Phrasal Feature API Tutorial
Spence Green // 27 June 2013

Introduction

The Phrasal feature API supports rich feature templates. For most of the last decade "rich features" and "machine translation" seldom appeared in the same sentence. Times have changed. Our recent work (Green et al., 2013) on tuning algorithms for these models showed how to learn "sparse" models with millions of features in less time than it took to build classical "dense" models containing a few features. The new learning procedure permits the style of discriminative feature engineering familiar in other NLP systems such as parsers and taggers. However, the MT search algorithm differs from those systems, and those differences are material for potential feature designs:

1. MT search is approximate
2. Features guide the search heuristics
3. Therefore, "bad" features can cause search errors

Because of (3), awareness of the search procedure is important for MT feature engineering.

Approximate Hypergraph Inference

Phrase-based MT can be cast as a hypergraph inference problem. **Hypergraphs** generalize the idea of a graph by allowing edges to connect two or more nodes. Define a hypergraph $G = (V, E)$, where V is a set of vertices and E is a set of directed hyperedges. Each directed hyperedge is a tuple (T, H), where $T \subseteq V$ is a set of **tail nodes** and $H \subseteq V$ is a set of **head nodes**. For MT, $|H| = 1$, and the resulting hypergraph is called a **B-hypergraph** (Figure 1).

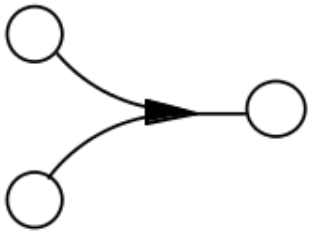

Figure 1: A B-hypergraph with two tail nodes and one head node.

Hypergraph inference can be written as a **deductive system**. Deductive inference involves determination of a new true statement from a set of statements that are assumed to be true. The existing statements are called **antecedents**, from which the procedure deduces the **consequent**. Concretely, phrase-based MT systems combine an existing partial translation with a translation rule to form a new partial translation. In terms of Figure 1, the initial hypothesis and rule would be the tail nodes, and the new hypothesis would be the head node.

Let's define some notation:

- $r = \langle s, t \rangle$: a **translation rule** that translates a source sequence s to a target sequence t.
- R : the set of all such rules. R is conventionally called a **phrase table**.
- $d = \{r_i\}_{i=1...N}$: a **derivation** consisting of N rules.
- $w(\cdot)$: a function that returns the weight of a rule or derivations.
- $cov(d)$: a function that returns a bit vector—called a **coverage set**—indicating which source positions have been translated in the derivation.[1] The expression $r \notin cov(d)$ means that r maps to an empty/uncovered span in d.
- $comb(d, r)$: returns the weight/cost of combining derivation d and rule r.

For an input sequence S, the MT system performs the following procedure:

$\frac{}{r : w(r)}\ r \in R$	axiom
$\frac{d : w(d) \quad r : w(r)}{d' : w(d) \cdot w(r) \cdot comb(d,r)}\ r \notin cov(d)$	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 items are precisely B-hyperedges in the search hypergraph. The MT search space contains exponentially many items, so approximate inference in the form of a **beam search** is typically applied. Phrasal implements a beam-filling algorithm known as **cube pruning**.

The weights are computed via an inner product between a weight vector and a feature map. For example, $w(r) = \theta^\top \Phi(r)$, where θ is the model's weight vector and $\Phi(r)$ is a feature map. The Phrasal feature API specifies that feature map.

Feature Types

In Phrasal, feature extractors are known as featurizers. Featurizer functionality is controlled via Java interfaces. There are two major featurizer interfaces:

Interface	Correspondence
`edu.stanford.nlp.mt.decoder.feat.RuleFeaturizer`	axiom : $w(r)$
`edu.stanford.nlp.mt.decoder.feat.DerivationFeaturizer`	item : $comb(d, r)$

Featurizers return `FeatureValue` objects, which are simply key/value pairs. **Feature values must be monotone**.

[1] MT systems impose the constraint that each source token must be translated exactly once.

RuleFeaturizer

`RuleFeaturizer` is called when a rule r is queried from the phrase table, i.e., for the axiom in Table 1. Feature values are cached with the rule. Therefore, this type of featurizer is appropriate for features that can be extracted independent of context in a derivation.

```
public interface RuleFeaturizer<TK, FV> extends Featurizer<TK,FV> {

  /**
   * This call is made *before* decoding with a rule.
   * Do any setup here.
   *
   * @param featureIndex
   */
  void initialize(Index<String> featureIndex);

  /**
   * Extract and return features for <code>f.rule</code>.
   *
   * @return a list of features or null.
   */
  List<FeatureValue<FV>> ruleFeaturize(Featurizable<TK, FV> f);
}
```

Example 1. `WordPenaltyFeaturizer` returns the cardinality of the target side of each translation rule. These counts are summed for each derivation, and balance the strength of the target language model, which prefers shorter strings.

```
public class WordPenaltyFeaturizer implements
    RuleFeaturizer<IString, String> {

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

    List<FeatureValue<String>> features = Generics.newLinkedList();
    features.add(new FeatureValue<String>("WordPenalty",
                                          f.targetPhrase.size()));
    return features;
  }

  @Override
  public void initialize(Index<String> featureIndex) {}
}
```

DerivationFeaturizer

`DerivationFeaturizer` is called each time a new derivation is created, i.e. for the item in Table 1. It should score the new translation rule in the context of derivation, i.e., generate features that depend on context in the derivation. The canonical example is the language model, which requires some n-gram history for scoring.

```
public interface DerivationFeaturizer<TK, FV>
  extends Featurizer<TK,FV> {

  /**
   * This call is made *before* decoding a new input begins.
   */
  void initialize(int sourceInputId,
                  List<ConcreteRule<TK,FV>> ruleList,
                  Sequence<TK> source,
                  Index<String> featureIndex);

  /**
   * Extract and return a list of features. If features overlap
   * in the list, their values will be added.
   *
   * @return a list of features or null.
   */
  List<FeatureValue<FV>> featurize(Featurizable<TK, FV> f);
}
```

Example 2. The `NGramLanguageModelFeaturizer` scores a derivation based on some fixed-length n-gram history that must be extracted from the partial translation. This history cannot be computed until the derivation is created.

```
public class NGramLanguageModelFeaturizer implements
    DerivationFeaturizer<IString, String>,
    RuleIsolationScoreFeaturizer<IString, String> {

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

    double lmScore = getScore(startPos, limit, f.targetPrefix);

    List<FeatureValue<String>> features = Generics.newLinkedList();
    features.add(new FeatureValue<String>(featureName, lmScore));

    return features;
  }
}
```

Notice that `NGramLanguageModelFeaturizer` also implements the interface `RuleIsolationScoreFeaturizer`, which extends `RuleFeaturizer`. This is because the LM also provides rule scores that are used as part of the search heuristic. However, we want to discard these estimates when the derivation is actually created. `RuleIsolationScoreFeaturizer` prevents the decoder from caching the feature value. Typically this interface is implemented in conjunction with `DerivationFeaturizer`.

Featurizer Concurrency Issues

Phrasal parallelizes decoding by first instantiating a threadpool and then assigning one search instance—called an **inferer**—to each thread. By default, featurizers are not cloned. Therefore, if the featurizer maintains any local state, then it will not be re-entrant. To instruct the decoder to clone one featurizer per inferer, implement the `NeedsCloneable` interface (Table 2)

Additional Interfaces

The feature API provides additional functionality via companion interfaces. The interfaces may or may not specify required methods. These interfaces may only be implemented along with either `RuleFeaturizer` or `DerivationFeaturizer` (or both). Table 2 describes the interfaces.

Interface	Description
`NeedsInternalAlignments`	Loads phrase-internal alignments
`NeedsCloneable`	Required for re-entrancy
`NeedsState`	Permits the decoder to store additional state
`NeedsReorderingRecombination`	Enables reordering recombination history

Table 2: Interfaces that provide additional featurizer functionality. These interfaces are located in the package `edu.stanford.nlp.mt.decoder.feat`.

Here are example feature types for each interface:

- `NeedsInternalAlignments`—suppose that you want to penalize phrase-internal alignment errors such as *maison⇒the*. Alignment to punctuation and function words—a phenomenon known as **garbage collection**—is a common problem in MT.
- `NeedsCloneable`—you want to write source-side features so you load CoreNLP. When initialize() is called, you annotate the source sentence and save it in the featurizer.
- `NeedsState`—you want to implement a target-side syntactic feature that saves some syntactic information about the target prefix.
- `NeedsReorderingRecombination`—you want to implement a new lexicalized reordering model.

Practical Matters

Phrasal loads featurizers by reflection. The featurizer should be specified in the *.ini file that configures the system.[2] Let's look at an example *.ini file:

```
[additional-featurizers]
edu.stanford.nlp.mt.decoder.efeat.DiscriminativePhraseTable(true,true)
edu.stanford.nlp.mt.decoder.feat.LexicalReorderingFeaturizer()

[localprocs]
2

[stack]
800

[weights-file]
phrasal.online.binwts

[n-best-list]
200
```

This file loads two featurizers, the first of which takes two boolean arguments. Constructors with variable-length argument lists are specified as follows:

```
public DiscriminativePhraseTable(String...args) {
  doSource = args.length > 0 ? Boolean.parseBoolean(args[0]) : true;
  doTarget = args.length > 1 ? Boolean.parseBoolean(args[1]) : true;
}
```

The example *.ini file also specifies a few other parameters germane to learning:

- localprocs—the number of decoding threads. We usually set this parameter to the number of *physical* cores on the machine. Hyperthreading offers not performance benefit for Phrasal.
- stack—the beam size of the search procedure. Larger beam sizes allow the search procedure to explore more of the search space. Search is slower, but this may be a worthwhile tradeoff when experimenting with a new feature.
- weights-file—the initial weight vector.
- n-best-list—the size of the ranked translation list generated for each input instance. The learning algorithm benefits from more examples.

[2] The file `$JAVANLP_HOME/projects/mt/scripts-private/README.phrasal` describes the *.ini file format. It also demonstrates how to build and run the system. Each featurizer change requires a full system tuning run, which may last several hours. Think carefully before committing yourself to that endeavor!

How to inspect weight vectors

The learner generates intermediate weight vectors after each epoch. The weight vectors are binary files. To inspect them, run:

```
1 java edu.stanford.nlp.mt.base.IOTools print-wts my.binwts
```

Tips and Tricks

Getting features to work involves tradeoffs among (1) the frequency with which the feature fires, (2) the speed with which the feature can be extracted, and (3) the effect of the feature on search. If the feature fires infrequently, then it is unlikely to affect the corpus-level error metric (BLEU). If it cannot be extracted quickly, then the learning time may be unacceptable. Finally, if the feature causes search errors, then translation quality may fall below the baseline.

Here are a few common problems and associated diagnostics:

- Feature weight set to 0 (does not appear in the weight vector)
  - Increase the beam size
  - Increase the n-best list size
  - Lower the regularization strength
- Search is slow
  - Use primitive arrays instead of HashMaps!
  - Move computation to the initialize() method
- The model overfits to the tuning set (to see the tuning score, `grep` for "BLEU" in the tuning log)
  - Tuning set has multiple references
    - Increase the regularization strength
  - Tuning set has a single reference
    - Try feature filtering (see usage of OnlineTuner.java)
- The tuning score is unstable
  - Are the feature weights monotone? If not, then consider splitting the feature into multiple non-decreasing features.
- Unexplainable `ArrayIndexOutOfBoundsException` or `NullPointerException` when running the featurizer
  - Set localprocs to 1. If the exceptions disappear, then the featurizer is not threadsafe. Implement the `NeedsCloneable` interface.