Designing an Extensible API for Integrating Language Modeling and Realization

Michael White
School of Informatics
University of Edinburgh
Edinburgh EH8 9LW, UK
`http://www.iccs.informatics.ed.ac.uk/~mwhite/`

Abstract

We present an extensible API for integrating language modeling and realization, describing its design and efficient implementation in the OpenCCG surface realizer. With OpenCCG, language models may be used to select realizations with preferred word orders, promote alignment with a conversational partner, avoid repetitive language use, and increase the speed of the best-first anytime search. The API enables a variety of n-gram models to be easily combined and used in conjunction with appropriate edge pruning strategies. The n-gram models may be of any order, operate in reverse ("right-to-left"), and selectively replace certain words with their semantic classes. Factored language models with generalized backoff may also be employed, over words represented as bundles of factors such as form, pitch accent, stem, part of speech, supertag, and semantic class.

1 Introduction

The OpenCCG[1] realizer (White and Baldridge, 2003; White, 2004a; White, 2004c) is an open source surface realizer for Steedman's (2000a; 2000b) Combinatory Categorial Grammar (CCG). It is designed to be the first practical, reusable realizer for CCG, and includes implementations of CCG's unique accounts of coordination and information structure–based prosody.

Like other surface realizers, the OpenCCG realizer takes as input a logical form specifying the propositional meaning of a sentence, and returns one or more surface strings that express this meaning according to the lexicon and grammar. A distinguishing feature of OpenCCG is that it implements a hybrid symbolic-statistical chart realization algorithm that combines (1) a theoretically grounded approach to syntax and semantic composition, with (2) the use of integrated language models for making choices among the options left open by the grammar (thereby reducing the need for hand-crafted rules). In contrast, previous chart realizers (Kay, 1996; Shemtov, 1997; Carroll et al., 1999; Moore, 2002) have not included a statistical component, while previous statistical realizers (Knight and Hatzivassiloglou, 1995; Langkilde, 2000; Bangalore and Rambow, 2000; Langkilde-Geary, 2002; Oh and Rudnicky, 2002; Ratnaparkhi, 2002) have employed less general approaches to semantic representation and composition, and have not typically made use of fine-grained logical forms that include specifications of such information structural notions as theme, rheme and focus.

In this paper, we present OpenCCG's extensible API (application programming interface) for integrating language modeling and realization, describing its design and efficient implementation in Java. With OpenCCG, language models may be used to select realizations with preferred word orders (White, 2004c), promote alignment with a conversational partner (Brockmann et al., 2005), and avoid repetitive language use. In addition,

[1] `http://openccg.sourceforge.net`

by integrating language model scoring into the search, it also becomes possible to use more accurate models to improve realization times, when the realizer is run in anytime mode (White, 2004b).

To allow language models to be combined in flexible ways—as well as to enable research on how to best combine language modeling and realization—OpenCCG's design includes interfaces that allow user-defined functions to be used for scoring partial realizations and for pruning low-scoring ones during the search. The design also includes classes for supporting a range of language models and typical ways of combining them. As we shall see, experience to date indicates that the benefits of employing a highly generalized approach to scoring and pruning can be enjoyed with little or no loss of performance.

The rest of this paper is organized as follows. Section 2 gives an overview of the realizer architecture, highlighting the role of the interfaces for plugging in custom scoring and pruning functions, and illustrating how n-gram scoring affects accuracy and speed. Sections 3 and 4 present OpenCCG's classes for defining scoring and pruning functions, respectively, giving examples of their usage. Finally, Section 5 summarizes the design and concludes with a discussion of future work.

2 Realizer Overview

The UML class diagram in Figure 1 shows the high-level architecture of the OpenCCG realizer; sample Java code for using the realizer appears in Figure 2. A realizer instance is constructed with a reference to a CCG grammar (which supports both parsing and realization). The grammar's lexicon has methods for looking up lexical items via their surface forms (for parsing), or via the principal predicates or relations in their semantics (for realization). A grammar also has a set of hierarchically organized atomic types, which can serve as the values of features in the syntactic categories, or as ontological sorts for the discourse referents in the logical forms (LFs).

Lexical lookup yields lexical signs. A sign pairs a list of words with a category, which itself pairs a syntactic category with a logical form. Lexical signs are combined into derived signs using the rules in the grammar's rule group. Derived signs maintain a derivation history, and their word lists share structure with the word lists of their input signs.

As mentioned in the introduction, for generality, the realizer makes use of a configurable sign scorer and pruning strategy. A sign scorer implements a function that returns a number between 0 and 1 for an input sign. For example, a standard trigram language model can be used to implement a sign scorer, by returning the probability of a sign's words as its score. A pruning strategy implements a method for determining which edges to prune during the realizer's search. The input to the method is a ranked list of edges for signs that have equivalent categories (but different words); grouping edges in this way ensures that pruning cannot "break" the realizer, i.e. prevent it from finding some grammatical derivation when one exists. By default, an N-best pruning strategy is employed, which keeps the N highest scoring input edges, pruning the rest (where N is determined by the current preference settings).

The realization algorithm is implemented by the `realize` method. As in the chart realizers cited earlier, the algorithm makes use of a chart and an agenda to perform a bottom-up dynamic programming search for signs whose LFs completely cover the elementary predications in the input logical form. See Figure 9 (Section 3.1) for a realization trace; the algorithm's details and a worked example appear in (White, 2004a; White, 2004c). The `realize` method returns the edge for the best realization of the input LF, as determined by the sign scorer. After a realization request, the N-best complete edges—or more generally, all the edges for complete realizations that survived pruning—are also available from the chart.

The search for complete realizations proceeds in one of two modes, anytime and two-stage (packing/unpacking). In the anytime mode, a best-first search is performed with a configurable time limit (which may be a limit on how long to look for a better realization, after the first complete one is found). With this mode, the scores assigned by the sign scorer determine the order of the edges on the agenda, and thus have an impact on realization speed. In the two-stage mode, a packed forest

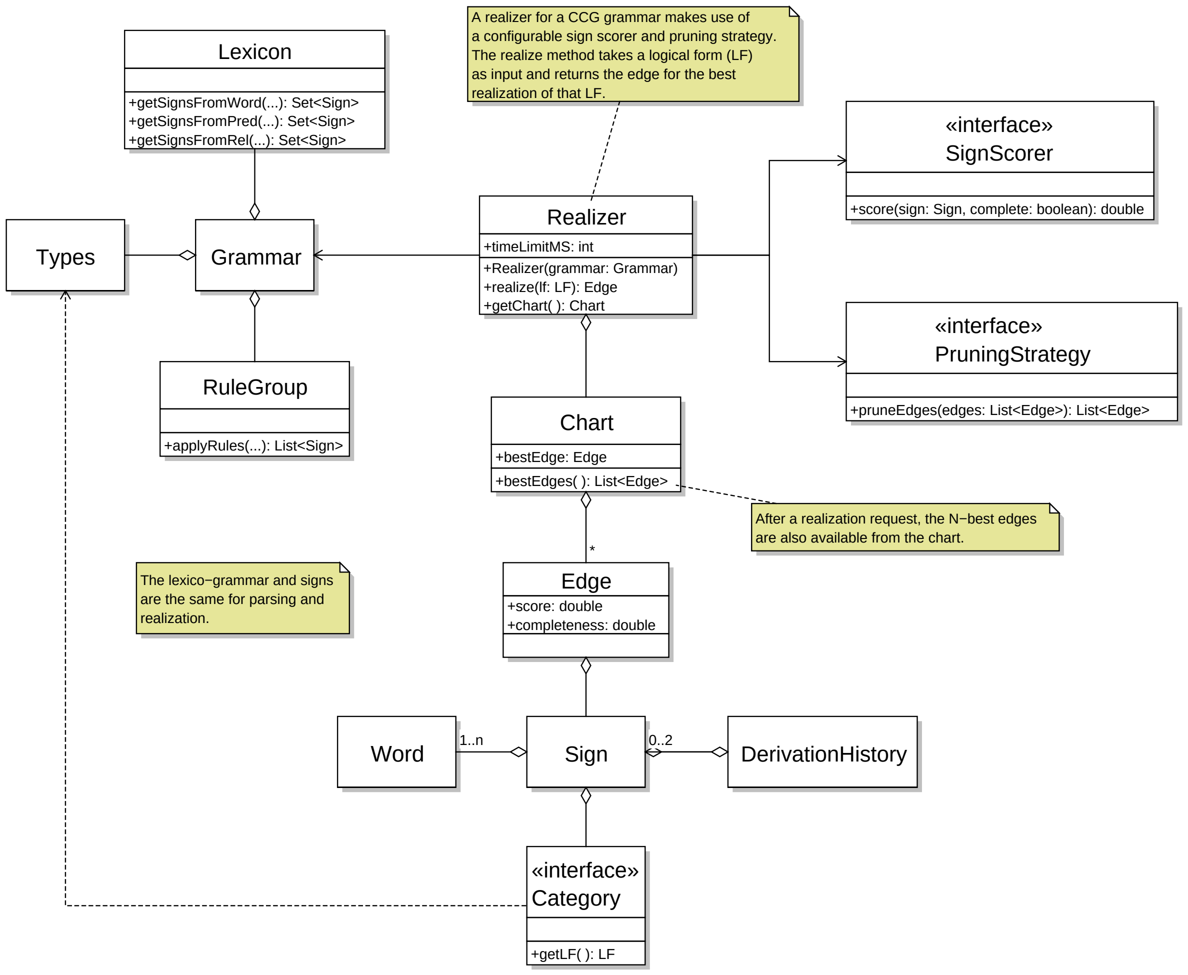
A realizer for a CCG grammar makes use of a configurable sign scorer and pruning strategy. The realize method takes a logical form (LF) as input and returns the edge for the best realization of that LF.
Lexicon
+getSignsFromWord(...): Set<Sign>
+getSignsFromPred(...): Set<Sign>
+getSignsFromRel(...): Set<Sign>
«interface» SignScorer
+score(sign: Sign, complete: boolean): double
Types
Grammar
Realizer
+timeLimitMS: int
+Realizer(grammar: Grammar)
+realize(lf: LF): Edge
+getChart( ): Chart
«interface» PruningStrategy
+pruneEdges(edges: List<Edge>): List<Edge>
RuleGroup
+applyRules(...): List<Sign>
Chart
+bestEdge: Edge
+bestEdges( ): List<Edge>
After a realization request, the N−best edges are also available from the chart.
*
Edge
+score: double
+completeness: double
The lexico−grammar and signs are the same for parsing and realization.
Word
1..n
Sign
0..2
DerivationHistory
«interface» Category
+getLF( ): LF

Figure 1: High-level architecture of the OpenCCG realizer

```
// load grammar, instantiate realizer
URL grammarURL = ...;
Grammar grammar = new Grammar(grammarURL);
Realizer realizer = new Realizer(grammar);

// configure realizer with trigram backoff model
// and 10-best pruning strategy
realizer.signScorer = new StandardNgramModel(3, "lm.3bo");
realizer.pruningStrategy = new NBestPruningStrategy(10);

// ... then, for each request:

// get LF from input XML
Document inputDoc = ...;
LF lf = realizer.getLfFromDoc(inputDoc);

// realize LF and get output words in XML
Edge bestEdge = realizer.realize(lf);
Document outputDoc = bestEdge.sign.getWordsInXml();

// return output
... outputDoc ...;
```

Figure 2: Example realizer usage

of all possible realizations is created in the first stage; then in the second stage, the packed representation is unpacked in bottom-up fashion, with scores assigned to the edge for each sign as it is unpacked, much as in (Langkilde, 2000). In both modes, the pruning strategy is invoked to determine whether to keep or prune newly constructed edges. For single-best output, the anytime mode can provide signficant time savings by cutting off the search early; see (White, 2004c) for discussion. For N-best output—especially when a complete search (up to the edges that survive the pruning strategy) is desirable—the two-stage mode can be more efficient.

To illustrate how n-gram scoring can guide the best-first anytime search towards preferred realizations and reduce realization times, we reproduce in Table 1 and Figures 3 through 5 the cross-validation tests reported in (White, 2004b). In these tests, we measured the realizer's accuracy and speed, under a variety of configurations, on the regression test suites for two small but linguistically rich grammars: the English grammar for the COMIC[2] dialogue system—the core of which is shared with the FLIGHTS system (Moore et al., 2004)—and the Worldcup grammar discussed in (Baldridge, 2002). Table 1 gives the sizes of the test suites. Using these two test suites, we timed how long it took on a 2.2 GHz Linux PC to realize each logical form under each realizer configuration. To measure accuracy, we counted the number of times the best scoring realization exactly matched the target, and also computed a modified version of the Bleu n-gram precision metric (Papineni et al., 2001) employed in machine translation evaluation, using 1- to 4-grams, with the longer n-grams given more weight (cf. Section 3.4). To rank candidate realizations, we used standard n-gram backoff models of orders 2 through 6, with semantic class replacement, as described in Section 3.1. For smoothing, we used Ristad's natural discounting (Ristad, 1995), a parameter-free method that seems to work well with relatively small amounts of data.

To gauge how the amount of training data affects performance, we ran cross-validation tests with increasing numbers of folds, with 25 as the maximum number of folds. We also compared the realization results using the n-gram scorers with two baselines and one topline (oracle method). The first baseline assigns all strings a uniform score of zero, and adds new edges to the end of the agenda, corresponding to breadth-first search. The

[2] http://www.hcrc.ed.ac.uk/comic/

	LF/target	*Unique up*	*Length*			*Input nodes*		
	pairs	*to SC*	*Mean*	*Min*	*Max*	*Mean*	*Min*	*Max*
COMIC	549	219	13.1	6	34	8.4	2	20
Worldcup	276	138	9.2	4	18	6.8	3	13

Table 1: Test suite sizes.

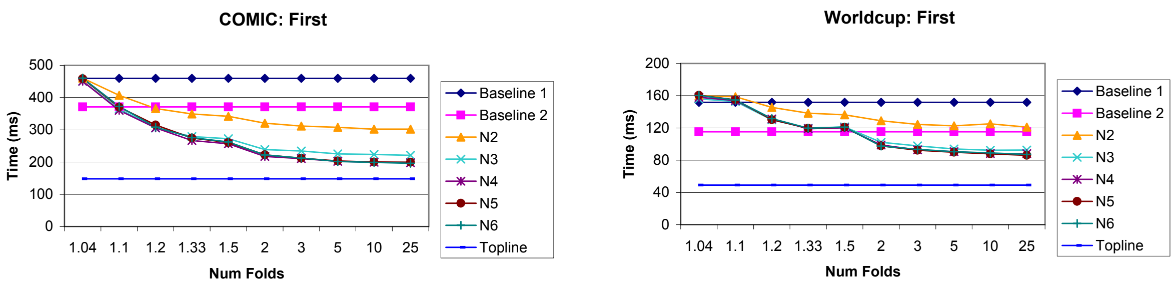

Figure 3: Mean time (in ms.) until first realization is found using n-grams of different orders and Ristad's natural discounting (N), for cross-validation tests with increasing numbers of folds.

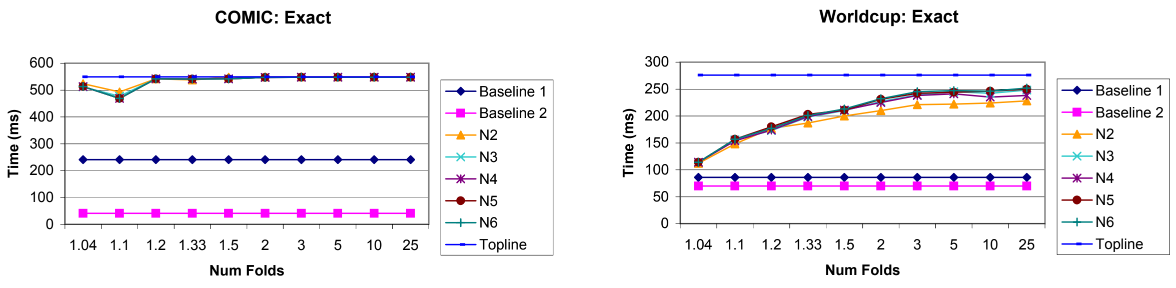

Figure 4: Number of realizations exactly matching target using n-grams of different orders.

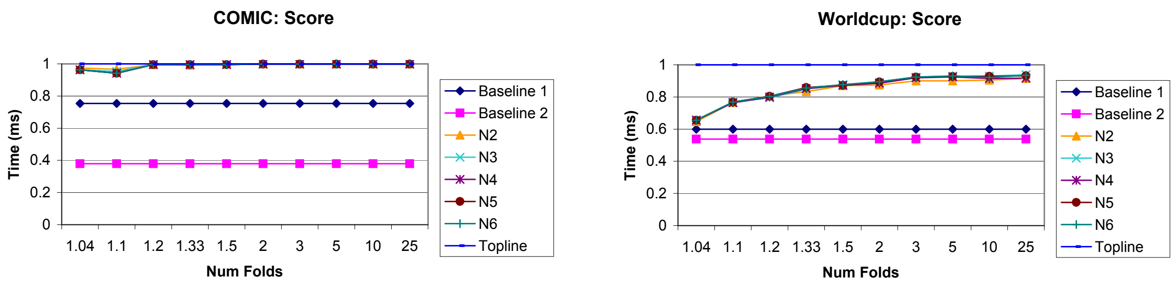

Figure 5: Modified BLEU scores using n-grams of different orders.

second baseline uses the same scorer, but adds new edges at the front of the agenda, corresponding to depth-first search. The topline uses the modified Bleu score, computing n-gram precision against just the target string. With this setup, Figures 3-5 show how initial realization times decrease and accuracy increases when longer n-grams are employed. Figure 3 shows that trigrams offer a substantial speedup over bigrams, while n-grams of orders 4-6 offer a small further improvement. Figures 4 and 5 show that with the COMIC test suite, all n-gram orders work well, while with the Worldcup test suite, n-grams of orders 3-6 offer some improvement over bigrams.

To conclude this section, we note that together with OpenCCG's other efficiency methods, n-gram scoring has helped to achieve realization times adequate for interactive use in both the COMIC and FLIGHTS dialogue systems, along with very high quality. Estimates indicate that n-gram scoring typically accounts for only 2-5% of the time until the best realization is found, while it can more than double realization speed by accurately guiding the best-first anytime search. This experience suggests that more complex scoring models can more than pay for themselves, efficiency-wise, if they yield significantly more accurate preference orders on edges.

3 Classes for Scoring Signs

The classes for implementing sign scorers appear in Figure 6. In the diagram, classes for n-gram scoring appear towards the bottom, while classes for combining scorers appear on the left, and the class for avoiding repetition appears on the right.

3.1 Standard N-gram Models

The `StandardNgramModel` class can load standard n-gram backoff models for scoring, as shown earlier in Figure 2. Such models can be constructed with the SRILM toolkit (Stolcke, 2002), which we have found to be very useful; in principle, other toolkits could be used instead, as long as their output could be converted into the same file formats. Since the SRILM toolkit has more restrictive licensing conditions than those of OpenCCG's LGPL license, OpenCCG includes its own classes for scoring with n-gram models, in order to avoid any necessary runtime dependencies on the SRILM toolkit.

The n-gram tables are efficiently stored in a trie data structure (as in the SRILM toolkit), thereby avoiding any arbitrary limit on the n-gram order. To save memory and speed up equality tests, each string is interned (replaced with a canonical instance) at load time, which accomplishes the same purpose as replacing the strings with integers, but without the need to maintain a separate mapping from integers back to strings. For better generalization, certain words may be dynamically replaced with the names of their semantic classes when looking up n-gram probabilities. Words are assigned to semantic classes in the lexicon, and the semantic classes to use in this way may be configured at the grammar level. Note that (Oh and Rudnicky, 2002) and (Ratnaparkhi, 2002) make similar use of semantic classes in n-gram scoring, by deferring the instantiation of classes (such as *departure city*) until the end of the generation process; our approach accomplishes the same goal in a slightly more flexible way, in that it also allows the specific word to be examined by other scoring models, if desired.

As discussed in (White, 2004c), with dialogue systems like COMIC n-gram models can do an excellent job of placing underconstrained adjectival and adverbial modifiers—as well as boundary tones—without resorting to the more complex methods investigated for adjective ordering in (Shaw and Hatzivassiloglou, 1999; Malouf, 2000). For instance, in examples like those in (1), they correctly select the preferred positions for *here* and *also* (as well as for the boundary tones), with respect to the verbal head and sister dependents:

(1) a. Here$_{L+H*}$ LH% we have a design in the classic$_{H*}$ style LL% .

b. This$_{L+H*}$ design LH% here$_{L+H*}$ LH% is also$_{H*}$ classic LL% .

We have also found that it can be useful to use reverse (or "right-to-left") models, as they can help to place adverbs like *though*, as in (2):

(2) The tiles are also$_{H*}$ from the Jazz$_{H*}$ series though LL% .

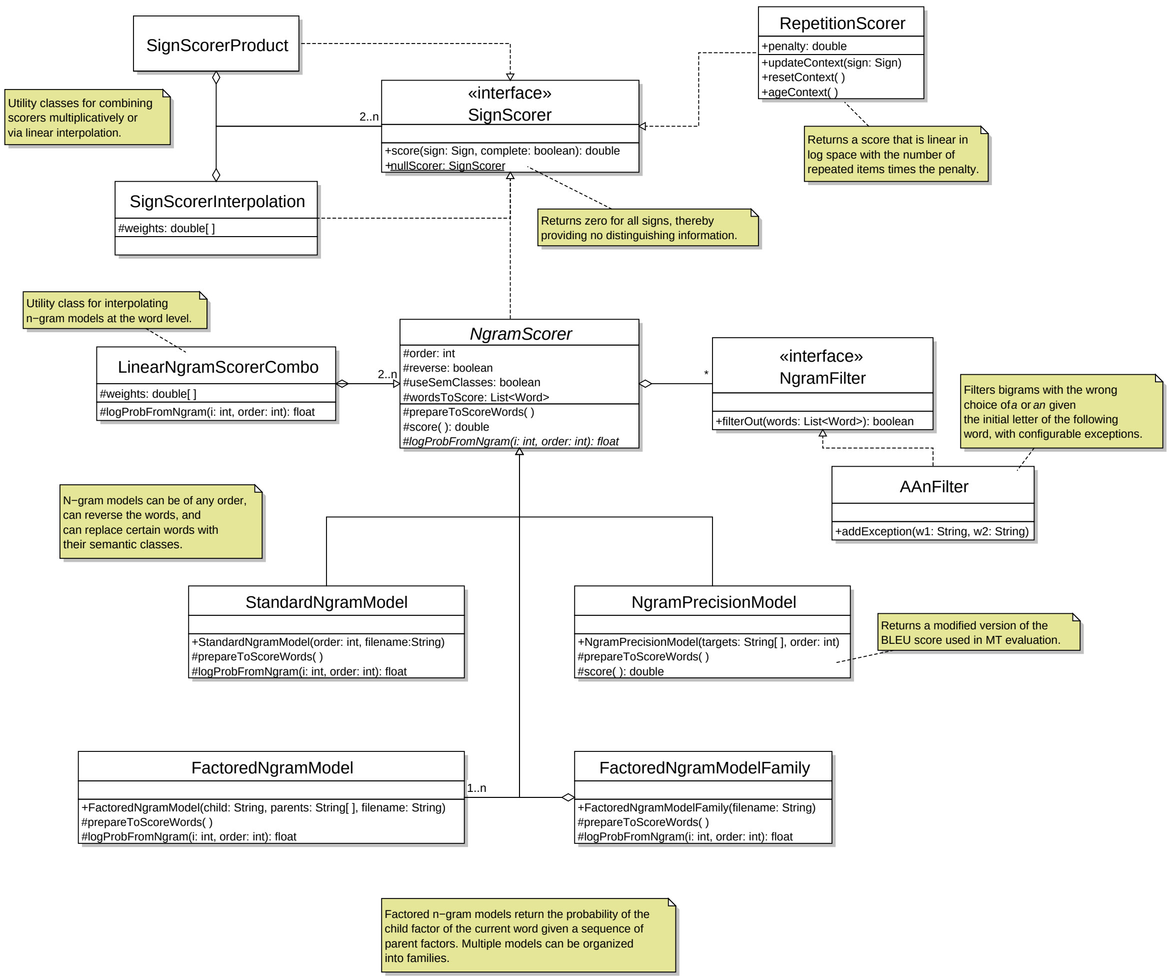
SignScorerProduct
Utility classes for combining scorers multiplicatively or via linear interpolation.
«interface» SignScorer
+score(sign: Sign, complete: boolean): double
+nullScorer: SignScorer
2..n
RepetitionScorer
+penalty: double
+updateContext(sign: Sign)
+resetContext( )
+ageContext( )
Returns a score that is linear in log space with the number of repeated items times the penalty.
SignScorerInterpolation
#weights: double[ ]
Returns zero for all signs, thereby providing no distinguishing information.
Utility class for interpolating n−gram models at the word level.
LinearNgramScorerCombo
#weights: double[ ]
#logProbFromNgram(i: int, order: int): float
2..n
NgramScorer
#order: int
#reverse: boolean
#useSemClasses: boolean
#wordsToScore: List<Word>
#prepareToScoreWords( )
#score( ): double
#logProbFromNgram(i: int, order: int): float
*
«interface» NgramFilter
+filterOut(words: List<Word>): boolean
Filters bigrams with the wrong choice of a or an given the initial letter of the following word, with configurable exceptions.
AAnFilter
+addException(w1: String, w2: String)
N−gram models can be of any order, can reverse the words, and can replace certain words with their semantic classes.
StandardNgramModel
+StandardNgramModel(order: int, filename:String)
#prepareToScoreWords( )
#logProbFromNgram(i: int, order: int): float
NgramPrecisionModel
+NgramPrecisionModel(targets: String[ ], order: int)
#prepareToScoreWords( )
#score( ): double
Returns a modified version of the BLEU score used in MT evaluation.
FactoredNgramModel
+FactoredNgramModel(child: String, parents: String[ ], filename: String)
#prepareToScoreWords( )
#logProbFromNgram(i: int, order: int): float
1..n
FactoredNgramModelFamily
+FactoredNgramModelFamily(filename: String)
#prepareToScoreWords( )
#logProbFromNgram(i: int, order: int): float
Factored n−gram models return the probability of the child factor of the current word given a sequence of parent factors. Multiple models can be organized into families.

Figure 6: Classes for scoring signs

In principle, the forward and reverse probabilities should be the same—as they are both derived via the chain rule from the same joint probability of the words in the sequence—but we have found that with sparse data the estimates can differ substantially. In particular, since *though* typically appears at the end of a variety of clauses, its right context is much more predictable than its left context, and thus reverse models yield more accurate estimates of its likelihood of appearing clause-finally. To illustrate, Figures 7 and 8 show the forward and reverse trigram probabilities for two competing realizations of (2) in a 2-fold cross-validation test (i.e. with models trained on the half of the test suite not including this example). With the forward trigram model, since *though* has not been observed following *series*, and since *series* is a frequently occurring word, the penalty for backing off to the unigram probability for *though* is high, and thus the probability is quite low. The medial placement (following $also_{H*}$) also yields a low probability, but not as low as the clause-final one, and thus the forward model ends up preferring the medial placement, which is quite awkward. By contrast, the reverse model yields a very clear preference for the clause-final position of *though*, and for this reason interpolating the forward and reverse models (see Section 3.3) also yields the desired preference order.

Figure 9 shows a trace of realizing (2) with such an interpolated model. In the trace, the interpolated model is loaded by the class `MyEvenScorer`. The input LF appears at the top. It is flattened into a list of elementary predications, so that coverage of these predications can be tracked using bit vectors. The LF chunks ensure that the subtrees under `h1` and `s1` are realized as independent subproblems; cf. (White, 2004a) for discussion. The edges produced by lexical lookup and instantiation appear next, under the heading `Initial Edges`, with only the edges for $also_{H*}$ and *though* shown in the figure. For each edge, the coverage percentage and score (here a probability) appear first, followed by the word(s) and the coverage vector, then the syntactic category (with features suppressed), and finally any active LF chunks. The edges added to the chart appear (unsorted) under the heading `All Edges`. As this trace shows, in the best-first search, high probability phrases such as *the tiles are* $also_{H*}$ can be added to the chart before low-frequency words such as *though* have even left the agenda. The first complete realization, corresponding to (2), also turns out to be the best one here. As noted in the figure, complete realizations are scored with sentence delimiters, which—by changing the contexts of the initial and final words—can result in a complete realization having a higher probability than its input partial realizations (see next section for discussion). One way to achieve more monotonic scores—and thus more efficient search, in principle—could be to include sentence delimiters in the grammar; we leave this question for future work.

3.2 N-gram Scorers

The `StandardNgramModel` class is implemented as a subclass of the base class `NgramScorer`. All `NgramScorer` instances may have any number of `NgramFilter` instances, whose `filterOut` methods are invoked prior to n-gram scoring; if any of these methods return true, a score of zero is immediately returned. The `AAnFilter` provides one concrete implementation of the `NgramFilter` interface, and returns true if it finds a bigram consisting of *a* followed by a vowel-inital word, or *an* followed by a consonant-initial word, subject to a configurable set of exceptions that can be culled from bigram counts. We have found that such n-gram filters can be more efficient, and more reliable, than relying on n-gram scores alone; in particular, with *a/an*, since the unigram probability for *a* tends to be much higher than that of *an*, with unseen words beginning with a vowel, there may not be a clear preference for the bigram beginning with *an*.

The base class `NgramScorer` implements the bulk of the `score` method, using an abstract `logProbFromNgram` method for subclass-specific calculation of the log probabilities (with backoff) for individual n-grams. The `score` method also invokes the `prepareToScoreWords` method, in order to allow for subclass-specific pre-processing of the words in the given sign. With `StandardNgramModel`, this method is used to extract the word forms or semantic classes into a list of strings to score. It also appends any pitch accents to the

```
the tiles are also_H* from the SERIES_H* series though LL% .
    p( the | <s> )  = [2gram] 0.0999418 [ -1.00025 ]
    p( tiles | the ...)  = [3gram] 0.781102 [ -0.107292 ]
    p( are | tiles ...)  = [3gram] 0.484184 [ -0.31499 ]
    p( also_H* | are ...)  = [3gram] 0.255259 [ -0.593018 ]
    p( from | also_H* ...)  = [3gram] 0.0649038 [ -1.18773 ]
    p( the | from ...)  = [3gram] 0.5 [ -0.30103 ]
    p( SERIES_H* | the ...)  = [3gram] 0.713421 [ -0.146654 ]
    p( series | SERIES_H* ...)  = [3gram] 0.486827 [ -0.312626 ]
    p( though | series ...)  = [1gram] 1.58885e-06 [ -5.79892 ]
    p( LL% | though ...)  = [2gram] 0.416667 [ -0.380211 ]
    p( . | LL% ...)  = [3gram] 0.75 [ -0.124939 ]
    p( </s> | . ...)  = [3gram] 0.999977 [ -1.00831e-05 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -10.2677 ppl= 7.17198 ppl1= 8.57876

the tiles are also_H* though from the SERIES_H* series LL% .
    p( the | <s> )  = [2gram] 0.0999418 [ -1.00025 ]
    p( tiles | the ...)  = [3gram] 0.781102 [ -0.107292 ]
    p( are | tiles ...)  = [3gram] 0.484184 [ -0.31499 ]
    p( also_H* | are ...)  = [3gram] 0.255259 [ -0.593018 ]
    p( though | also_H* ...)  = [1gram] 1.11549e-05 [ -4.95254 ]
    p( from | though ...)  = [1gram] 0.00805451 [ -2.09396 ]
    p( the | from ...)  = [2gram] 0.509864 [ -0.292545 ]
    p( SERIES_H* | the ...)  = [3gram] 0.713421 [ -0.146654 ]
    p( series | SERIES_H* ...)  = [3gram] 0.486827 [ -0.312626 ]
    p( LL% | series ...)  = [3gram] 0.997543 [ -0.00106838 ]
    p( . | LL% ...)  = [3gram] 0.733867 [ -0.134383 ]
    p( </s> | . ...)  = [3gram] 0.999977 [ -1.00831e-05 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -9.94934 ppl= 6.74701 ppl1= 8.02574
```

Figure 7: Forward probabilities for two placements of *though* (COMIC test suite, 2-fold cross validation)

```
the tiles are also_H* from the SERIES_H* series though LL% .
    p( . | <s> )  = [2gram] 0.842366 [ -0.0744994 ]
    p( LL% | . ...)  = [3gram] 0.99653 [ -0.00150975 ]
    p( though | LL% ...)  = [3gram] 0.00677446 [ -2.16913 ]
    p( series | though ...)  = [1gram] 0.00410806 [ -2.38636 ]
    p( SERIES_H* | series ...)  = [2gram] 0.733867 [ -0.134383 ]
    p( the | SERIES_H* ...)  = [3gram] 0.744485 [ -0.128144 ]
    p( from | the ...)  = [3gram] 0.765013 [ -0.116331 ]
    p( also_H* | from ...)  = [3gram] 0.0216188 [ -1.66517 ]
    p( are | also_H* ...)  = [3gram] 0.5 [ -0.30103 ]
    p( tiles | are ...)  = [3gram] 0.432079 [ -0.364437 ]
    p( the | tiles ...)  = [3gram] 0.9462 [ -0.0240173 ]
    p( </s> | the ...)  = [3gram] 0.618626 [ -0.208572 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -7.57358 ppl= 4.27692 ppl1= 4.88098

the tiles are also_H* though from the SERIES_H* series LL% .
    p( . | <s> )  = [2gram] 0.842366 [ -0.0744994 ]
    p( LL% | . ...)  = [3gram] 0.99653 [ -0.00150975 ]
    p( series | LL% ...)  = [3gram] 0.0948425 [ -1.023 ]
    p( SERIES_H* | series ...)  = [3gram] 0.733867 [ -0.134383 ]
    p( the | SERIES_H* ...)  = [3gram] 0.744485 [ -0.128144 ]
    p( from | the ...)  = [3gram] 0.765013 [ -0.116331 ]
    p( though | from ...)  = [1gram] 3.50735e-08 [ -7.45502 ]
    p( also_H* | though ...)  = [1gram] 0.00784775 [ -2.10525 ]
    p( are | also_H* ...)  = [2gram] 0.2291 [ -0.639975 ]
    p( tiles | are ...)  = [3gram] 0.432079 [ -0.364437 ]
    p( the | tiles ...)  = [3gram] 0.9462 [ -0.0240173 ]
    p( </s> | the ...)  = [3gram] 0.618626 [ -0.208572 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -12.2751 ppl= 10.5421 ppl1= 13.0594
```

Figure 8: Reverse probabilities for two placements of *though* (COMIC test suite, 2-fold cross validation)

```
Input LF:
@b1:state(be ^ <info>rh ^ <mood>dcl ^ <tense>pres ^ <owner>s ^ <kon>- ^
          <Arg>(t1:phys-obj ^ tile ^ <det>the ^ <num>pl ^ <info>rh ^ <owner>s ^ <kon>-) ^
          <Prop>(h1:proposition ^ has-rel ^ <info>rh ^ <owner>s ^ <kon>- ^
                 <Of>t1:phys-obj ^
                 <Source>(s1:abstraction ^ series ^ <det>the ^ <num>sg ^ <info>rh ^ <owner>s ^ <kon>- ^
                          <HasProp>(j1:series ^ Jazz ^ <kon>+ ^ <info>rh ^ <owner>s))) ^
          <HasProp>(a1:proposition ^ also ^ <kon>+ ^ <info>rh ^ <owner>s) ^
          <HasProp>(t2:proposition ^ though ^ <info>rh ^ <owner>s ^ <kon>-))

Instantiating scorer from class: MyEvenScorer

Preds:
ep[0]:  @a1:proposition(also)
ep[1]:  @a1:proposition(<info>rh)
ep[2]:  @a1:proposition(<kon>+)
ep[3]:  @a1:proposition(<owner>s)
ep[4]:  @b1:state(be)
ep[5]:  @b1:state(<info>rh)
...

LF chunks:
chunk[0]:  {14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
chunk[1]:  {20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}

Initial Edges:
{0.12} [0.011] also_H* {0, 1, 2, 3, 12} :- s\.s
{0.12} [0.011] also_H* {0, 1, 2, 3, 12} :- s\np/^(s\np)
{0.12} [0.011] also_H* {0, 1, 2, 3, 12} :- s/^s
...
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\np/^(s\np)
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\.s
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s/^s
...

Uninstantiated Semantically Null Edges:
{0.00} [0.073] LL% {} :- s$1\*(s$1)
{0.00} [0.011] L {} :- s$1\*(s$1)

All Edges:
{0.02} [0.059] . {7} :- sent\*s
{0.02} [0.059] . {7} :- sent\*(s\np)
{0.02} [0.052] the {25} :- np/^n  < 0 1 >
{0.02} [0.052] the {32} :- np/^n
{0.12} [0.032] tiles {31, 33, 34, 35, 36} :- n
{0.02} [0.018] from {19} :- n\n/<np  < 0 >
{0.15} [0.018] from {14, 15, 16, 17, 18, 19} :- s\!np/<np  < 0 >
{0.17} [0.017] is {4, 5, 6, 8, 9, 10, 11} :- s\np/(s\!np)
...
{0.15} [0.009] the tiles {31, 32, 33, 34, 35, 36} :- np
{0.15} [0.009] the tiles {31, 32, 33, 34, 35, 36} :- s/@i(s\@inp)
{0.15} [0.009] the tiles {31, 32, 33, 34, 35, 36} :- s$1\@i(s$1/@inp)
...
{0.44} [0.001] the tiles are also_H* {0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 31, 32, 33, 34, 35, 36} :- s/(s\!np)
...
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\np/^(s\np)
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\.s
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s/^s
...
{0.85} [1.32E-5] the tiles are also_H* from the Jazz_H* series {...} :- s
{0.85} [1.32E-5] the tiles are also_H* from the Jazz_H* series LL% {...} :- s
...
{0.24} [2.64E-6] also_H* though {0, 1, 2, 3, 12, 13, 37, 38, 39, 40} :- s\np/^(s\np)
{0.24} [2.64E-6] also_H* though {0, 1, 2, 3, 12, 13, 37, 38, 39, 40} :- s\.s
...
{0.88} [5.44E-8] the tiles are from the Jazz_H* series though LL% . {...} :- sent
...
{0.85} [4.85E-8] the tiles are from the Jazz_H* series also_H* {...} :- s
...
{0.88} [3.14E-9] the tiles also_H* are from the Jazz_H* series LL% . {...} :- sent
...
{0.98} [2.96E-9] the tiles are also_H* from the Jazz_H* series though {...} :- s
...
{0.98} [1.51E-9] the tiles are also_H* from the Jazz_H* series though LL% {...} :- s
{1.00} [1.34E-8] the tiles are also_H* from the Jazz_H* series though LL% . {...} :- sent

     ****** first complete realization; scored with <s> and </s> tags ******

...
{0.24} [1.44E-9] also_H* though L {...} :- s\np/^(s\np)
...
{0.56} [2E-10] though the tiles are also_H* LL% {...} :- s/(s\!np)
...

Complete Edges (sorted):
{1.00} [1.34E-8] the tiles are also_H* from the Jazz_H* series though LL% . {...} :- sent
{1.00} [1.33E-8] the tiles are also_H* from the Jazz_H* series LL% though LL% . {...} :- sent
...
```

Figure 9: Realizer trace for example (2) with interpolated model

word forms or semantic classes, effectively treating them as integral parts of the words.

Since the realizer builds up partial realizations bottom-up rather than left-to-right, it only adds start of sentence (and end of sentence) tags with complete realizations. As a consequence, the words with less than a full $n-1$ words of history are scored with appropriate sub-models. For example, the first word of a phrase is scored with a unigram sub-model, without imposing backoff penalties.

Another consequence of bottom-up realization is that both the left- and right-contexts may change when forming new signs from a given input sign. Consequently, it is often not possible (even in principle) to use the score of an input sign directly in computing the score of a new result sign. If one could make assumptions about how the score of an input sign has been computed—e.g., by a bigram model—one could determine the score of the result sign from the scores of the input signs together with an adjustment for the word(s) whose context has changed. However, our general approach to sign scoring precludes making such assumptions. Nevertheless, it is still possible to improve the efficiency of n-gram scoring by caching the log probability of a sign's words, and then looking up that log probability when the sign is used as the first input sign in creating a new combined sign—thus retaining the same left context—and only recomputing the log probabilities for the words of any input signs past the first one. (With reverse models, the sign must be the last sign in the combination.) In principle, the derivation history could be consulted further to narrow down the words whose n-gram probabilities must be recomputed to the minimum possible, though `NgramScorer` only implements a single-step lookup at present.[3] Finally, note that a Java `WeakHashMap` is used to implement the cache, in order to avoid an undesirable buildup of entries across realization requests.

3.3 Interpolation

Scoring models may be linearly interpolated in two ways. Sign scorers of any variety may be combined using the `SignScorerInterpolation` class. For example, Figure 10 shows how forward and reverse n-gram models may be interpolated.

With n-gram models of the same direction, it is also possible to linearly interpolate models at the word level, using the `LinearNgramScorerCombo` class. Word-level interpolation makes it easier to use cache models created with maximum likelihood estimation, as word-level interpolation with a base model avoids problems with zero probabilities in the cache model. As discussed in (Brockmann et al., 2005), cache models can be used to promote alignment with a conversational partner, by constructing a cache model from the bigrams in the partner's previous turn, and interpolating it with a base model.[4] Figure 11 shows one way to create such an interpolated model.

3.4 N-gram Precision Models

The `NgramPrecisionModel` subclass of `Ngram-Scorer` computes a modified version of the Bleu score used in MT evaluation (Papineni et al., 2001). Its constructor takes as input an array of target strings—from which it extracts the n-gram sequences to use in computing the n-gram precision score—and the desired order. Unlike with the Bleu score, rank order centroid weights (rather than the geometric mean) are used to combine scores of different orders, which avoids problems with scoring partial realizations which have no n-gram matches of the target order. For simplicity, the score also does not include the Bleu score's bells and whistles to make cheating on length difficult.

We have found n-gram precision models to be very useful for regression testing the grammar, as an n-gram precision model created just from the target string nearly always leads the realizer to choose that exact string as its preferred realization. Such models can also be useful for evaluating the success of different scoring models in a cross-validation setup, though with high quality output, manual inspection is usually necessary to determine the importance of any differences between

[3]Informal experiments indicate that caching log probabilities in this way can yield an overall reduction in best-first realization times of 2-3% on average.

[4]At present, such cache models must be constructed with a call to the SRILM toolkit; it would not be difficult to add OpenCCG support for constructing them though, since these models do not require smoothing.

```
// configure realizer with 4-gram forward and reverse backoff models,
// interpolated with equal weight
NgramScorer forwardModel = new StandardNgramModel(4, "lm.4bo");
NgramScorer reverseModel = new StandardNgramModel(4, "lm-r.4bo");
reverseModel.setReverse(true);
realizer.signScorer = new SignScorerInterpolation(
    new SignScorer[] { forwardModel, reverseModel }
);
```

Figure 10: Example interpolated n-gram model

```
// configure realizer with 4-gram backoff base model,
// interpolated at the word level with a bigram maximum-likelihood
// cache model, with more weight given to the base model
NgramScorer baseModel = new StandardNgramModel(4, "lm.4bo");
NgramScorer cacheModel = new StandardNgramModel(2, "lm-cache.mle");
realizer.signScorer = new LinearNgramScorerCombo(
    new SignScorer[] { baseModel, cacheModel },
    new double[] { 0.6, 0.4 }
);
```

Figure 11: Example word-level interpolation of a cache model

the preferred realization and the target string.

3.5 Factored Language Models

A factored language model (Bilmes and Kirchhoff, 2003) is a new kind of language model that treats words as bundles of factors. To support scoring with such models, OpenCCG represents words as objects with a surface form, pitch accent, stem, part of speech, supertag, and semantic class. Words may also have any number of further attributes, such as associated gesture classes, in order to handle in a general way elements like pitch accents that are "coarticulated" with words.

To represent words efficiently, and to speed up equality tests, all attribute values are interned, and the `Word` objects themselves are interned via a factory method. Note that in Java, it is straightforward to intern objects other than strings by employing a `WeakHashMap` to map from an object key to a weak reference to itself as the canonical instance. (Using a weak reference avoids accumulating interned objects that would otherwise be garbage collected.)

With the SRILM toolkit, factored language models can be constructed that support *generalized parallel backoff*: that is, backoff order is not restricted to just dropping the most temporally distant word first, but rather may be specified as a path through the set of contextual parent variables; additionally, parallel backoff paths may be specified, with the possibility of combining these paths dynamically in various ways. In OpenCCG, the `FactoredNgramModel` class supports scoring with factored language models that employ generalized backoff, though parallel backoff is not yet supported, as it remains somewhat unclear whether the added complexity of parallel backoff is worth the implementation effort. Typically, several related factored language models are specified in a single file and loaded by a `FactoredNgramModelFamily`, which can multiplicatively score models for different child variables, and include different sub-models for the same child variable.

To illustrate, let us consider a simplified version of the factored language model family used in the COMIC realizer. This model computes the probability of the current word given the preceding ones according to the formula shown in (3), where a word consists of the factors word (W), pitch accent (A), gesture class (GC), and gesture instance (GI), plus the other standard factors which the model ignores:

$$(3)\quad \begin{array}{l} P(\langle W,A,GC,GI\rangle \mid \langle W,A,GC,GI\rangle_{-1} \ldots) \approx \\ \quad P(W \mid W_{-1}W_{-2}A_{-1}A_{-2}) \times \\ \quad P(GC \mid W) \times \\ \quad P(GI \mid GC) \end{array}$$

In (3), the probability of the current word is approximated by the probability of the current word form given the preceding two word forms and preceding two pitch accents, multiplied by the probability of the current gesture class given the current word form, and by the probability of the current gesture instance given the current gesture class. Note that in the COMIC grammar, the choice of pitch accent is entirely rule governed, so the current pitch accent is not scored separately in the model. However, the preceding pitch accents are taken into account in predicting the current word form, as perplexity experiments have suggested that they do provide additional information beyond that provided by the previous word forms.

The specification file for this model appears in Figure 12. The format of the file is a restricted form of the files used by the SRILM toolkit to build factored language models. The file specifies four models, where the first, third and fourth models correspond to those in (3). With the first model, since the previous words are typically more informative than the previous pitch accents, the backoff order specifies that the most distant accent, `A(-2)`, should be dropped first, followed by the previous accent, `A(-1)`, then the most distant word, `W(-2)`, and finally the previous word, `W(-1)`. The second model is considered a sub-model of the first—since it likewise predicts the current word—to be used when there is only one word of context available (i.e. with bigrams). Note that when scoring a bigram, the second model will take the previous pitch accent into account, whereas the first model would not. For documentation of the file format as it is used in the SRILM toolkit, see (Kirchhoff et al., 2002).

Like `StandardNgramModel`, the `FactoredNgramModel` class stores its n-gram tables in a trie data structure, except that it stores an interned factor key (i.e. a factor name and value pair, or just a string, in the case of the word form) at each node, rather than a simple string. During scoring, the `logProbFromNgram` method determines the log probability (with backoff) of a given n-gram by extracting the appropriate sequence of factor keys, and using them to compute the log probability as with standard n-gram models. The `FactoredNgramModelFamily` class computes log probabilities by delegating to its component factored n-gram models (choosing appropriate sub-models, when appropriate) and summing the results.

3.6 Avoiding Repetition

While cache models appear to be a promising avenue to promote lexical and syntactic alignment with a conversational partner, a different mechanism appears to be called for to avoid "self-alignment"—that is, to avoid the repetitive use of words and phrases. As a means to experiment with avoiding repetition, OpenCCG includes the `RepetitionScorer` class. This class makes use of a configurable penalty plus a set of methods for dynamically managing the context. It returns a score of $10^{-c_r \times p}$, where c_r is the count of repeated items, and p is the penalty. Note that this formula returns 1 if there are no repeated items, and returns a score that is linear in log space with the number of repeated items otherwise.

A repetition scorer can be combined multiplicatively with an n-gram model, in order to discount realizations that repeat items from the recent context. Figure 13 shows such a combination, together with the operations for updating the context. By default, open class stems are the considered the relevant items over which to count repetitions, though this behavior can be specialized by subclassing `RepetitionScorer` and overriding the `updateItems` method. Note that in counting repetitions, full counts are given to items in the previous words or recent context, while fractional counts are given to older items; the exact details may likewise be changed in a subclass, by overriding the `repeatedItems` method.

4 Pruning Strategies

The classes for defining edge pruning strategies appear in Figure 14. As mentioned in Section 2, an N-best pruning strategy is employed by default, where N is determined by the current preference settings. It is also possible to define custom strategies. To support the definition of a certain kind

```
## Simplified COMIC realizer FLM spec file

## Trigram Word model based on previous words and accents, dropping accents first,
##   with bigram sub-model;
## Unigram Gesture Class model based on current word; and
## Unigram Gesture Instance model based on current gesture class

4

## 3gram with A
W : 4 W(-1) W(-2) A(-1) A(-2) w_w1w2a1a2.count w_w1w2a1a2.lm 5
  W1,W2,A1,A2  A2 ndiscount gtmin 1
  W1,W2,A1   A1 ndiscount gtmin 1
  W1,W2  W2 ndiscount gtmin 1
  W1  W1 ndiscount gtmin 1
  0   0  ndiscount gtmin 1

## bigram with A
W : 2 W(-1) A(-1) w_w1a1.count w_w1a1.lm 3
  W1,A1  A1  ndiscount gtmin 1
  W1  W1  ndiscount gtmin 1
  0   0   ndiscount gtmin 1

## Gesture class depends on current word
GC : 1 W(0) gc_w0.count gc_w0.lm 2
  W0  W0 ndiscount gtmin 1
  0   0  ndiscount gtmin 1

## Gesture instance depends only on class
GI : 1 GC(0) gi_gc0.count gi_gc0.lm 2
  GC0  GC0 ndiscount gtmin 1
  0 0
```

Figure 12: Example factored language model family specification

```
// set up n-gram scorer and repetition scorer
String lmfile = "ngrams/combined.flm";
boolean semClasses = true;
NgramScorer ngramScorer = new FactoredNgramModelFamily(lmfile, semClasses);
ngramScorer.addFilter(new AAnFilter());
RepetitionScorer repetitionScorer = new RepetitionScorer();

// combine n-gram scorer with repetition scorer
realizer.signScorer = new SignScorerProduct(
    new SignScorer[] { ngramScorer, repetitionScorer }
);

// ... then, after each realization request,
Edge bestEdge = realizer.realize(lf);

// ... update repetition context for next realization:
repetitionScorer.ageContext();
repetitionScorer.updateContext(bestEdge.getSign());
```

Figure 13: Example combination of an n-gram scorer and a repetition scorer

of custom strategy, the abstract class `DiversityPruningStrategy` provides an N-best pruning strategy that promotes diversity in the edges that are kept, according to the equivalence relation established by the abstract `notCompellinglyDifferent` method. In particular, in order to determine which edges to keep, a diversity pruning strategy clusters the edges into a ranked list of equivalence classes, which are sequentially sampled until the limit N is reached. If the `singleBestPerGroup` flag is set, then a maximum of one edge per equivalence class is retained.

As an example, the COMIC realizer's diversity pruning strategy appears in Figure 15. The idea behind this strategy is to avoid having the N-best lists become full of signs whose words differ only in the exact gesture instance associated with one or more of the words. With this strategy, if two signs differ in just this way, the edge for the lower-scoring sign will be considered "not compellingly different" and pruned from the N-best list, making way for other edges whose signs exhibit more interesting differences.

OpenCCG also provides a concrete subclass of `DiversityPruningStrategy` named `NgramDiversityPruningStrategy`, which generalizes the approach to pruning described in (Langkilde, 2000). With this class, two signs are considered not compellingly different if they share the same $n-1$ initial and final words, where n is the n-gram order. When one is interested in single-best output, an n-gram diversity pruning strategy can increase efficiency while guaranteeing no loss in quality—as long as the reduction in the search space outweighs the extra time necessary to check for the same initial and final words—since any words in between an input sign's $n-1$ initial and final ones cannot affect the n-gram score of a new sign formed from the input sign. However, when N-best outputs are desired, or when repetition scoring is employed, it is less clear whether it makes sense to use an n-gram diversity pruning strategy; for this reason, a simple N-best strategy remains the default option.

5 Conclusions and Future Work

In this paper, we have presented OpenCCG's extensible API for efficiently integrating language modeling and realization, in order to select realizations with preferred word orders, promote alignment with a conversational partner, avoid repetitive language use, and increase the speed of the best-first anytime search. As we have shown, the design enables a variety of n-gram models to be easily combined and used in conjunction with appropriate edge pruning strategies. The n-gram models may be of any order, operate in reverse ("right-to-left"), and selectively replace certain words with their semantic classes. Factored language models with generalized backoff may also be employed, over words represented as bundles of factors such as form, pitch accent, stem, part of speech, supertag, and semantic class.

In future work, we plan to further explore how to best employ factored language models; in particular, inspired by (Bangalore and Rambow, 2000), we plan to examine whether factored language models using supertags can provide an effective way to combine syntactic and lexical probabilities. We also plan to implement the capability to use `one-of` alternations in the input logical forms (Foster and White, 2004), in order to more efficiently defer lexical choice decisions to the language models.

Acknowledgements

Thanks to Jason Baldridge, Carsten Brockmann, Mary Ellen Foster, Philipp Koehn, Geert-Jan Kruijff, Johanna Moore, Jon Oberlander, Miles Osborne, Mark Steedman, Sebastian Varges and the anonymous reviewers for helpful discussion.

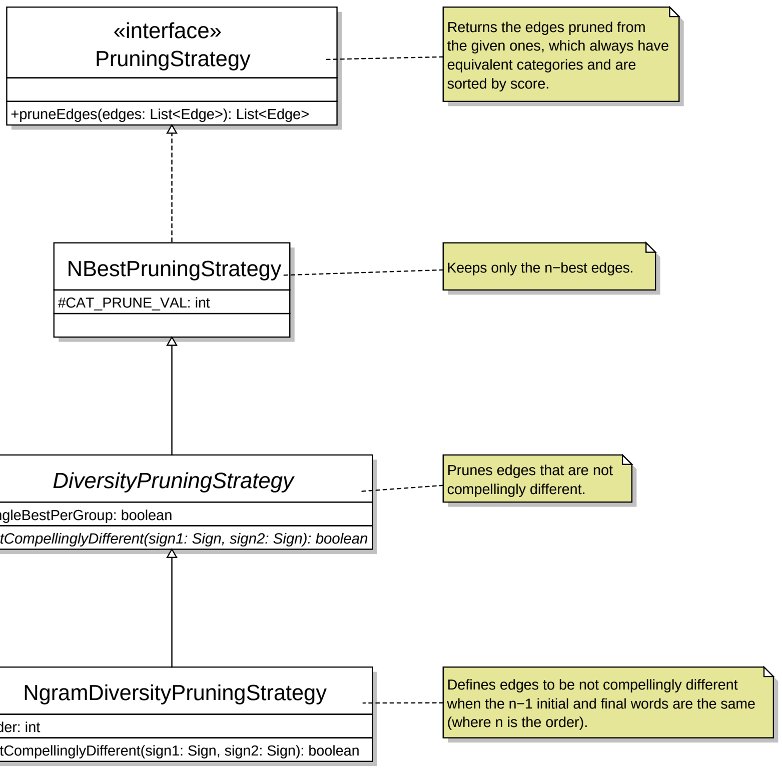
«interface»
PruningStrategy
+pruneEdges(edges: List<Edge>): List<Edge>
Returns the edges pruned from the given ones, which always have equivalent categories and are sorted by score.
NBestPruningStrategy
#CAT_PRUNE_VAL: int
Keeps only the n−best edges.
DiversityPruningStrategy
+singleBestPerGroup: boolean
+notCompellinglyDifferent(sign1: Sign, sign2: Sign): boolean
Prunes edges that are not compellingly different.
NgramDiversityPruningStrategy
#order: int
+notCompellinglyDifferent(sign1: Sign, sign2: Sign): boolean
Defines edges to be not compellingly different when the n−1 initial and final words are the same (where n is the order).

Figure 14: Classes for defining pruning strategies

```
// configure realizer with gesture diversity pruner
realizer.pruningStrategy = new DiversityPruningStrategy() {
    /**
     * Returns true iff the given signs are not compellingly different;
     * in particular, returns true iff the words differ only in their
     * gesture instances. */
    public boolean notCompellinglyDifferent(Sign sign1, Sign sign2) {
        List words1 = sign1.getWords(); List words2 = sign2.getWords();
        if (words1.size() != words2.size()) return false;
        for (int i = 0; i < words1.size(); i++) {
            Word w1 = (Word) words1.get(i); Word w2 = (Word) words2.get(i);
            if (w1 == w2) continue;
            if (w1.getForm() != w2.getForm()) return false;
            if (w1.getPitchAccent() != w2.getPitchAccent()) return false;
            if (w1.getVal("GC") != w2.getVal("GC")) return false;
            // nb: assuming that they differ in the val of GI at this point
        }
        return true;
    }
};
```

Figure 15: Example diversity pruning strategy

References

Jason Baldridge. 2002. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. Ph.D. thesis, School of Informatics, University of Edinburgh.

Srinivas Bangalore and Owen Rambow. 2000. Exploiting a probabilistic hierarchical model for generation. In *Proc. COLING-00*.

Jeff Bilmes and Katrin Kirchhoff. 2003. Factored language models and general parallelized backoff. In *Proc. HLT-03*.

Carsten Brockmann, Amy Isard, Jon Oberlander, and Michael White. 2005. Variable alignment in affective dialogue. In *Proc. UM-05 Workshop on Affective Dialogue Systems*. To appear.

John Carroll, Ann Copestake, Dan Flickinger, and Victor Poznański. 1999. An efficient chart generator for (semi-) lexicalist grammars. In *Proc. EWNLG-99*.

Mary Ellen Foster and Michael White. 2004. Techniques for Text Planning with XSLT. In *Proc. 4th NLPXML Workshop*.

Martin Kay. 1996. Chart generation. In *Proc. ACL-96*.

Katrin Kirchhoff, Jeff Bilmes, Sourin Das, Nicolae Duta, Melissa Egan, Gang Ji, Feng He, John Henderson, Daben Liu, Mohamed Noamany, Pat Schone, Richard Schwartz, and Dimitra Vergyri. 2002. Novel Approaches to Arabic Speech Recognition: Report from the 2002 Johns-Hopkins Summer Workshop.

Kevin Knight and Vasileios Hatzivassiloglou. 1995. Two-level, many-paths generation. In *Proc. ACL-95*.

Irene Langkilde-Geary. 2002. An empirical verification of coverage and correctness for a general-purpose sentence generator. In *Proc. INLG-02*.

Irene Langkilde. 2000. Forest-based statistical sentence generation. In *Proc. NAACL-00*.

Robert Malouf. 2000. The order of prenominal adjectives in natural language generation. In *Proc. ACL-00*.

Johanna Moore, Mary Ellen Foster, Oliver Lemon, and Michael White. 2004. Generating tailored, comparative descriptions in spoken dialogue. In *Proc. FLAIRS-04*.

Robert C. Moore. 2002. A complete, efficient sentence-realization algorithm for unification grammar. In *Proc. INLG-02*.

Alice H. Oh and Alexander I. Rudnicky. 2002. Stochastic natural language generation for spoken dialog systems. *Computer, Speech & Language*, 16(3/4):387–407.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2001. Bleu: a Method for Automatic Evaluation of Machine Translation. Technical Report RC22176, IBM.

Adwait Ratnaparkhi. 2002. Trainable approaches to surface natural language generation and their application to conversational dialog systems. *Computer, Speech & Language*, 16(3/4):435–455.

Eric S. Ristad. 1995. A Natural Law of Succession. Technical Report CS-TR-495-95, Princeton Univ.

James Shaw and Vasileios Hatzivassiloglou. 1999. Ordering among premodifiers. In *Proc. ACL-99*.

Hadar Shemtov. 1997. *Ambiguity Management in Natural Language Generation*. Ph.D. thesis, Stanford University.

Mark Steedman. 2000a. Information structure and the syntax-phonology interface. *Linguistic Inquiry*, 31(4):649–689.

Mark Steedman. 2000b. *The Syntactic Process*. MIT Press.

Andreas Stolcke. 2002. SRILM — An extensible language modeling toolkit. In *Proc. ICSLP-02*.

Michael White and Jason Baldridge. 2003. Adapting Chart Realization to CCG. In *Proc. EWNLG-03*.

Michael White. 2004a. Efficient Realization of Coordinate Structures in Combinatory Categorial Grammar. *Research on Language and Computation*. To appear.

Michael White. 2004b. Experiments with multimodal output in human-machine interaction. IST Project COMIC Public Deliverable 7.4.

Michael White. 2004c. Reining in CCG Chart Realization. In *Proc. INLG-04*.