PBML

logo

The Prague Bulletin of Mathematical Linguistics

NUMBER ??? JANUARY 2016 ??–???

Updating the Feature Function Framework in the Moses Decoder

Hieu Hoang[a], Kenneth Heafield[a], Barry Haddow[a], Matt Post[b], Eva Hasler[a], Phil Williams[a], Chris Dyer[c], Philipp Koehn[a]

[a] University of Edinburgh
[b] Human Language Technology Center of Excellence, Johns Hopkins University
[c] Carnegie Mellon University

Abstract

Best practices for developing a SMT decoder have progressed enormously since the initial release of the Moses decoder in 2006. The emergence of syntax-inspired models, sparse features and more features, have been the driver for the major changes in the framework of the decoder.

In this paper, we describe the changes in the Moses decoder and supporting programs to support such changes, while preserving the toolkit's existing functionality. Particular emphasis will be place in describing the feature function framework.

We compare the implementation of the Moses SCFG decoder to that of cdec and Joshua and find that there are some surprising differences in implementation of even very fundamental features in each decoder.

1. Introduction

In terms of extensibility, the Moses(**?**) decoder was a vast improvement over its immediate predecessor, Pharoah(**?**). For instance, Moses allows multiple translation model and language model *instances* and *implementations*.

Allowing multiple instances has enabled researchers to combine multiple datasets in novel ways such as the use of secondary models with differing linguistic information(**???**). Allowing multiple implementations has spurred on the developments of smaller, faster, better implementation of language models and phrase tables(**???**).

Corresponding author: **MISSING** `e-mail`
Cite as: Hieu Hoang, Kenneth Heafield, Barry Haddow, Matt Post, Eva Hasler, Phil Williams, Chris Dyer, Philipp Koehn. Updating the Feature Function Framework in the Moses Decoder. The Prague Bulletin of Mathematical Linguistics No. ???, 2016, pp. ??–???. doi: HERE-WILL-BE-A-DOI-NUMBER

However, a direction of machine translation research in the last few years has been on improving translation quality with the addition of novel feature functions, for example, (**?**). The use of sparse features during decoding has also been popular. Research such as (**??**) typifies this approach.

Enabling novel features functions in Moses requires a framework where new feature functions can be easily added. Adding support for sparse features entails fundamental changes in the way features function scores are stored and calculated. It is generally known that MERT(**?**) is unstable with a large number of scores, using sparse features will also require tuning algorithms which are able to deal with a large, non-predetermined number of feature scores.

This paper describes the changes to the Moses decoder in order to incorporate more feature functions, sparse feature scores, and compares it with that implemented in cdec(**?**) and Joshua(**?**).

2. Feature Function Framework

2.1. Original Framework

The standard feature functions in the phrase-based models are:

- Translation model
- Language model
- Distortion model
- Lexicalized reordering model
- Word penalty
- Unknown word penalty

The hierarchical and syntax model also uses these feature functions, except for the distortion and lexicalized reordering model.

Each feature function is implemented as a separate class, with no common class hierarchy. The initialisation and evaluation of each feature function was entirely bespoked.

Feature function *state* information is an essential component of the search procedure in decoding. State information control hypothesis recombination, enabling the decoder to effectively search through many more hypotheses than it actually has. Higher recombination results in higher effective number of hypotheses created, leading to better model score.

Initially, state information was only available for language models, therefore, only language models states are compared.

2.2. Current Framework

All feature functions inherit indirectly from the class *FeatureFunction*, which provides basic services to all feature functions such as storing the name and number of dense features, and providing a central repository of all feature functions.

Each feature function can be stateless or stateful, depending on whether it emits a state for each hypothesis. Stateful features inherits from *StatefulFeatureFunction*, otherwise they inherits from *StatelessFeatureFunction*.

The state comparison of the original framework was generalised to compare the state of all stateful feature functions, Figure **??**.

```
hypothesis a, hypothesis b
coverage of a != coverage of b  not equal
feature function state s state s of hypothesis a != state s for hypothesis b not equal equal
```

Figure 1. Comparing Hypothesis States

A detailed example of how to add a new feature function to the Moses decoder is contained in **??**.

2.3. cdec and Joshua

cdec and Joshua both have modern feature function frameworks. Both of them have no distinction between stateful and stateless features; stateless features have zero size states.

3. Hypothesis and Translation Rule Evaluation

3.1. Original Framework

The set of feature functions are hardcoded when evaluating the scores of hypotheses and translation rules. Translation rules are evaluated according to Equation **??**:

$$score = \sum_{m \in T} \sum_{i \in m} (\lambda_{m,i} * score_{m,i}) + \sum_{l \in L} (\lambda_l * score_i) + \lambda_{wp} * words \qquad (1)$$

where T is the set of translation models, and L is the set of language models, $\lambda_{m,i}$ is the i'th weight for translation model m, λ_l is the weight for language model l, λ_{wp} is the word penalty weight. Translation rules are evaluated before the search process. The score are used to inform table pruning and cube pruning.

Similarly, hypotheses are evaluated as follows during the search process, Equation **??**:

$$\begin{aligned} score = & \sum_{m \in T} \sum_{i \in m} (\lambda_{m,i} * score_{m,i}) \\ & + \sum_{l \in L} (\lambda_l * score_l) \\ & + \lambda_{wp} * words \\ & + \lambda_d * score_d \\ & + \sum_{m \in R} \sum_{i \in m} (\lambda_{m,i} * score_{m,i}) \\ & + score_{future} \end{aligned} \tag{2}$$

where λ_d is the distortion weight, and R is the set of lexicalized reordering models, and $score_{future}$ is the estimate of the score to completion for the hypothesis.

When adding a new feature function, its scores would not be included in the translation rule or hypothesis scores until the implementation of Equation **??** and Equation **??** are updated.

3.2. Current Framework

The new framework better supports the creation of arbitrary feature functions by generalizing all code that deals with feature functions.

Equation **??** becomes Equation **??**

$$score = \sum_{m \in FF} \sum_{i \in m} (\lambda_{m,i} * score_{m,i}) \tag{3}$$

where FF is the set of all feature functions.

Similarly, Equation **??** becomes Equation **??**:

$$score = \sum_{m \in FF} \sum_{i \in m} (\lambda_{m,i} * score_{m,i}) + score_{future} \tag{4}$$

Furthermore, there are three junctures during decoding where each feature function can be evaluated. At each point, a different range of information is available to the feature function.

1. During creation of translation rules. Has access to the words in the source and target side of translation rule, but not the entire input sentence.
2. Just before decoding. Has access to the entire source sentence to be decoded.

3. During decoding. Access to the context information is possible, containing the current and antecedent hypotheses. The exact information contained in the hypotheses is dependent on the formalism used.

However, the stage at which the feature function is evaluated determines whether the feature function needs to be evaluated, given a different sentence. Scores that are evaluated at stage (1) are only dependent the source phrase, not the entire source sentence, and therefore can be reused inter-sentence. Scores evaluated during stage (2) and (3) must be re-evaluated given a different sentence. Given a simple, source phrase-level cache, only feature function evaluated at stage (1) can be cached.

Additionally, the evaluation at stage (1) can return an estimate of the future cost to guide search. This is a technique used by the phrase-based decoder but not by the SCFG decoder.

Some feature functions do not override any of the stages. These feature is so intertwined with the search mechanism that their scores are set by search mechanism themselves. Features functions such as the translation models, unknown word penalty, generation models fall into this category. The input feature, which scores lattices and confusion network input is also hardcoded into the decoder.

3.3. Examples

Each feature function can override any or all of the methods to evaluate at each of these stage. For example. a simple word penalty feature would typically implement its evaluation in stage (1) as access to the translation rule is sufficient information to allow it to evaluate the word penalty score.

On the other hand, a feature that depends on the bag of words of the input sentence must override the evaluation method for stage (2).

Stateful features need to override stage (3). Features that depend on the segmentation of the input sentence also need to override this stage.

The most efficient way to implement language models is to override stage (1) and (3). As a stateful feature, it must override stage (3) to access adjacent target words to score overlapping n-grams, and to set the current state. However, it is able to evaluate the n-grams within each translation rule without the context, therefore, language models should also override stage (1). The calculation of estimate future cost should also occur in stage (1). This behaviour codify existing practises by building the framework to support it, allowing other feature functions to take advantage of this efficient mechanism.

3.4. Joshua

As with cdec and Moses' SCFG decoder, Joshua's decoding algorithm is an implementation of CKY+ (**?**).

Joshua makes only a single CKY pass over the source language sentence; it does not build a "stateless" hypergraph (or –LM forest, in the language of **?**). To enable this,

Joshua interleaves the extension of partially-constructed rules with the application of complete rules while building the hypergraph, Figure **??**:

spans (i, j) in bottom-up order extend incomplete rules seeking the terminal at $(j-1, j)$ split points k extend incomplete rules over (i, k) seeking a nonterminal N such that chart item (N, k, j) exists in the hypergraph construct new hypergraph node by applying cube pruning to all completed rules over (i, j)

Figure 2. Interweaving of partially-completed and completed rules in Joshua.

The chart containing partially-completed rules is called the *dot chart*, and no pruning is applied to it; rule features are computed only when complete rules are applied.

3.5. cdec

In contrast to Moses and Joshua which use a one-pass search strategy, cdec uses a multi-pass search strategy. Each pass creates a hypergraph that is used as the input for the next pass. cdec evaluates stateless feature functions in the first pass, followed by stateful feature functions in the second.

cdec also supports an arbitrary number of passes, feature functions can be assigned to a particular pass. cdec's multi-pass strategy for feature function evaluation build on its course-to-fine framework.

4. Sparse Feature Function

Sparse features contain scores that may only be present occasionally. However, whereas there are typically 14 scores from five feature functions in a standard phrase-based model and 7 scores in a hierarchical phrase-based model, there may be millions of sparse scores in a system with sparse features. Only a few will be present in a particular hypothesis or translation rule.

Storing and using sparse feature efficiently present a challenge which each decoder has attempted to solve differently.

4.1. Moses

Moses implements supports for both dense and sparse features. Each feature must be identifiable as one or the other. The intuition behind this is that most systems will still contain dense features. Therefore, it would be more efficient to use existing dedicated data-structures to efficiently handle dense vectors. A sparse feature map was added to handle sparse scores.

4.2. cdec

cdec eschews Moses' dense and sparse hybrid approach and implement only sparse maps. Dense features can still be modelled in the framework as sparse features, whose name are suffixed with indices. However, it optimizes sparse feature handling by using different data structures, depending on the number of feature scores.

Also, to improve speed and memory usage, hypotheses only store features deltas with respect to antecedent hypotheses.

4.3. Joshua

As with cdec, Joshua only supports sparse features, however, it uses a different strategy to improve efficiency.

Hypotheses only retain the weighted sum of the feature scores. The individual score components are not stored unless required, for example, in the n-best list. The intuition behind this approach is that the score breakdown is only required in some situations. In most other circumstances, they can be traded for better speed and lower memory consumption.

5. Results

The results from the three decoders are comparable, given the same formalism and model data, and adjusting for scaling differences and constants. Figure **??** shows the comparison of model scores and BLEU scores for a hierarchical model. Moses and cdec are roughly comparable in terms of speed. Also, Moses and cdec converges to roughly the same model scores, while Joshua makes more search errors, leading to overall lower BLEU scores.

Joshua's lower model score is likely due to the fact that in Moses and cdec, the calculation of language model state is delegated to the language model implementation (**?**) which uses the internal data-structure of the language model to determine state. Joshua, on the other hand, uses the words in the left and right context to determine language model state. This results in more aggressive hypothesis recombination in Moses and cdec, leading to better model score. Joshua will soon be updated [1] to incorporate better left-state minimisation to match the performance of Moses and cdec.

6. Conclusion

Statistical machine translation is a dynamic field which is constantly evolving. A major challenge for the Moses project, in common with many long-term software

[1] `https://github.com/hieuhoang/papers/commit/e417a6adcf4f1e49954836b14799509fecdea11c#mt-marathon.2013`

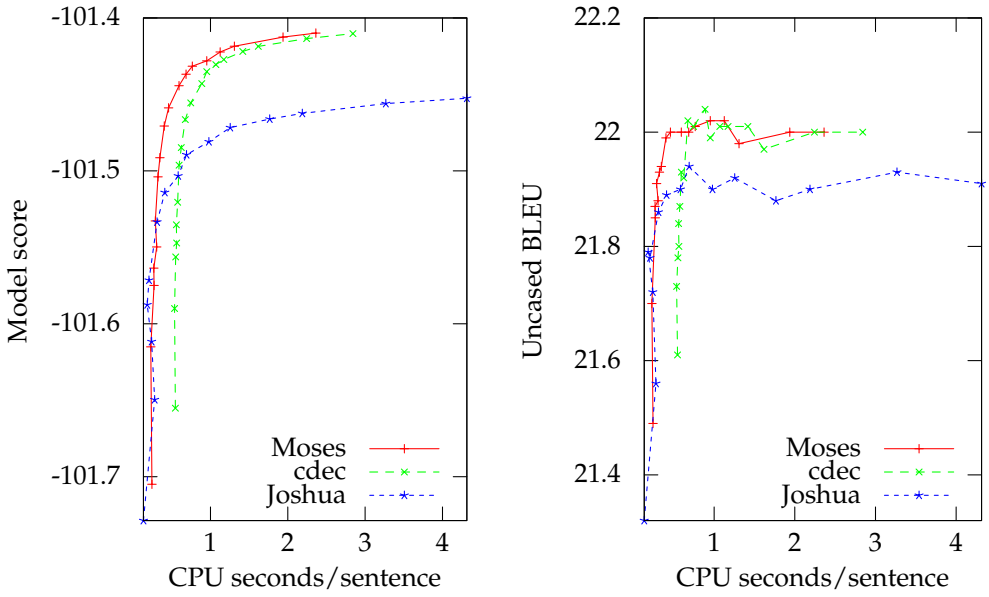

Figure 3. Model scores and BLEU using cube pruning with pop limits ranging from 5 to 1000.

projects, is to maintain and update the the underlying framework and infrastructure to enable this evolution to continue. This paper documents the major changes that were made to Moses to enable this to happen and compare it two other decoders which implements the same decoding formalism.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 288487 (MosesCore).

Appendix: Adding a New Feature Function to Moses, an Example

The section describes how to add a new feature function to the current (July 2013) version of Moses, and use it during decoding.

1. Create a new class in the decoder source code. By convention, the files should be in the folder or subfolder of either
   ```
   moses/FF
   moses/LM
   moses/TranslationModel
   ```
   Inherit the class from either of
   ```
   StatelessFeatureFunction
   StatefulFeatureFunction
   ```
   or any class that inherit from one of these classes. Other common classes to inherit from are:
   ```
   PhraseDictionary
   LanguageModelImplementation
   ```
2. Create a constructor of the form
   ```
   ClassName(const std::string &line);
   ```
3. Implement the method
   ```
   bool IsUseable(const FactorMask &mask) const;
   ```
   The argument is a bit vector of which factors are available. The method should return *true* if it can be evaluated with these factors, otherwise return *false*. If the feature does not use any factors, always return *true*.
   The constructor, and this method, are the only mandatory function for feature functions. All others methods override default methods (that often do nothing).
4. Useful methods to override:
   (a) If the feature needs to load data, override the method
   ```
   void Load();
   ```
   (b) If the feature functions takes arguments
   ```
   void SetParameter(const std::string& key, const std::string& value);
   ```

name, *num-features*, *tuneable* are reserved keys that the new feature should not use.

(c) If it requires initialisation and cleaning after each sentence (this is not particularly thread-safe):

```
void InitializeForInput(InputType const& source);
void CleanUpAfterSentenceProcessing(const InputType& source);
```

5. Override one or more of the Evaluate() method.

```
void Evaluate(const Phrase &source
  , const TargetPhrase &targetPhrase
  , ScoreComponentCollection &scoreBreakdown
  , ScoreComponentCollection &estimatedFutureScore) const;
void Evaluate(const InputType &source
          , ScoreComponentCollection &scoreBreakdown) const;
```

Also for stateless features:

```
void Evaluate(const PhraseBasedFeatureContext& context,
              ScoreComponentCollection* accumulator) const;
void EvaluateChart(const ChartBasedFeatureContext& context,
              ScoreComponentCollection* accumulator) const;
```

The method signature is similar for stateful features, but the methods return a state object:

```
FFState* Evaluate(const Hypothesis& cur_hypo,
  const FFState* prev_state,
  ScoreComponentCollection* accumulator) const;
FFState* EvaluateChart(const ChartHypothesis&,
    int featureID,
    ScoreComponentCollection* accumulator) const;
```

6. Register the new feature function with the decoder by adding the following line to *Factory.cpp*:

```
MOSES_FNAME(ClassName);
```

7. To instantiate the feature during decoding, add the following line to the *[feature]* section of the moses.ini file:

```
ClassName [key=value]*
```

If it has dense features, the weights should be in the *[weight]* section, for example:

```
ClassName0= 0.1 0.2 0.3
```

Address for correspondence:
MISSING first name **MISSING** surname
MISSING `e-mail`
MISSING address