Phrasal: A Toolkit for New Directions in Statistical Machine Translation

Spence Green, Daniel Cer, and **Christopher D. Manning**
Computer Science Department, Stanford University
{spenceg,danielcer,manning}@stanford.edu

Abstract

We present a new version of Phrasal, an open-source toolkit for statistical phrase-based machine translation. This revision includes features that support emerging research trends such as (a) tuning with large feature sets, (b) tuning on large datasets like the bitext, and (c) web-based interactive machine translation. A direct comparison with Moses shows favorable results in terms of decoding speed and tuning time.

1 Introduction

In the early part of the last decade, phrase-based machine translation (MT) (Koehn et al., 2003) emerged as the preeminent design of statistical MT systems. However, most systems were proprietary or closed-source, so progress was initially constrained by the high engineering barrier to entry into the field. Then Moses (Koehn et al., 2007) was released. What followed was a flowering of work on all aspects of the translation problem, from rule extraction to deployment issues. Other toolkits appeared including Joshua (Post et al., 2013), Jane (Wuebker et al., 2012), cdec (Dyer et al., 2010) and the first version of our package, Phrasal (Cer et al., 2010), a Java-based, open source package.

This paper presents a completely re-designed release of Phrasal that lowers the barrier to entry into several exciting areas of MT research. First, Phrasal exposes a simple yet flexible feature API for building large-scale, feature-rich systems. Second, Phrasal provides multi-threaded decoding and online tuning for learning feature-rich models on very large datasets, including the bitext. Third, Phrasal supplies the key ingredients for web-based, interactive MT: an asynchronous RESTful JSON web service implemented as a J2EE servlet, integrated pre- and post-processing, and fast search.

Revisions to Phrasal were guided by several design choices. First, we optimized the system for multi-core architectures, eschewing distributed infrastructure like Hadoop and MapReduce. While "scaling-out" with distributed infrastructure is the conventional industry and academic choice, we find that "scaling-up" on a single large-node is an attractive yet overlooked alternative (Appuswamy et al., 2013). A single "scale-up" node is usually competitive in terms of cost and performance, and multi-core code has fewer dependencies in terms of software and expertise. Second, Phrasal makes extensive use of Java interfaces and reflection. This is especially helpful in the feature API. A feature function can be added to the system by simply implementing an interface and specifying the class name on the decoder command line. There is no need to modify or recompile anything other than the new feature function.

This paper presents a direct comparison of Phrasal and Moses that shows favorable results in terms of decoding speed and tuning time. An indirect comparison via the WMT2014 shared task (Neidert et al., 2014) showed that Phrasal compares favorably to Moses in an evaluation setting. The source code is freely available at: `http://nlp.stanford.edu/software/phrasal/`

2 Standard System Pipeline

This section describes the steps required to build a phrase-based MT system from raw text. Each step is implemented as a stand-alone executable. For convenience, the Phrasal distribution includes a script that coordinates the steps.

2.1 Prerequisites

Phrasal assumes offline preparation of word alignments and at least one target-side language model.

Word Alignment The rule extractor can accommodate either unsymmetrized or symmetrized alignments. Unsymmetrized alignments can be produced with either GIZA++ or the Berkeley Aligner (Liang et al., 2006). Phrasal then applies symmetrization on-the-fly using heuristics such as grow-diag or grow-diag-final. If the alignments are symmetrized separately, then Phrasal accepts alignments in the i-j Pharaoh format, which indicates that source token i is aligned to target token j.

Language Modeling Phrasal can load any n-gram language model saved in the ARPA format. There are two LM loaders. The Java-based loader is used by default and is appropriate for small-scale experiments and pure-Java environments. The C++ KenLM (Heafield, 2011) loader[1] is best for large-scale LMs such as the unfiltered models produced by lmplz (Heafield et al., 2013). Profiling shows that LM queries often account for more than 50% of the CPU time in a Phrasal decoding run, so we designed the Phrasal KenLM loader to execute queries mostly in C++ for efficiency. The KenLM binding efficiently passes full strings to C++ via JNI. KenLM then iterates over the string, returning a score and a state length. Phrasal can load multiple language models, and includes native support for the class-based language models that have become popular in recent evaluations (Wuebker et al., 2012; Ammar et al., 2013; Durrani et al., 2013).

2.2 Rule Extraction

The next step in the pipeline is extraction of a phrase table. Phrasal includes a multi-threaded version of the rule extraction algorithm of Och and Ney (2004). Phrase tables can be filtered to a specific data set—as is common in research environments. When filtering, the rule extractor lowers memory utilization by splitting the data into arbitrary-sized chunks and extracting rules from each chunk.

The rule extractor includes a feature API that is independent of the decoder feature API. This allows for storage of **static rule feature** values in the phrase table. Static rule features are useful in two cases. First, if a feature value depends on bitext statistics, which are not accessible during tuning or decoding, then that feature should be stored in the phrase table. Examples are the standard phrase translation probabilities, and the dense rule count and rule uniqueness indicators described by Green et al. (2013). Second, if a feature depends only on the rule and is unlikely to change, then it may be more efficient to store that feature value in the phrase table. An example is a feature template that indicates inclusion in a specific data domain (Durrani et al., 2013). Rule extractor feature templates must implement the `FeatureExtractor` interface and are loaded via reflection.

[1]Invoked by prefixing the LM path with the "kenlm:".

The rule extractor can also create lexicalized reordering tables. The standard phrase orientation model (Tillmann, 2004) and the hierarchical model of Galley and Manning (2008) are available.

2.3 Tuning

Once a language model has been estimated and a phrase table has been extracted, the next step is to estimate model weights. Phrasal supports tuning over n-best lists, which permits rapid experimentation with different error metrics and loss functions. Lattice-based tuning, while in principle more powerful, requires metrics and losses that factor over lattices, and in practice works no better than n-best tuning (Cherry and Foster, 2012).

Tuning requires a parallel set $\{(f_t, e_t)\}_{t=1}^{T}$ of source sentences f_t and target references e_t.[2] Phrasal follows the log-linear approach to phrase-based translation (Och and Ney, 2004) in which the predictive translation distribution $p(e|f;w)$ is modeled directly as

$$p(e|f;w) = \frac{1}{Z(f)} \exp\left[w^\top \phi(e,f)\right] \qquad (1)$$

where $w \in \mathbb{R}^d$ is the vector of model parameters, $\phi(\cdot) \in \mathbb{R}^d$ is a feature map, and $Z(f)$ is an appropriate normalizing constant.

MT differs from other machine learning settings in that it is not common to tune to log-likelihood under (1). Instead, a gold error metric $G(e', e)$ is chosen that specifies the similarity between a hypothesis e' and a reference e, and that error is minimized over the tuning set. Phrasal includes Java implementations of BLEU (Papineni et al., 2002), NIST, and WER, and bindings for TER (Snover et al., 2006) and METEOR (Denkowski and Lavie, 2011). The error metric is incorporated into a loss function ℓ that returns the loss at either the sentence- or corpus- level.

For conventional corpus-level (batch) tuning, Phrasal includes multi-threaded implementations of MERT (Och, 2003) and PRO (Hopkins and May, 2011). The MERT implementation uses the line search of Cer et al. (2008) to directly minimize corpus-level error. The PRO implementation uses a pairwise logistic loss to minimize the number of inversions in the ranked n-best lists. These batch implementations accumulate n-best lists across epochs.

[2]For simplicity, we assume one reference, but the multi-reference case is analogous.

Online tuning is faster and more scalable than batch tuning, and sometimes leads to better solutions for non-convex settings like MT (Bottou and Bousquet, 2011). Weight updates are performed after each tuning example is decoded, and n-best lists are not accumulated. Consequently, online tuning is preferable for large tuning sets, or for rapid iteration during development. Phrasal includes the AdaGrad-based (Duchi et al., 2011) tuner of Green et al. (2013). The regularization options are L_2, efficient L_1 for feature selection (Duchi and Singer, 2009), or $L_1 + L_2$ (elastic net). There are two online loss functions: a pairwise (PRO) objective and a listwise minimum expected error objective (Och, 2003). These online loss functions require sentence-level error metrics, several of which are available in the toolkit: BLEU+1 (Lin and Och, 2004), Nakov BLEU (Nakov et al., 2012), and TER.

2.4 Decoding

The Phrasal decoder can be invoked either programmatically as a Java object or as a standalone application. In both cases the decoder is configured via options that specify the language model, phrase table, weight vector w, etc. The decoder is multithreaded, with one decoding instance per thread. Each decoding instance has its own weight vector, so in the programmatic case, it is possible to decode simultaneously under different weight vectors.

Two search procedures are included. The default is the phrase-based variant of cube pruning (Huang and Chiang, 2007). The standard multi-stack beam search (Och and Ney, 2004) is also an option. Either procedure can be configured in one of several recombination modes. The "Pharaoh" mode only considers linear distortion, source coverage, and target LM history. The "Exact" mode considers these states in addition to any feature that declares recombination state (see section 3.3).

The decoder includes several options for deployment environments such as an unknown word API, pre-/post-processing APIs, and both full and prefix-based force decoding.

2.5 Evaluation and Post-processing

All of the error metrics available for tuning can also be invoked for evaluation. For significance testing, the toolkit includes an implementation of the permutation test of Riezler and Maxwell (2005), which was shown to be less susceptible to Type-I error than bootstrap re-sampling (Koehn, 2004).

$\dfrac{}{r : s(r,w)}$	$r \in R$	axiom
$\dfrac{d : w(d) \quad r : s(r,w)}{d' : s(d',w)}$	$r \notin cov(d)$	item
$\lvert cov(d)\rvert = \lvert s\rvert$		goal

Table 1: Phrase-based MT as deductive inference. This notation can be read as follows: if the antecedents on the top are true, then the consequent on the bottom is true subject to the conditions on the right. The new item d' is creating by appending r to the ordered sequence of rules that define d.

Phrasal also includes two truecasing packages. The LM-based truecaser (Lita et al., 2003) requires an LM estimated from cased, tokenized text. A subsequent detokenization step is thus necessary. A more convenient alternative is the CRF-based post-processor that can be trained to invert an arbitrary pre-processor. This post-processor can perform truecasing and detokenization in one pass.

3 Feature API

Phrasal supports dynamic feature extraction during tuning and decoding. In the API, feature templates are called **featurizers**. There are two types with associated interfaces: `RuleFeaturizer` and `DerivationFeaturizer`. One way to illustrate these two featurizers is to consider phrase-based decoding as a deductive system. Let $r = \langle f, e\rangle$ be a rule in a set R, which is conventionally called the phrase table. Let $d = \{r_i\}_{i=1}^{N}$ be an ordered sequence of derivation N rules called a derivation, which specifies a translation for some source input sequence s (which, by some abuse of notation, is equivalent to f in Eq. (1)). Finally, define functions $cov(d)$ as the source coverage set of d as a bit vector and $s(\cdot, w)$ as the score of a rule or derivation under w.[3] The expression $r \notin cov(d)$ means that r maps to an empty/uncovered span in $cov(d)$. Table 1 shows the deductive system.

3.1 Dynamic Rule Features

RuleFeaturizers are invoked when scoring axioms, which do not require any derivation context. The static rule features described in section 2.2 also contribute to axiom scoring, and differ only from RuleFeaturizers in that they are stored permanently in the phrase table. In contrast, RuleFeaturizers

[3]Note that $s(d, w) = w^{\top}\phi(d)$ in the log-linear formulation of MT (see Eq. (1)).

Listing 1: A `RuleFeaturizer`, which depends only on a translation rule.

```
public class WordPenaltyFeaturizer
 implements RuleFeaturizer {

 @Override
 public List<FeatureValue>
  ruleFeaturize(Featurizable f) {

  List<FeatureValue> features =
   Generics.newLinkedList();

  // Extract single feature
  features.add(new FeatureValue(
   "WordPenalty", f.targetPhrase.size()));

  return features;
 }
}
```

are extracted during decoding. An example feature template is the word penalty, which is simply the dimension of the target side of r (Listing 1).

`Featurizable` wraps decoder state from which features can be extracted. RuleFeaturizers are extracted during each phrase table query and cached, so they can be simply efficiently retrieved during decoding.

Once the feature is compiled, it is simply specified on the command-line when the decoder is executed. No other configuration is required.

3.2 Derivation Features

DerivationFeaturizers are invoked when scoring items, and thus depend on some derivation context. An example is the LM, which requires the n-gram context from d to score r when creating the new hypothesis d' (Listing 2).

The LM featurizer first looks up the recombination state of the derivation, which contains the n-gram context. Then it queries the LM by passing the rule and context, and sets the new state as the result of the LM query. Finally, it returns a feature "LM" with the value of the LM query.

3.3 Recombination State

Listing 2 shows a state lookup during feature extraction. Phrase-based MT feature design differs significantly from that of convex classifiers in terms of the interaction with inference. For example, in a maximum entropy classifier inference is exact, so a good optimizer can simply nullify bad features to retain baseline accuracy. In contrast, MT feature templates affect search through both future cost heuristics and recombination state. Bad features can introduce search errors and thus decrease

Listing 2: A `DerivationFeaturizer`, which must lookup and save recombination state for extraction.

```
public class NGramLanguageModelFeaturizer
 extends DerivationFeaturizer {

 @Override
 public List<FeatureValue> featurize(
  Featurizable f) {

  // Get recombination state
  LMState priorState = f.prior.getState(this);

  // LM query
  LMState state = lm.score(f.targetPhrase, priorState);

  List<FeatureValue> features =
    Generics.newLinkedList();

  // Extract single feature
  features.add(
    new FeatureValue("LM", state.getScore()));

  // Set new recombination state
  f.setState(this, state);

  return features;
 }
}
```

accuracy, sometimes catastrophically.

The feature API allows DerivationFeaturizers to explicitly declare recombination state via the `FeaturizerState` interface.[4] The interface requires a state equality operator and a hash code function. Then the search procedure will only recombine derivations with equal states. For example, the state of the n-gram LM DerivationFeaturizer (Listing 2) is the n-1 gram context, and the hashcode is a hash of that context string. Only derivations for which the equality operator of LMState returns true can be recombined.

4 Web Service

Machine translation output is increasingly utilized in computer-assisted translation (CAT) workbenches. To support deployment, Phrasal includes a lightweight J2EE servlet that exposes a RESTful JSON API for querying a trained system. The toolkit includes a standalone servlet container, but the servlet may also be incorporated into a J2EE server. The servlet requires just one input parameter: the Phrasal configuration file, which is also used for tuning and decoding. Consequently, after running the standard pipeline, the trained system can be deployed with one command.

[4]To control future cost estimation, the designer would need to write a new heuristic that considers perhaps a subset of the full feature map. There is a separate API for future cost heuristics.

4.1 Standard Web Service

The standard web service supports two types of requests. The first is `TranslationRequest`, which performs full decoding on a source input. The JSON message structure is:

Listing 3: `TranslationRequest` message.

```
TranslationRequest {
  srcLang :(string),
  tgtLang :(string),
  srcText :(string),
  tgtText :(string),
  limit :(integer),
  properties :(object)
}
```

The `srcLang` and `tgtLang` fields are ignored by the servlet, but can be used by a middleware proxy to route requests to Phrasal servlet instances, one per language pair. The `srcText` field is the source input, and `properties` is a Javascript associative array that can contain key/value pairs to pass to the feature API. For example, we often use the `properties` field to pass domain information with each request.

Phrasal will perform full decoding and respond with the message:

Listing 4: `TranslationReply` message, which is returned upon successful processing of `TranslationRequest`.

```
TranslationReply {
  resultList :[
   {tgtText :(string),
    align :(string),
    score :(float)
   },...]
}
```

`resultList` is a ranked n-best list of translations, each with target tokens, word alignments, and a score.

The second request type is `RuleRequest`, which enables phrase table queries. These requests are processed very quickly since decoding is not required. The JSON message structure is:

Listing 5: `RuleRequest` message, which prompts a direct lookup into the phrase table.

```
RuleRequest {
  srcLang :(string),
  tgtLang :(string),
  srcText :(string),
  limit :(integer),
  properties :(object)
}
```

`limit` is the maximum number of translations to return. The response message is analogous to that for TranslationRequest, so we omit it.

4.2 Interactive Machine Translation

Interactive machine translation (Bisbey and Kay, 1972) pairs human and machine translators in hopes of increasing the throughput of high quality translation. It is an old idea that is again in focus. One challenge is to present relevant machine suggestions to humans. To that end, Phrasal supports context-sensitive translation queries via prefix decoding. Consider again the `TranslationRequest` message. When the `tgtText` field is empty, the source input is decoded from scratch. But when this field contains a prefix, Phrasal returns translations that begin with the prefix. The search procedure force decodes the prefix, and then completes the translation via conventional decoding. Consequently, if the user has typed a partial translation, Phrasal can suggest completions conditioned on that prefix. The longer the prefix, the faster the decoding, since the user prefix constrains the search space. This feature allows Phrasal to produce increasingly precise suggestions as the user works.

5 Experiments

We compare Phrasal and Moses by restricting an existing large-scale system to a set of common features. We start with the Arabic–English system of Green et al. (2014), which is built from 6.6M parallel segments. The system includes a 5-gram English LM estimated from the target-side of the bitext and 990M English monolingual tokens. The feature set is their dense baseline, but without lexicalized reordering and the two extended phrase table features. This leaves the nine baseline features also implemented by Moses. We use the same phrase table, phrase table query limit (20), and distortion limit (5) for both decoders. The tuning set (mt023568) contains 5,604 segments, and the development set (mt04) contains 1,075 segments.

We ran all experiments on a dedicated server with 16 physical cores and 128GB of memory.

Figure 1 shows single-threaded decoding time of the dev set as a function of the cube pruning pop limit. At very low limits Moses is faster than Phrasal, but then slows sharply. In contrast, Phrasal scales linearly and is thus faster at higher pop limits.

Figure 2 shows multi-threaded decoding time of the dev set with the cube pruning pop limit fixed at 1,200. Here Phrasal is initially faster, but Moses becomes more efficient at four threads. There are two possible explanations. First, profiling shows that LM queries account for approximately 75%

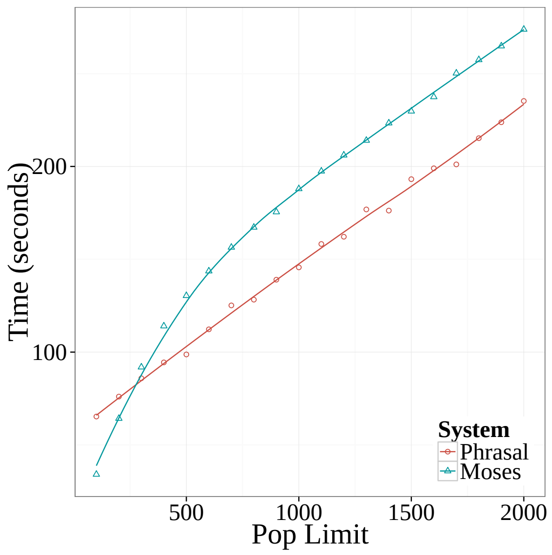

Figure 1: Development set decoding time as a function of the cube pruning pop limit.

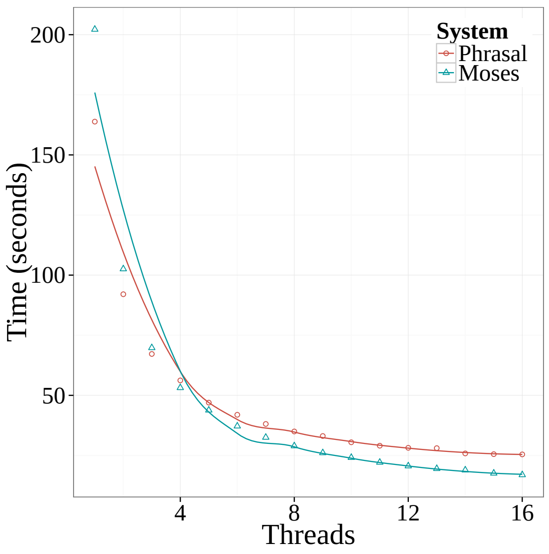

Figure 2: Development set decoding time as a function of the threadpool size.

of the Phrasal CPU-time. KenLM is written in C++, and Phrasal queries it via JNI. It appears as though multi-threading across this boundary is a source of inefficiency. Second, we observe that the Java parallel garbage collector (GC) runs up to seven threads, which become increasingly active as the number of decoder threads increases. These and other Java overhead threads must be scheduled, limiting gains as the number of decoding threads approaches the number of physical cores.

Finally, Figure 3 shows tuning BLEU as a function of wallclock time. For Moses we chose the batch MIRA implementation of Cherry and Foster (2012), which is popular for tuning feature-rich systems. Phrasal uses the online tuner with the expected BLEU objective (Green et al., 2014). Moses achieves a maximum BLEU score of 47.63 after 143 minutes of tuning, while Phrasal reaches this level after just 17 minutes, later reaching a maximum BLEU of 47.75 after 42 minutes. Much of the speedup can be attributed to phrase table and LM loading time: the Phrasal tuner loads these data structures just once, while the Moses tuner loads them every epoch. Of course, this loading time becomes more significant with larger-scale systems.

6 Conclusion

We presented a revised version of Phrasal, an open-source, phrase-based MT toolkit. The revisions support new directions in MT research including feature-rich models, large-scale tuning, and web-based interactive MT. A direct comparison with Moses showed favorable performance on a large-scale translation system.

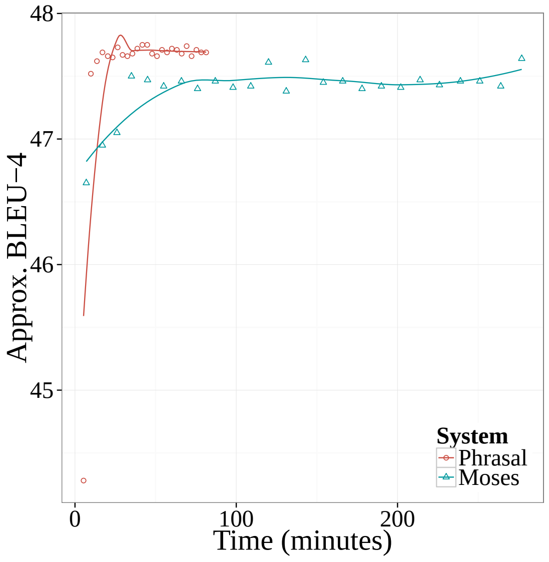

Figure 3: Approximate BLEU-4 during tuning as a function of time over 25 tuning epochs. The horizontal axis is accumulated time, while each point indicates BLEU at the end of an epoch.

Acknowledgments We thank Michel Galley for previous contributions to Phrasal. The first author is supported by a National Science Foundation Graduate Research Fellowship. This work was supported by the Defense Advanced Research Projects Agency (DARPA) Broad Operational Language Translation (BOLT) program through IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of DARPA or the US government.

References

W. Ammar, V. Chahuneau, M. Denkowski, G. Hanneman, W. Ling, A. Matthews, et al. 2013. The CMU machine translation systems at WMT 2013: Syntax, synthetic translation options, and pseudo-references. In *WMT*.

R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. 2013. Nobody ever got fired for buying a cluster. Technical report, Microsoft Corporation, MSR-TR-2013-2.

R. Bisbey and Kay. 1972. The MIND translation system: a study in man-machine collaboration. Technical Report P-4786, Rand Corp., March.

L. Bottou and O. Bousquet. 2011. The tradeoffs of large scale learning. In *Optimization for Machine Learning*, pages 351–368. MIT Press.

D. Cer, D. Jurafsky, and C. D. Manning. 2008. Regularization and search for minimum error rate training. In *WMT*.

D. Cer, M. Galley, D. Jurafsky, and C. D. Manning. 2010. Phrasal: A statistical machine translation toolkit for exploring new model features. In *HLT-NAACL, Demonstration Session*.

C. Cherry and G. Foster. 2012. Batch tuning strategies for statistical machine translation. In *HLT-NAACL*.

M. Denkowski and A. Lavie. 2011. Meteor 1.3: Automatic metric for reliable optimization and evaluation of machine translation systems. In *WMT*.

J. Duchi and Y. Singer. 2009. Efficient online and batch learning using forward backward splitting. *JMLR*, 10:2899–2934.

J. Duchi, E. Hazan, and Y. Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159.

N. Durrani, B. Haddow, K. Heafield, and P. Koehn. 2013. Edinburgh's machine translation systems for European language pairs. In *WMT*.

C. Dyer, A. Lopez, J. Ganitkevitch, J. Weese, F. Ture, et al. 2010. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *ACL System Demonstrations*.

M. Galley and C. D. Manning. 2008. A simple and effective hierarchical phrase reordering model. In *EMNLP*.

S. Green, S. Wang, D. Cer, and C. D. Manning. 2013. Fast and adaptive online training of feature-rich translation models. In *ACL*.

S. Green, D. Cer, and C. D. Manning. 2014. An empirical comparison of features and tuning for phrase-based machine translation. In *WMT*.

K. Heafield, I. Pouzyrevsky, J. H. Clark, and P. Koehn. 2013. Scalable modified Kneser-Ney language model estimation. In *ACL, Short Papers*.

K. Heafield. 2011. KenLM: Faster and smaller language model queries. In *WMT*.

M. Hopkins and J. May. 2011. Tuning as ranking. In *EMNLP*.

L. Huang and D. Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In *ACL*.

P. Koehn, F. J. Och, and D. Marcu. 2003. Statistical phrase-based translation. In *NAACL*.

P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *ACL, Demonstration Session*.

P. Koehn. 2004. Statistical significance tests for machine translation evaluation. In *EMNLP*.

P. Liang, B. Taskar, and D. Klein. 2006. Alignment by agreement. In *NAACL*.

C.-Y. Lin and F. J. Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING*.

L. V. Lita, A. Ittycheriah, S. Roukos, and N. Kambhatla. 2003. tRuEcasIng. In *ACL*.

P. Nakov, F. Guzman, and S. Vogel. 2012. Optimizing for sentence-level BLEU+1 yields short translations. In *COLING*.

J. Neidert, S. Schuster, S. Green, K. Heafield, and C. D. Manning. 2014. Stanford University's submissions to the WMT 2014 translation task. In *WMT*.

F. J. Och and H. Ney. 2004. The alignment template approach to statistical machine translation. *Computational Linguistics*, 30(4):417–449.

F. J. Och. 2003. Minimum error rate training for statistical machine translation. In *ACL*.

K. Papineni, S. Roukos, T. Ward, and W. Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *ACL*.

M. Post, J. Ganitkevitch, L. Orland, J. Weese, Y. Cao, and C. Callison-Burch. 2013. Joshua 5.0: Sparser, better, faster, server. In *WMT*.

S. Riezler and J. T. Maxwell. 2005. On some pitfalls in automatic evaluation and significance testing in MT. In *ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*.

M. Snover, B. Dorr, R. Schwartz, L. Micciulla, and J. Makhoul. 2006. A study of translation edit rate with targeted human annotation. In *AMTA*.

C. Tillmann. 2004. A unigram orientation model for statistical machine translation. In *NAACL*.

J. Wuebker, M. Huck, S. Peitz, M. Nuhn, M. Freitag, J. T. Peter, S. Mansour, and H. Ney. 2012. Jane 2: Open source phrase-based and hierarchical statistical machine translation. In *COLING: Demonstration Papers*.