A Dynamic Approach for Automatic Error Detection in Generation Grammars

Tim vor der Brück[1] and **Holger Stenzhorn**[2]

1 Introduction

In any real–world application scenario, natural language generation (NLG) systems have to employ grammars consisting of tremendous amounts of rules. Detecting and fixing errors in such grammars is therefore a highly tedious task.

In this work we present a data mining algorithm which deduces incorrect grammar rules by abductive reasoning out of positive and negative training examples. More specifically, the constituency trees belonging to successful generation processes and the incomplete trees of failed ones are analyzed. From this a quality score is derived for each grammar rule by analyzing the occurrences of the rules in the trees and by spotting the exact error locations in the incomplete trees.

In prior work on automatic error detection v.d.Brück et al. [5] proposed a static error detection algorithm for generation grammars. The approach of Cussens et al. creates missing grammar rules for parsing using abduction [1]. Zeller introduced a dynamic approach in the related area of detecting errors in computer programs [6].

2 Error Detection

The basic purpose of NLG, as considered here, is to convert an input structure, given as feature value pairs, by means of grammar rules into a constituency tree where the surface text can be read off as the terminal yield (we use the formalism of Busemann [3]). Each non–leaf node in this tree is associated to a particular input substructure, a category and the applied grammar rule while the leaf nodes are associated to text segments. The final surface text is created by concatenating the text segments of the leaf nodes.

In case of success, the generation system not only returns the generated surface text (or texts if multiple possible solutions have been found) but also the associated constituency tree (or trees). However, in the case of failure, no surface text is generated and no associated constituency tree exists. However, it is obvious that, in order to detect a specific spot where the generation process fails, it is highly advantageous to have partial constituency trees for the failed generation attempts as well.

For this reason, the employed generation engine has been extended to provide two types of partial trees in case of generation failures: The tree of the first type is the largest tree to result from the generation process – we call the *maximum tree*.

The other alternative tree, representing a non–successful generation, is the one having the smallest total difference to a complete tree – we call the *minimum tree*. Usually both types of trees are incomplete and hence can have non–terminal categories at its leaf nodes.

In the following, a complete tree resulting from a successful generation is called a *positive tree* while an incomplete tree (either maximum or minimum) is called a *negative tree*.

The detection of incorrect rules is basically done in several consecutive steps:

1. First, a global (i.e., independent of any specific input structure) rule quality score (*gqs*) is derived for each rule.
2. For each input structure leading to a generation failure, the most probable error location in the associated constituency tree is detected.
3. Both information are put together to derive a local rule quality score (*lqs*) which is associated to a certain input structure. The rules with the lowest *lqs* (and *gqs* below a given threshold) are considered as potentially erroneous.

1. Deriving a Global Rule Quality Score

If a certain rule appears in a positive generation tree, this generally indicates that the rule is correct. However, the fact that a rule is appearing in a negative tree or in no constituency tree at all, is an indicator for an incorrect rule. By using this information, a *gqs* is defined for each rule.

The *gqs* of a rule reflects the probability that a generation fails if this rule appears in the associated constituency tree. Thus, the *gqs* is defined as the negated probability that a tree is negative if the rule r occurs in that tree: $gqs := -P(t \in T^- | (lhs(r), r) \in t)$ where

- T^- : the set of negative trees
- $lhs(r)$: left-hand side (LHS) category of rule r (see [3])

As usual, the probability is estimated by the relative frequency of a tree being negative if a certain rule appears in it. If a rule never appears in either the positive or negative trees then its *gqs* is set to -1 since this is a strong indication of a potential error. A rule is assumed to be correct if its *gqs* exceeds a given threshold h.

To account for the fact that the probabilities for rules leading to negative trees (or that they appear in no tree at all) are not independent from each other (i.e., a rule might be assigned to a low score because of an error in an ancestor rule), a small portion of the *gqs* is propagated upwards and added to each rule's *gqs* where that rule could,

[1] FernUniversität in Hagen, Hagen, Germany, tim.vorderbrueck@fernuni-hagen.de

[2] Institute for Medical Biometry and Medical Informatics, University Medical Center Freiburg, Freiburg, Germany, holger.stenzhorn@uniklinik-freiburg.de

according to its LHS, be possible applied at a superior node in a constituency tree. Note that only scores are modified or propagated which are assigned to rules which are not assumed to be correct ($gqs < h$).

2. Spotting the Error in the Generation Tree: The *gqs* already results in a good approximation for identifying an incorrect rule. However this method also has a drawback in that the identified rules are not related to any input structure. This is obviously an important information for the grammar developer if (s)he wants to know why no output has been generated for a specific input structure. Furthermore this information can potentially be necessary to automatically correct the error (which is planned for future work). Hence we additionally try to determine for each input structure leading to a generation failure the most probable location (node) in the constituency tree where the error occurred and use this information to calculate a local rule quality score (i.e., a score which relates to a certain input structure). The identified node is supposed to be associated to the LHS category of the erroneous rule [3].

Naturally, since positive trees do not lead to a generation error, for spotting the erroneous nodes only the negative trees have to be examined. An error is defined for each negative tree separately which means that different errors can relate to different constituency trees. Like the determination of potential erroneous rules there is again the possibility to employ either the maximum or the minimum tree where both methods have been evaluated. To spot the error location, each node in a negative tree is assigned to its node quality score (*nqs*).

For the calculation of the *nqs* the following two aspects relevant to many machine learning approaches are taken into account:

1. How do the negative examples (i.e., negative trees) differ from the positive ones?
2. What do all negative examples (i.e., negative trees) have in common?

To account for the first aspect the probability of a node is determined in that a tree is negative if it contains this node (pair of category and rule): $q_1 = P(t \in T^- | (r,c) \in t)$ where the probability is estimated by the relative frequency. A node is assigned the maximum value of 1 if it occurs *only* in the negative and *never* in positive trees.

To account for the second aspect the probability is estimated in that a negative tree contains a given pair of category and rule: $q_2 = P((r,c) \in t | t \in T^-)$. A node is assigned the highest possible value of 1 if it occurs in *all* negative trees.

The *nqs* for a tree node (r,c) is then given as $nqs(r,c) = -q_1 q_2$. A node is considered to appear in a constituency tree if this tree contains a node associated to identical category and rule. Note that a leaf node of the incomplete constituency tree might not be associated with any rule. Such a node matches all nodes with identical category. The nodes associated with the lowest *nqs* are assumed to be erroneous, i.e., one of them is assumed to contain the LHS category of the erroneous rule.

3. Putting Both Types of Information Together: Finally, the *gqs* and the expected error location are combined to the *lqs*. Even if the error location in the constituency tree is not correctly determined by this algorithm, the actual error location (i.e., the node containing the LHS category of the erroneous rule) is often a sibling/child or parent of the indicated location. Thus, for determining the *lqs* of a rule, its *gqs* is weighted depending of the minimum possible distance in a constituency tree of that rule's LHS category from any node representing one of the indicated error locations using an exponential decay. If this distance could not be determined because the rule's LHS category is not reachable at all, the distance is set to the number of categories in the grammar.

[3] Note that this approach is not suitable for detecting rules with an incorrect LHS. In this case, only the *gqs* should be used instead.

3 Evaluation and Conclusion

Table 1. Erroneous rule is among the top 5/3/2 suggestions; for all examples/examples with both positive and negative trees, in percent (%).

For the evaluation, we randomly changed the path expressions [5] of a rule's RHS (right hand side) in the evaluation grammar and determined how often the erroneous rule appeared under the top five/three/two rules with the lowest *lqs*. The evaluation shows that the accuracy raises significantly if at least one positive constituency tree exists (see Table 1).

The described algorithm has been implemented in a plug-in for the grammar workbench eGram [3] which supports the GUI-based development of grammar rules for the grammar formalisms of the TG/2 [2] and XtraGen [4] NLG systems.

The automatic detection and correction of grammar errors remains a very difficult task but it is an important and necessary step towards creating NLG systems that are easy to deploy in real-world application scenarios with large amounts of rules.

ACKNOWLEDGEMENTS

We are especially obliged to Stephan Busemann for providing one of the authors with a research license of eGram and XtraGen. Furthermore we thank all members of our departments who contributed to this work.

REFERENCES

[1] J.Cussens and S.Pulman, 'Incorporating linguistics constraints into ILP', in *Proc. of CoNLL*, Lisbon,Portugal, (2000).
[2] S.Busemann, 'Best-first surface realization', in *Proc. of INLG*, Herstmonceux, UK, (1996).
[3] S.Busemann, 'eGram — a grammar development environment and its usage for language generation', in *Proc. of LREC*, Lisbon, Portugal, (2004).
[4] H. Stenzhorn, 'XtraGen. A NLG system using Java and XML technologies', in *Proc. of NLPXML*, Taipeh, Taiwan, (2002).
[5] T. v.d.Brück and S. Busemann, 'Suggesting error corrections of path expressions and categories for tree–mapping grammars', *Zeitschrift für Sprachwissenschaft*, **26**(2), (2007).
[6] A. Zeller, 'Locating causes of program failures', in *Proc. of ICSE*, Saint Louis, Missouri, USA, (2005).