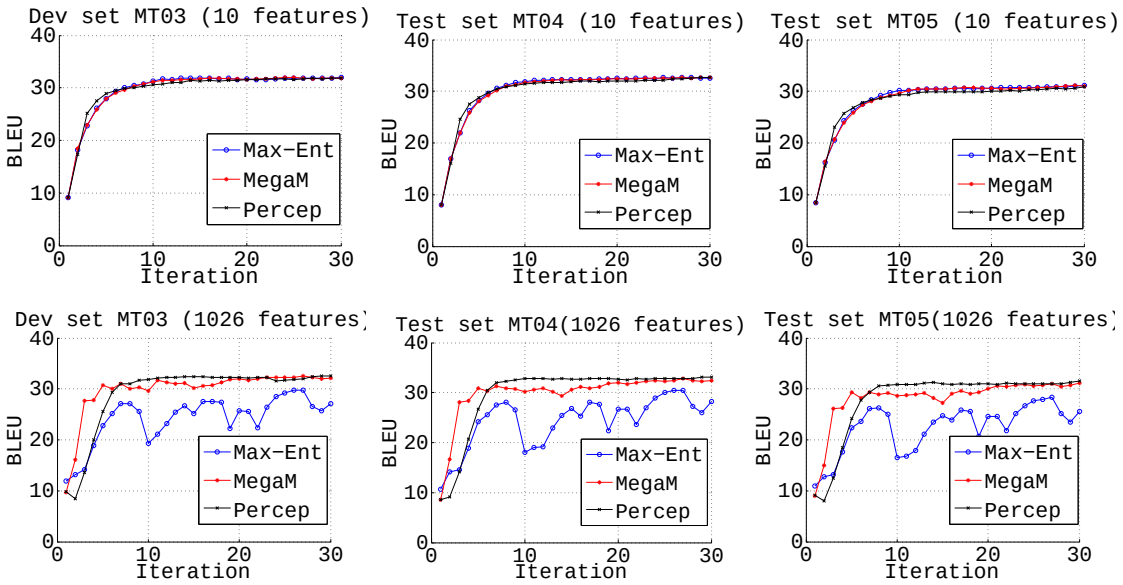

Figure 2: Experimental results on the development and test sets. The x-axis is the number of iterations (up to 30) and the y-axis is the BLEU score. The three curves in each figure correspond to three classifiers. Upper row: results trained using only dense features (10 features); Lower row: results trained using dense+sparse features (1026 features). Left column: development set (MT03); Middle column: test set (MT04); Right column: test set (MT05).

- *Perceptron* (Rosenblatt, 1958): the perceptron is self-contained in J-PRO, no external resources required.
- *MegaM* (Daumé III and Marcu, 2006): the classifier used by Hopkins and May (2011).[2]
- *Maximum entropy classifier* (Manning and Klein, 2003): the Stanford toolkit for maximum entropy classification.[3]

The user may specify which classifier he wants to use and the classifier-specific parameters in the J-PRO configuration file.

The PRO approach is capable of handling a large number of features, allowing the use of sparse discriminative features for machine translation. However, Hollingshead and Roark (2008) demonstrated that naively tuning weights for a heterogeneous feature set composed of both dense and sparse features can yield subpar results. Thus, to better handle the relation between dense and sparse features and provide a flexible selection of training schemes, J-PRO supports the following four training modes. We assume M dense features and N sparse features are used:

1. Tune the dense feature parameters only, just like Z-MERT (M parameters to tune).
2. Tune the dense + sparse feature parameters together ($M + N$ parameters to tune).
3. Tune the sparse feature parameters only with the dense feature parameters fixed, and sparse feature parameters scaled by a manually specified constant (N parameters to tune).
4. Tune the dense feature parameters and the scaling factor for sparse features, with the sparse feature parameters fixed (M+1 parameters to tune).

J-PRO supports n-best list input with a sparse feature format which enumerates only the firing features together with their values. This enables a more compact feature representation when numerous features are involved in training.

[2] `hal3.name/megam`
[3] `nlp.stanford.edu/software`

Datasets	Z-MERT	J-PRO		
		Percep	MegaM	Max-Ent
Dev (MT03)	32.2	31.9	32.0	32.0
Test (MT04)	32.6	32.7	32.7	32.6
Test (MT05)	30.7	30.9	31.0	30.9

Table 2: Comparison between the results given by Z-MERT and J-PRO (trained with 10 features).

3.2 Experiments

We did our experiments using J-PRO on the NIST Chinese-English data, and BLEU score was used as the quality metric for experiments reported in this section.[4] The experimental settings are as the following:

Datasets: MT03 dataset (998 sentences) as development set for parameter tuning, MT04 (1788 sentences) and MT05 (1082 sentences) as test sets.

Features: Dense feature set include the 10 regular features used in the Hiero system; Sparse feature set includes 1016 target-side rule POS bi-gram features as used in (Li et al., 2010b).

Classifiers: Perceptron, MegaM and Maximum entropy.

PRO parameters: $\Gamma = 8000$ (number of candidate pairs sampled uniformly from the n-best list), $\alpha = 1$ (sample acceptance probability), $\Xi = 50$ (number of top candidates to be added to the training set).

Figure 2 shows the BLEU score curves on the development and test sets as a function of iterations. The upper and lower rows correspond to the results trained with 10 dense features and 1026 dense+sparse features respectively. We intentionally selected very bad initial parameter vectors to verify the robustness of the algorithm. It can be seen that with each iteration, the BLEU score increases monotonically on both development and test sets, and begins to converge after a few iterations. When only 10 features are involved, all classifiers give almost the same performance. However, when scaled to over a thousand features, the maximum entropy classifier becomes unstable and the curve fluctuates significantly. In this situation MegaM behaves well, but the J-PRO built-in perceptron gives the most robust performance.

Table 2 compares the results of running Z-MERT and J-PRO. Since MERT is not able to handle numerous sparse features, we only report results for the 10-feature setup. The scores for both setups are quite close to each other, with Z-MERT doing slightly better on the development set but J-PRO yielding slightly better performance on the test set.

4 Thrax: Grammar Extraction at Scale

4.1 Translation Grammars

In previous years, our grammar extraction methods were limited by either memory-bounded extractors. Moving towards a parallelized grammar extraction process, we switched from Joshua's formerly built-in extraction module to Thrax for WMT11. However, we were limited to a simple pseudo-distributed Hadoop setup. In a pseudo-distributed cluster, all tasks run on separate cores on the same machine and access the local file system simultaneously, instead of being distributed over different physical machines and harddrives. This setup proved unreliable for larger extractions, and we were forced to reduce the amount of data that we used to train our translation models.

For this year, however, we had a permanent cluster at our disposal, which made it easy to extract grammars from all of the available WMT12 data. We found that on a properly distributed Hadoop setup Thrax was able to extract both Hiero grammars and the much larger SAMT grammars on the complete WMT12 training data for all tested language pairs. The runtimes and resulting (unfiltered) grammar sizes for each language pair are shown in Table 3 (for Hiero) and Table 4 (for SAMT).

4.2 Paraphrase Extraction

Recently English-to-English text generation tasks have seen renewed interest in the NLP commu-

[4] We also experimented with other metrics including TER, METEOR and TER-BLEU. Similar trends as reported in this section were observed. These results are omitted here due to limited space.

Source Bitext	Sentences	Words	Pruning	Rules
Fr – En	1.6M	45M	$p(e_1\|e_2), p(e_2\|e_1) > 0.001$	49M
{Da + Sv + Cs + De + Es + Fr} – En	9.5M	100M	$p(e_1\|e_2), p(e_2\|e_1) > 0.02$ $p(e_1\|e_2), p(e_2\|e_1) > 0.001$	31M 91M

Table 5: Large paraphrase grammars extracted using Thrax. The sentence and word counts refer to the English side of the bitexts used.

Language Pair	Time	Rules
Cs – En	4h41m	133M
De – En	5h20m	219M
Fr – En	16h47m	374M
Es – En	16h22m	413M

Table 3: Extraction times and grammar sizes for Hiero grammars using the Europarl and News Commentary training data for each listed language pair.

Language Pair	Time	Rules
Cs – En	7h59m	223M
De – En	9h18m	328M
Fr – En	25h46m	654M
Es – En	28h10m	716M

Table 4: Extraction times and grammar sizes for the SAMT grammars using the Europarl and News Commentary training data for each listed language pair.

nity. Paraphrases are a key component in large-scale state-of-the-art text-to-text generation systems. We present an extended version of Thrax that implements distributed, Hadoop-based paraphrase extraction via the pivoting approach (Bannard and Callison-Burch, 2005). Our toolkit is capable of extracting syntactically informed paraphrase grammars at scale. The paraphrase grammars obtained with Thrax have been shown to achieve state-of-the-art results on text-to-text generation tasks (Ganitkevitch et al., 2011).

For every supported translation feature, Thrax implements a corresponding *pivoted feature* for paraphrases. The pivoted features are set up to be aware of the prerequisite translation features they are derived from. This allows Thrax to automatically detect the needed translation features and spawn the corresponding map-reduce passes before the pivoting stage takes place. In addition to features useful for translation, Thrax also offers a number of features geared towards text-to-text generation tasks such as sentence compression or text simplification.

Due to the long tail of translations in unpruned translation grammars and the combinatorial effect of pivoting, paraphrase grammars can easily grow very large. We implement a simple feature-level pruning approach that allows the user to specify upper or lower bounds for any pivoted feature. If a paraphrase rule is not within these bounds, it is discarded. Additionally, pivoted features are aware of the bounding relationship between their value and the value of their prerequisite translation features (i.e. whether the pivoted feature's value can be guaranteed to never be larger than the value of the translation feature). Thrax uses this knowledge to discard overly weak translation rules before the pivoting stage, leading to a substantial speedup in the extraction process.

Table 5 gives a few examples of large paraphrase grammars extracted from WMT training data. With appropriate pruning settings, we are able to obtain paraphrase grammars estimated over bitexts with more than 100 million words.

5 Additional New Features

- With the help of the respective original authors, the language model implementations by Heafield (2011) and Pauls and Klein (2011) have been integrated with Joshua, dropping support for the slower and more difficult to compile SRILM toolkit (Stolcke, 2002).

- We modified Joshua so that it can be used as a parser to analyze pairs of sentences using a synchronous context-free grammar. We implemented the two-pass parsing algorithm of Dyer (2010).

References

Colin Bannard and Chris Callison-Burch. 2005. Paraphrasing with bilingual parallel corpora. In *Proceedings of ACL*.

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

Hal Daumé III and Daniel Marcu. 2006. Domain adaptation for statistical classifiers. *Journal of Artificial Intelligence Research*, 26(1):101–126.

Chris Dyer. 2010. Two monolingual parses are better than one (synchronous parse). In *Proceedings of HLT/NAACL*, pages 263–266. Association for Computational Linguistics.

Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation? In *Proceedings of WMT06*, pages 94–101. Association for Computational Linguistics.

Juri Ganitkevitch, Chris Callison-Burch, Courtney Napoles, and Benjamin Van Durme. 2011. Learning sentential paraphrases from bilingual parallel corpora for text-to-text generation. In *Proceedings of EMNLP*.

Kenneth Heafield. 2011. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197. Association for Computational Linguistics.

Kristy Hollingshead and Brian Roark. 2008. Reranking with baseline system scores and ranks as features. Technical report, Center for Spoken Language Understanding, Oregon Health & Science University.

Mark Hopkins and Jonathan May. 2011. Tuning as ranking. In *Proceedings of EMNLP*.

Zhifei Li, Chris Callison-Burch, Chris Dyer, Juri Ganitkevitch, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. 2009. Joshua: An open source toolkit for parsing-based machine translation. In *Proc. WMT*, Athens, Greece, March.

Zhifei Li, Chris Callison-Burch, Chris Dyer, Juri Ganitkevitch, Ann Irvine, Sanjeev Khudanpur, Lane Schwartz, Wren N.G. Thornton, Ziyuan Wang, Jonathan Weese, and Omar F. Zaidan. 2010a. Joshua 2.0: a toolkit for parsing-based machine translation with syntax, semirings, discriminative training and other goodies. In *Proc. WMT*.

Zhifei Li, Ziyuan Wang, and Sanjeev Khudanpur. 2010b. Unsupervised discriminative language model training for machine translation using simulated confusion sets. In *Proceedings of COLING*, Beijing, China, August.

Hang Li. 2011. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool Publishers.

Chris Manning and Dan Klein. 2003. Optimization, maxent models, and conditional estimation without magic. In *Proceedings of HLT/NAACL*, pages 8–8. Association for Computational Linguistics.

Franz Och. 2003. Minimum error rate training in statistical machine translation. In *Proceedings of the 41rd Annual Meeting of the Association for Computational Linguistics (ACL-2003)*, Sapporo, Japan.

Adam Pauls and Dan Klein. 2011. Faster and smaller n-gram language models. In *Proceedings of ACL*, pages 258–267, Portland, Oregon, USA, June. Association for Computational Linguistics.

Frank Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Andreas Stolcke. 2002. Srilm - an extensible language modeling toolkit. In *Seventh International Conference on Spoken Language Processing*.

Jonathan Weese, Juri Ganitkevitch, Chris Callison-Burch, Matt Post, and Adam Lopez. 2011. Joshua 3.0: Syntax-based machine translation with the Thrax grammar extractor. In *Proceedings of WMT11*.

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*, 91:79–88.

Richard Zens and Hermann Ney. 2007. Efficient phrase-table representation for machine translation with applications to online MT and speech translation. In *Proceedings of HLT/NAACL*, pages 492–499, Rochester, New York, April. Association for Computational Linguistics.