Inducing Clause-Combining Rules: A Case Study with the SPaRKy Restaurant Corpus

Michael White
Department of Linguistics
The Ohio State University
Columbus, Ohio 43210, USA
mwhite@ling.ohio-state.edu

David M. Howcroft
Computational Linguistics and Phonetics
Saarland University
66123 Saarbrücken, Germany
howcroft@coli.uni-saarland.de

Abstract

We describe an algorithm for inducing clause-combining rules for use in a traditional natural language generation architecture. An experiment pairing lexicalized text plans from the SPaRKy Restaurant Corpus with logical forms obtained by parsing the corresponding sentences demonstrates that the approach is able to learn clause-combining operations which have essentially the same coverage as those used in the SPaRKy Restaurant Corpus. This paper fills a gap in the literature, showing that it is possible to learn microplanning rules for both aggregation and discourse connective insertion, an important step towards ameliorating the knowledge acquisition bottleneck for NLG systems that produce texts with rich discourse structures using traditional architectures.

1 Introduction

In a traditional natural language generation (NLG) system (Reiter and Dale, 2000), a pipeline of hand-crafted components is used to generate high quality text, albeit at considerable knowledge-engineering expense. While there has been progress on using machine learning to ameliorate this issue in content planning (Duboue and McKeown, 2001; Barzilay and Lapata, 2005) and broad coverage surface realization (Reiter, 2010; Rajkumar and White, 2014), the central stage of sentence planning (or microplanning) has proved more difficult to automate. More recently, Angeli et al. (2010) and Konstas and Lapata (2013), inter alia, have developed end-to-end learning methods for NLG systems; however, as discussed further in the next section, these systems assume quite limited discourse structures in comparison to those with more traditional architectures.

In this paper, we describe a method of inducing clause-combining rules of the kind used in traditional sentence planners. In particular, we base our approach on the architecture used in the SPaRKy restaurant recommendation system (Walker et al., 2007), where a sentence plan generator is used to map a text plan to a range of possible sentence plans, from which one is selected for output by a sentence plan ranker.[1] To demonstrate the viability of our method, we present an experiment demonstrating that rules corresponding to all of the hand-crafted operators for aggregation and discourse connective insertion used in the SPaRKy Restaurant Corpus can be effectively learned from examples of their use. To our knowledge, these induced rules for the first time incorporate the constraints necessary to be functionally equivalent to the hand-crafted clause-combining operators; in particular, our method goes beyond the one Stent and Molina (2009) develop for learning clause-combining rules, which focuses on learning domain-independent rules for discourse connective insertion, ignoring aggregation rules and any potentially domain-dependent aspects of the rules. As such, our approach promises to be of immediate benefit to NLG system developers, while also taking an important step towards reducing the knowledge acquisition bottleneck for developing NLG systems requiring rich discourse structures in their outputs.

2 Related Work

Angeli et al. (2010) present an end-to-end trainable NLG system that generates by selecting a

[1]The sentence plan ranker uses machine learning to rank sentence plans based on features derived from the sentence plan and its realization, together with accompanying human ratings for the realizations in the training data. As such, the SPaRKy architecture differs from traditional ones in using machine learning to rank potential outputs, but it follows the traditional architecture in making use of lexicalization, aggregation and referring expression rules in a distinct sentence planning stage.

sequence of database records to describe, a sequence of fields on those records to mention, and finally a sequence of words for expressing the values of those fields. Though Konstas and Lapata (2013) generalize Angeli et al.'s approach, they acknowledge that handling discourse-level document structure remains for future work. Given this limitation, under their approach there is no need to explicitly perform aggregation: instead, it suffices to "pre-aggregate" propositions about the same entity onto the same record. However, in the general case aggregation should be subject to discourse structure; for example, when contrasting the positive and negative attributes of an entity according to a given user model, it makes sense to aggregate the positive and negative attributes separately, rather than lumping them together (White et al., 2010). Consequently, we aim to learn aggregation rules that are sensitive to discourse structure, as with the SPaRKy architecture.

Other notable recent approaches (Lu et al., 2009; Dethlefs et al., 2013; Mairesse and Young, 2014) are similar in that they learn to map semantic representations to texts using conditional random fields or factored language models with no explicit model of syntactic structure, but the content to be expressed is assumed to be pre-aggregated in the input. Kondadadi et al. (2013) develop a rather different approach where large-scale templates are learned that can encapsulate typical aggregation patterns, but the templates cannot be dynamically combined in a way that is sensitive to discourse structure

Previous work on aggregation in NLG, e.g. with SPaRKy itself or earlier work by Pan and Shaw (2004), focuses on learning when to apply aggregation rules, which are themselves hand-crafted rather than learned. The clause-combining rules our system learns—based on lexico-semantic dependency edits—are closely related to the lexico-syntactic rewrite rules learned by Angrosh and Siddharthan's (2014) system for text simplification. However, our learned rules go beyond theirs in imposing (non-)equivalence constraints crucial for accurate aggregation. Finally, work on text compression (Woodsend and Lapata, 2011; Cohn and Lapata, 2013) is also related, but focuses on simple constituent deletion, and to our knowledge does not implement aggregation constraints such as those here.

3 SPaRKy Restaurant Corpus

Walker et al. (2007) developed SPaRKy (a Sentence Planner with Rhetorical Knowledge) to extend the MATCH system (Walker et al., 2004) for restaurant recommendations. In the course of their study they produced the SPaRKy Restaurant Corpus (SRC), a collection of content plans, text plans and the surface realizations of those plans evaluated by users.[2]

While the restaurant recommendation domain is fairly narrow in terms of the kinds of propositions represented, it requires careful application of aggregation operations to make concise, natural realizations. This is evident both in the care taken in incorporating clause-combining rules into SPaRKy and in subsequent work on the expression of contrast which used this domain to motivate extensions of CCG to the discourse level (Nakatsu and White, 2010; Howcroft et al., 2013). Five kinds of clause-combining operations are included in SPaRKy, most of which involve lexically specific constraints. These are illustrated in Table 2, using propositions corresponding to sentences (1)–(4) from Table 1 as input (combined with either a CONTRAST or INFER relation). MERGE combines two clauses if they have the same verb with the same arguments and adjuncts except for one. WITH-REDUCTION replaces an instance of *have* plus an object X with the phrase *with* X. REL-CLAUSE subordinates one clause to another when they have a common subject. CUE-WORD-CONJUNCTION combines clauses using the conjunctions *and*, *but*, and *while*, while CUE-WORD-INSERTION combines clauses by inserting *however* or *on the other hand* into the second clause. Table 2 also shows two operations, VP-COORDINATION and NP-APPOSITION, which go beyond those in SPaRKy; these are discussed further in Section 5.5. Finally, it's also possible to leave sentences as they are, simply juxtaposing them in sequence.

For the experiments reported in this paper, we have reimplemented SPaRKy to work with OpenCCG's broad coverage English grammar for parsing and realization (Espinosa et al., 2008; White and Rajkumar, 2009; White and Rajkumar,

[2]Available from `http://users.soe.ucsc.edu/~maw/downloads.html` under *the textplans/utterances*. To our knowledge, the SRC remains the only publicly available corpus of input–output pairs for an NLG system using discourse structures with rhetorical relations.

Operator	Sents	Result
MERGE	1, 2	Sonia Rose has good decor and good service.
WITH-REDUCTION	1, 2	Sonia Rose has good decor, with good service.
REL-CLAUSE	1, 2	Sonia Rose, which has good service, has good decor.
CUE-WORD-CONJUNCTION	1, 3	Sonia Rose has good service, but Bienvenue has very good service.
CUE-WORD-INSERTION	1, 3	Sonia Rose has good service. However, Bienvenue has very good service.
VP-COORDINATION	3, 4	Bienvenue is a French restaurant and has very good service.
NP-APPOSITION	3, 4	Bienvenue, a French restaurant, has very good service.

Table 2: SRC clause-combining operations plus two additional operations we examined.

(1)	Sonia Rose has good decor.
(2)	Sonia Rose has good service.
(3)	Bienvenue has very good service.
(4)	Bienvenue is a French restaurant.

Table 1: Example sentences from the SRC domain.

2012).[3] As in the original SPaRKy, the sentence planner takes as input a text plan, which encodes the propositions to be expressed at the leaves of a tree whose internal nodes are marked with rhetorical relations. The sentence planner then rewrites the text-plan tree using a sequence of lexicalization, clause-combining and referring expression rules. The obligatory lexicalization rules straightforwardly rewrite the domain-specific propositions into a domain-general OpenCCG lexico-semantic dependency graph, or logical form (referred to as a TPLF in Section 5.1). After lexicalization, the clause-combining and referring expression rules optionally apply to rewrite the logical form into a set of alternative logical forms, among which is ideally one or more options that will express the content concisely and fluently after each sentence is realized; if none of the clause-combining and referring expression rules apply, the text will be realized as a sequence of very simple one-clause sentences, with proper names used for all restaurant references.

As noted earlier, the task of choosing a particular logical form alternative belongs to the sentence plan ranker; since its task is largely independent of the task of generating alternative logical forms, we do not address it in this paper. Indeed, to the extent that our sentence planner produces logical forms that are functionally equivalent to the alternative sentence plans in the SRC, we can expect the output quality of the reimplemented system with a suitably trained sentence plan ranker to be essentially unchanged, and thus an evaluation of this kind would be uninformative.

[3]The lexicon is extended by the addition of the restaurant names as proper nouns to avoid spurious bad parses resulting from unknown words.

An example aggregation rule for the OpenCCG-based system (going beyond the options in the SRC) appears in Figure 1, and an example input–output pair for this rule appears in Figure 2. As the latter figure shows, a dependency graph consists of a set of nodes and relations between them, where sibling nodes are taken to be unordered. Nodes themselves comprise an identifier, a predicate label and an unordered list of attribute-value pairs. The graphs have a primary tree structure; the graphs in Figure 2 are in fact trees, but in the general case, node references can be used to represent nodes with multiple parents or even cycles (in the case of relative clauses). The alignments between nodes in the input and output graphs are shown in the figure by using the same identifier for corresponding nodes.

Clause-combining rules such as the one in Figure 1 are applied by unifying the left hand side of the rule against an input dependency graph, returning the graph specified by the right hand side of the rule if unification succeeds and any further specified constraints are satisfied. The rules are implemented in Prolog using an enhanced unification routine that allows sibling nodes to be treated as unordered, and which allows list variables (shown between dots in the figure) to unify with the tail (i.e., remainder) of a list of child nodes or a list of attributes. In the example at hand, the variables `G`, `C`, `E` and `I` are unified with node identifiers `n(1)`, `n(2)`, `n(7)` and `n(8)`, respectively. The list variables `...D...` and `...F...` unify with the empty list since the nodes for *Bienvenue* have no child nodes, while `...L...` unifies with a list consisting of the relation `Arg1` together with the subgraph headed by `n(3)`, and `...K...` unifies with a list consisting of the relations `Det` and `Mod` together with their respective subgraphs headed by `n(9)` and `n(10)`. The list variables over attributes unify trivially. Finally, after checking the

```
rule:
_ infer-rel [ .._.. ]
  Arg1
    G have [ ..H.. ]
      Arg0
        C A [ ..B.. ]
          ...D...
      ...L...
  Arg2
    _ be [ .._.. ]
      Arg0
        E A [ ..B.. ]
          ...F...
      Arg1
        I restaurant [ ..J.. ]
          ...K...

s.t. [equiv(node(C,A,B,D),node(E,A,B,F))]

==>

G have [ ..H.. ]
  Arg0
    C A [ ..B.. ]
      ApposRel
        I restaurant [ ..J.. ]
          ...K...
      ...D...
  ...L...
```

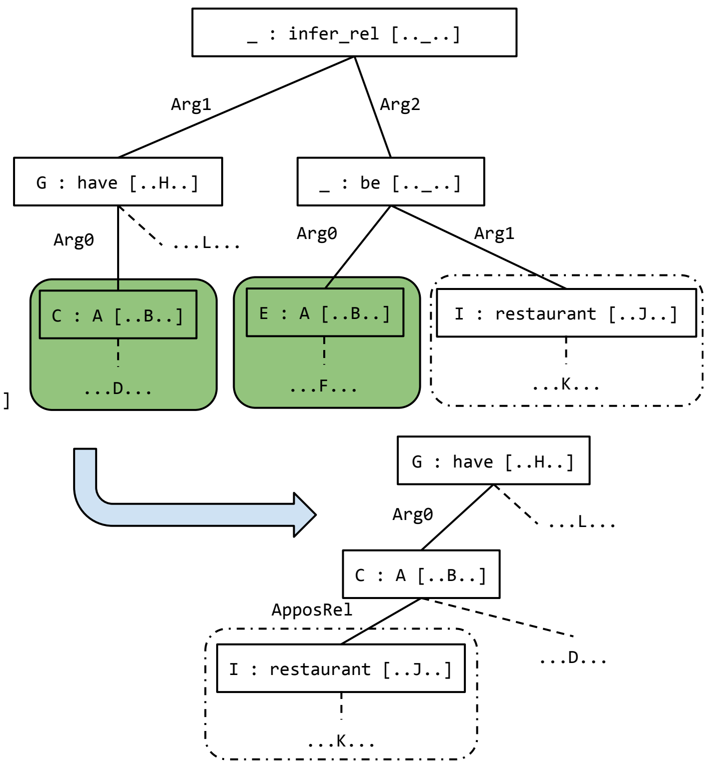

Figure 1: On the left is a textual representation of an NP-APPOSITION operation inferred from combining sentences 3 and 4 as shown in Table 2; on the right is a graphical representation. Capital letters represent variables, and underscores represent anonymous variables; variables over lists of attributes or dependencies are shown between dots. The solid rounded boxes highlight content that must be equivalent for the rule to apply, while the dotted rounded box shows the content preserved by the rule.

nodes headed by `n(2)` and `n(7)` for equivalence (i.e, isomorphism), the right hand side of the rule is returned, where the root `infer-rel`, `be` and second `Bienvenue` nodes have been left out, and the `restaurant` node `n(8)` has been retained under `n(2)` via an `ApposRel`.

In comparison to Angrosh and Siddharthan's (2014) lexical rewrite rules—which consist simply of a list of edit operations—we find the clause-combining rules induced by our approach to be quite readable, thus in principle facilitating their manual inspection by NLG developers.

4 Rule Induction Algorithm

In this section, we present the rule induction algorithm at an overview level; for complete details, see the rule induction code to be released on the OpenCCG website.[4]

[4]`http://openccg.sf.net`

4.1 Input–Output Pre-Processing

The induction method takes as input pairs of text plans (taken to be ordered) and sentences realizing the text plans, and returns as output induced rules, such as the one just seen in Figure 1. To begin, the text plans are lexicalized using simple hand-crafted lexicalization rules, as noted above, yielding initial OpenCCG logical forms (LFs). Meanwhile, the sentences are parsed to LFs serving as the target output of the rule, after any anaphoric expressions have been resolved (in an ad hoc way) with respect to the source LF; in particular, the LF nodes for the expressions *it*, *its* and *this CUISINE restaurant* are replaced with versions using the proper name of the restaurant, e.g. LF nodes for *Bienvenue*, *Bienvenue's* and *Bienvenue, which is a French restaurant*.[5]

[5]The restaurant name is located by searching for the first predicate under an `Arg0` relation, unless there is a `GenOwn` (possessor) relation under it, in which the predicate under `GenOwn` is returned. This method is generally reliable, though errors are sometimes introduced when multiple restaurants are mentioned in alternation. A more accurate method would take alignments into account.

Input dependency graph for *Bienvenue has very good service. Bienvenue is a French restaurant.*

```
n(0) infer-rel
  Arg1
    n(1) have [ mood=dcl tense=pres ]
      Arg0
        n(2) Bienvenue [ num=sg ]
      Arg1
        n(3) service [ det=nil num=sg ]
          Mod
            n(4) good
              Mod
                n(5) very
  Arg2
    n(6) be [ mood=dcl tense=pres ]
      Arg0
        n(7) Bienvenue [ num=sg ]
      Arg1
        n(8) restaurant [ num=sg ]
          Det
            n(9) a
          Mod
            n(10) French [ num=sg ]
```

Output dependency graph for *Bienvenue, a French restaurant, has very good service.*

```
n(1) have [ mood=dcl tense=pres ]
  Arg0
    n(2) Bienvenue [ num=sg ]
      ApposRel
        n(8) restaurant [ num=sg ]
          Det
            n(9) a
          Mod
            n(10) French [ num=sg ]
  Arg1
    n(3) service [ det=nil num=sg ]
      Mod
        n(4) good
          Mod
            n(5) very
```

Figure 2: Example input–ouput dependency graphs for NP-APPOSITION clause-combining rule

Input dependency graph for *Mangia has very good food quality. Mangia has decent decor.*

```
n(0) infer-rel
  Arg1
    n(1) have [ mood=dcl tense=pres ]
      Arg0
        n(2) Mangia [ num=sg ]
      Arg1
        n(3) quality [ det=nil num=sg ]
          Mod
            n(4) food [ num=sg ]
          Mod
            n(5) good
              Mod
                n(6) very
  Arg2
    n(7) have [ mood=dcl tense=pres ]
      Arg0
        n(8) Mangia [ num=sg ]
      Arg1
        n(9) decor [ det=nil num=sg ]
          Mod
            n(10) decent
```

Output dependency graph for *Mangia has very good food quality, with decent decor.*

```
n(1) have [ mood=dcl tense=pres ]
  Arg0
    n(2) Mangia [ num=sg ]
  Arg1
    n(3) quality [ det=nil num=sg ]
      Mod
        n(4) food [ num=sg ]
      Mod
        n(5) good
          Mod
            n(6) very
  Mod
    n(11) with [ emph-final=+ ]
      Arg1
        n(9) decor [ det=nil num=sg ]
          Mod
            n(10) decent
```

Figure 3: Example input–ouput dependency graphs for WITH-REDUCTION clause-combining rule

The next step is to align the nodes of the input and output LFs. We have found that a simple greedy alignment routine works reliably with the SRC, where nodes with unique lexical matches are aligned first, and then the remaining nodes are greedily aligned according to the number of parent and child nodes already aligned. After alignment, the parts of the output LF corresponding to each sentence are rebracketed to better match the grouping in the input LF (revising an initial right-branching structure). To rebracket the sentence-level LFs, the adjacent pair of LFs whose aligned nodes in the source LF have the minimum path distance is iteratively grouped together under an INFER relation until the structure has a single root.

4.2 Edit Analysis and Rule Construction

Following alignment and sentence-level rebracketing, the difference between the input and output dependency graphs is calculated, in terms of inserted/deleted nodes, relations and attributes. Next, these edits are analyzed to determine whether any equivalent nodes have been **factored out**—that is, whether a node that is isomorphic to another one in the input has been removed. For example, in Figure 1, nodes C and E are derived from the isomorphic nodes for the restaurant name NPs, with node E left out of the output.

Based on the edit analysis, one of four general kinds of clause-combining rules may be inferred: two kinds of aggregation rules, one involving a shared argument and one a shared predication, as well as two kinds of rules for adding discourse connectives based on discourse relations, one where clausal LFs are combined, and another where a connective is inserted into the second LF. The kinds of rules for discourse connectives correspond directly to those in SPaRKy; for aggregation, shared predication rules correspond to oper-

```
rule:
_ one_of(_,[infer-rel,justify-rel]) [ .._.. ]
  Arg1
    G H [ ..I.. ]
      Arg0
        C A [ ..B.. ]
          ...D...
      ...N...
  Arg2
    _ have [ .._.. ]
      Arg0
        E A [ ..B.. ]
          ...F...
      Arg1
        J K [ ..L.. ]
          ...M...

s.t. [equiv(node(C,A,B,D),node(E,A,B,F))]

==>

G H [ ..I.. ]
  Arg0
    C A [ ..B.. ]
      ...D...
  Mod
    _ with [ emph-final=+ ]
      Arg1
        J K [ ..L.. ]
          ...M...
  ...N...
```

Figure 4: Inferred WITH-REDUCTION clause-combining rule with generalized lexical constraints

ations of the MERGE kind, with shared argument rules accounting for the rest.

To keep the rule induction straightforward, exactly one node is required to have been factored out with the aggregation rules, while with the discourse connective rules, the edits must be localized to the level directly below the triggering discourse relation. When these conditions are satisfied, an induced rule is constructed based on the edits and any applicable constraints. First, the left hand side of the rule is constructed so that it matches any deleted nodes, attributes and relations, as well as the path to both the factored out node (if any), the one it is equivalent to, and the parents of any inserted nodes. Along the way, lexical predicates are included as requirements for the rule to match, except in the case of factored out nodes, where it is assumed that the lexical predicate does not matter. Next, the right hand side of the rule is constructed, leaving out any matched nodes, attributes or relations to be deleted, while adding any nodes, attributes or relations to be inserted. Finally, any applicable constraints are added to the rule. (Though both kinds of aggregation rules are triggered off of factored out nodes, shared predication rules actually involve a stronger constraint, namely that all but one argument of a predicate be equivalent.)

4.3 Constraints and Generalization

The constraints included in the aggregation rules are essential for their accurate application, as noted earlier. For example, in the absence of the shared argument constraint for an inferred RELATIVE-CLAUSE rule, adjacent clauses for *Sonia Rose has good service* and *Bienvenue has very good service* could be mistakenly combined into *Sonia Rose, which has very good service, has good service*, as nothing would check whether *Sonia Rose* and *Bienvenue* were equivalent. The lexical predicate constraints are also essential, for example to ensure that only *have*-predications are reduced to *with*-phrases, and that only *be*-predications are eligible to become NP-appositives.

After a first pass of rule induction, the rules are generalized by combining rules that differ only in a lexical predicate, and if a sufficient number of lexical items has been observed (three in our experiments here), the lexical constraint is removed, much as in Angrosh and Siddharthan's (2014) approach. For example, the rule in Figure 4—induced from the input–output pair in Figure 3 and others like it—has been generalized to work with either the `infer-rel` or `contrast-rel` relations, and the predication for the first argument of the relation (`H`) has been generalized to apply to any predicate.

4.4 Rule Interaction During Learning

Since evidence for a rule may not always be directly available in an input–output pair that illustrates the effect of that rule alone, rule induction is also attempted from all subgraphs of the input and output that are in an appropriate configuration—namely, where either the roots of the source and target subgraphs are aligned, or where at least one child node of the root of the source subgraph is aligned with the root of the target subgraph or one of its children.

Inducing rules from subgraphs in this way is a noisy process that can yield bad rules, that is, ones that mistakenly delete or insert nodes for words: nodes can be mistakenly deleted when the target subgraph is missing nodes supplying propositional content in the source, while nodes can be mistakenly inserted if they supply extra propositional content not present in the source rather than discourse connectives or function words, as intended. An example bad deletion rule appears in

```
rule:
_ quality [ .._.. ]
  Det
    E A [ ..B.. ]
      ...F...
  Mod
    G among [ ..H.. ]
      Arg1
        I restaurant [ ..J.. ]
          Det
            C A [ ..B.. ]
              ...D...
          ...K...
      ...L...
  Mod
    _ best [ .._.. ]
  Mod
    _ overall [ .._.. ]

s.t. [equiv(node(C,A,B,D),node(E,A,B,F))]

==>

G among [ ..H.. ]
  Arg1
    I restaurant [ ..J.. ]
      Det
        C A [ ..B.. ]
          ...D...
      ...K...
  ...L...
```

Figure 5: An undesirable rule that mistakenly reduces *the best overall quality among the selected restaurants* to *among the selected restaurants*, induced from LF subgraphs for these phrases

Figure 5. Another common cause of bad rules is errors in parsing the target sentences, especially longer ones. To lessen the prevalence of bad induced rules, we take inspiration from work on learning semantic parsers (Kwiatkowksi et al., 2010; Artzi and Zettlemoyer, 2013) and embed the process of inducing clause-combining rules within a process of learning a model of preferred derivations that use the induced rules. The model is learned using the structured averaged perceptron algorithm (Collins, 2002), with indicator features for each rule used in deriving an output LF from and input LF. With each input–output pair, the current model is used to generate the highest-scoring output LF using a bottom-up beam search. If the highest-scoring output LF is not equal to the target LF, then the search is run again to find the highest-scoring derivation of the target LF, using the distance to the target to help guide the search. When the target LF can be successfully derived, a perceptron update is performed, adjusting the weights of the current model by adding the features from the target LF derivation and subtracting the ones from the highest-scoring non-target LF. At the end of all training epochs, the parameters from the final model and all the intermediate models are averaged, which approximates the margin-maximizing voted perceptron algorithm.

For input–output pairs satisfying increasing size limits:

1. **Direct Epoch** Starting with an empty model, clause-combining rules are induced directly from input–output pairs.
2. **Generalization** The current set of rules is generalized and the training examples are revisited to update the weights for the newly added rules. Subsumed rules are removed, and the initial weight of the generalized rule is set to the maximum weight of the subsumed rules.
3. **Subgraphs Epoch** Rules are induced from all applicable subgraphs of the input–output pairs.
4. **Generalization** As above.
5. **Partial Epoch** For any examples where the target LF cannot be generated with the current ruleset, rules are induced from an n-best list of partially completed outputs paired with the target LF.
6. **Generalization** As above.
7. **Pruning** After switching to the final averaged model, any rules not used in the highest-scoring derivation of an example are pruned.

Figure 6: Algorithm Summary

It is often the case that desired rules cannot be induced either directly from an input–output pair or from their subgraphs: instead, other learned rules need to be applied to the input before the example illustrates the desired step.[6] Accordingly, for input–output pairs that cannot be derived with the current set of rules, we generate an n-best list of outputs and then attempt to induce a rule by pairing each partially complete output with the target LF as an input–output pair.

[6]Consider a text realizing the same content as (1) and (2) in addition to the content in *Sonia Rose is an Italian restaurant*. A single sentence realization of this content is *Sonia Rose, an Italian restaurant, has good decor and good service*. In order to learn NP-APPOSITION from this input–output pair, the system must first have learned and applied a MERGE rule to (1) and (2) so that the structures are sufficiently parallel for inference to proceed.

4.5 Staged Learning

A summary of the rule induction algorithm appears in Figure 6. In the outermost loop, the algorithm is run over input–output pairs that meet a given size limit on the output LF, with this limit increasing with each iteration. Since during development we found that rules induced (i) directly, (ii) from subgraphs and (iii) from partially completed inputs were of decreasing reliability, such rules are induced in separate training epochs in that order. A final pruning step removes any rules not used in the highest-scoring derivation of an input–output pair, using the averaged model. The pruning step is expected to remove most of the bad rules involving undesirable node insertions or deletions, as they are typically downweighted by the perceptron updates.

5 Evaluation

5.1 Data preparation

We convert the text plans used in Walker et al. (2007) to a more concise logical representation as described in Section 3.

Using OpenCCG's broad-coverage grammar, we then parse the SRC realizations corresponding to these text plans, resulting in one LF for each sentence in the SRC. Since nearly all realizations in the SRC include multiple sentences, this results in multiple LFs for each. In order to combine these sentence-level LFs into a single sentence plan LF (SENTLF), we impose an initial binary right-branching structure over the LFs and label the resulting superstructure nodes with the `infer-rel` predicate. As noted in Section 4, the initial right-branching structure is subsequently rebracketed to better match the rhetorical structure in the of the text plan LF (TPLF).

5.2 Dev, Training, and Test Splits

We limit our attention to realizations in the SRC containing 5 or fewer sentences and only one subject per sentence.[7] Of the 1,760 sparky pairs in the SRC, we used a set of 73 sparky pairs for development, defining a sparky pair as the TPLF–SENTLF pair corresponding to a single sparky alternative. These pairs were used primarily for debugging and testing and are not used further in the evaluation.

[7]The only multi-subject sentences in the SRC are of the form, "Restaurant A, (Restaurant B, ...) and Restaurant N offer exceptional value among the selected restaurants" and do not add to the variety of clause-combining operations of interest to us.

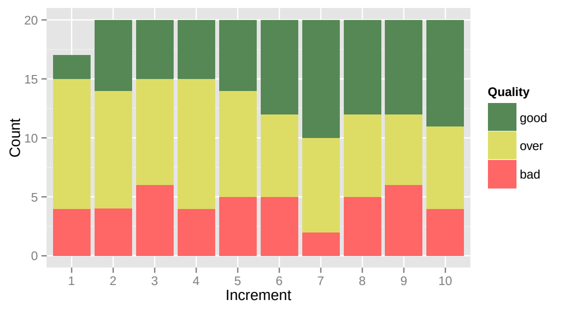

Figure 7: Manual evaluation of the quality of the top 20 rules as good, overspecified (but still valid) or bad, for training increments of increasing size

We use 700 sparky pairs for the training and test sets and reserve the remaining sparky pairs for future work. The training set consists of 200 sparky pairs used for rule induction, where we incrementally add 20 pairs at a time to the training set to evaluate how much training data is necessary for our approach to work. The test set consists of 500 sparky pairs for use in evaluating the coverage of the rules inferred during training.

5.3 Quality of Learned Rules

In our first evaluation we evaluate the rate of rule acquisition. We present the algorithm first with 20 sparky pairs and then add an additional 20 sparky pairs in each iteration, resulting in 10 sets of learned rules to compare to each other. This allows us to see how well new data allows the algorithm to generalize the induced rules.

To evaluate the quality of the learned rules, we conducted a manual evaluation of the top 20 rules as ranked by the perceptron model. We report the proportion of these rules rated as good, overspecified (more specific than desired, but still valid) or bad in Figure 7. As the figure shows, the proportion of good rules increases relative to the overspecified ones, with the proportion of bad rules remaining low. With the full training set, a total of 46 rules are learned, almost equally split between good, overspecified and bad (15/17/14, resp.).

In examining the learned rules, we observe that the learning algorithm manages to diminish the number of highly-ranked bad rules with spurious content changes, as these only infrequently contribute towards deriving a target LF. However, the presence of bad rules owing to parse errors persists, as certain parse errors occur with some regularity. As such, in future work we plan to investigate whether learning from n-best parses can manage to better work around erroneous parses.

5.4 Coverage

The first question of coverage is straighforward: do the rules we learn recover all of the types of clause-combining operations used by SPaRKy? Manual evaluation reveals good coverage for all five kinds of clause-combining operations in the top 20 rules of the final model. WITH-REDUCTION, MERGE, CUE-WORD-INSERTION, and CUE-WORD-CONJUNCTION (for all connectives in the corpus) are covered by good rules, i.e. ones that are comparable in quality to the kind we might write by hand. In addition to these good rules, WITH-REDUCTION and CUE-WORD-CONJUNCTION are also represented in several overspecified rules. RELATIVE-CLAUSE is only represented by overspecified rules.[8]

Additionally, in order to assess the extent to which the learned rules cover the contexts where the SPaRKy clause-combining operations can be applied in the SRC, we also applied the final set of learned rules to all of the input TPLFs in the test set. The test set contained 453 usable input pairs,[9] of which we were able to exactly reproduce 229 using the inferred rules. Naturally, we do not expect 100% coverage here as the test set will also contain some LFs suffering from parse errors, though this coverage level suggests our method would benefit from a larger training set. Importantly, applying the learned rules to the test set input generated 19,058 possible sentLFs (or 40 output LFs per input LF, on average), a sufficiently large number for a sentence plan ranker to learn from.

5.5 Experimenting with other clause-combining operations

Using a single-stage version of the algorithm, we also examined its capabilities with respect to learning clause-combining operations not present in the SRC. To create training examples, we created 167 input pairs based on TPLFS from the SRC using set of 16 hand-crafted rules including all the clause-combining operations pictured in Table 2. From these 167 pairs the algorithm induced 22 clause-combining operations, fully covering the 16 hand-crafted rules with some overspecification. Importantly, this preliminary finding suggests that developers can use this system to acquire a larger variety of clause-combining operations than those represented in the SRC with less need for extensive knowledge engineering.

6 Conclusions and Future Work

We have presented a clause-combining rule induction method that learns how to rewrite lexico-semantic dependency graphs in ways that go beyond current end-to-end NLG learning methods (Angeli et al., 2010; Konstas and Lapata, 2013), an important step towards ameliorating the knowledge acquisition bottleneck for NLG systems that produce texts with rich discourse structures. Future work will evaluate the system on multiple domains and push into the realm of robust, simultaneous induction of lexicalization, clause-combining and referring expression rules.

Acknowledgments

We thank the anonymous reviewers and the OSU Clippers Group for their helpful feedback and discussion. Supported in part by NSF grants IIS-1143635 and IIS-1319318 and by DFG SFB 1102.

References

Gabor Angeli, Percy Liang, and Dan Klein. 2010. A simple domain-independent probabilistic approach to generation. In *Proc. of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 502–512, Cambridge, MA, October.

Mandya Angrosh and Advaith Siddharthan. 2014. Text simplification using synchronous dependency grammars: Generalising automatically harvested rules. In *Proc. of the 8th International Natural Language Generation Conference*, pages 16–25, Philadelphia, Pennsylvania, USA, June.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *TACL*, 1:49–62.

Regina Barzilay and Mirella Lapata. 2005. Collective content selection for concept-to-text generation. In *Proc. of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 331–338, Vancouver.

[8]Naturally elements of these rules are also present in some of the genuinely bad rules that remain in the final model. Even in these bad rules, however, the operations are appropriately constrained: WITH- REDUCTION is predicated on the use of *have* and CUE-WORD-INSERTION of *on the other hand* on the use of `contrast-rels` and of *since* on the use of `justify-rels`.

[9]Of the original 500 pairs, we excluded 29 pairs larger than the maximum size used in our staged learning, as well as 18 pairs where the target realization could not be parsed.

Trevor Cohn and Mirella Lapata. 2013. An abstractive approach to sentence compression. *ACM Transactions on Intelligent Systems and Technology*, 4(3):1–35.

Michael Collins. 2002. Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms. In *Proc. of the 2002 Conference on Empirical Methods in Natural Language Processing*.

Nina Dethlefs, Helen Hastie, Heriberto Cuayáhuitl, and Oliver Lemon. 2013. Conditional random fields for responsive surface realisation using global features. In *Proc. of the 51st Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 1254–1263, Sofia, Bulgaria.

Pablo A. Duboue and Kathleen R. McKeown. 2001. Empirically estimating order constraints for content planning in generation. In *Proc. of 39th Annual Meeting of the Association for Computational Linguistics*, pages 172–179, Toulouse, France, July.

Dominic Espinosa, Michael White, and Dennis Mehay. 2008. Hypertagging: Supertagging for surface realization with CCG. In *Proc. of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 183–191, Columbus, Ohio, June.

David M Howcroft, Crystal Nakatsu, and Michael White. 2013. Enhancing the expression of contrast in the SPaRKy Restaurant Corpus. In *Proc. of the 14th European Workshop on Natural Language Generation*.

Ravi Kondadadi, Blake Howald, and Frank Schilder. 2013. A statistical nlg framework for aggregated planning and realization. In *Proc. of the 51st Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 1406–1415, Sofia, Bulgaria.

Ioannis Konstas and Mirella Lapata. 2013. A global model for concept-to-text generation. *Journal of Artificial Intelligence Research*, 48:305–346.

Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proc. of the 2010 Conference on Empirical Methods in Natural Language Processing*.

Wei Lu, Hwee Tou Ng, and Wee Sun Lee. 2009. Natural language generation with tree conditional random fields. In *Proc. of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 400–409, Singapore.

François Mairesse and Steve Young. 2014. Stochastic language generation in dialogue using factored language models. *Computational Linguistics*, 40(4):763–799.

Crystal Nakatsu and Michael White. 2010. Generating with discourse combinatory categorial grammar. *Language Issues in Language Technology*, 4(1):1–62.

Shimei Pan and James Shaw. 2004. Segue: A hybrid case-based surface natural language generator. In *Proc. of the 3rd International Conference of Natural Language Generation*, pages 130–140, Brockenhurst, UK.

Rajakrishnan Rajkumar and Michael White. 2014. Better surface realization through psycholinguistics. *Language and Linguistics Compass*, 8(10):428–448.

Ehud Reiter and Robert Dale. 2000. *Building natural generation systems*. Studies in Natural Language Processing. Cambridge University Press.

Ehud Reiter. 2010. Natural language generation. In Alexander Clark, Chris Fox, and Shalom Lappin, editors, *The Handbook of Computational Linguistics and Natural Language Processing*, Blackwell Handbooks in Linguistics, chapter 20. Wiley-Blackwell.

Amanda Stent and Martin Molina. 2009. Evaluating automatic extraction of rules for sentence plan construction. In *Proc. of the SIGDIAL 2009 Conference*, pages 290–297, London, UK.

Marilyn A. Walker, S. Whittaker, Amanda Stent, P. Maloor, J. Moore, M. Johnston, and G. Vasireddy. 2004. Generation and evaluation of user tailored responses in multimodal dialogue. *Cognitive Science*, 28(5):811–840, October.

Marilyn Walker, Amanda Stent, François Mairesse, and Rashmi Prasad. 2007. Individual and domain adaptation in sentence planning for dialogue. *Journal of Artificial Intelligence Research*, 30:413–456.

Michael White and Rajakrishnan Rajkumar. 2009. Perceptron reranking for CCG realization. In *Proc. of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 410–419, Singapore, August.

Michael White and Rajakrishnan Rajkumar. 2012. Minimal dependency length in realization ranking. In *Proc. of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 244–255, Jeju Island, Korea, July.

Michael White, Robert AJ Clark, and Johanna D Moore. 2010. Generating tailored, comparative descriptions with contextually appropriate intonation. *Computational Linguistics*, 36(2):159–201.

Kristian Woodsend and Mirella Lapata. 2011. Learning to simplify sentences with quasi-synchronous grammar and integer programming. In *Proc. of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 409–420, Edinburgh, Scotland, UK.