Quantitative Evaluation of Explanation-Based Learning as an Optimization Tool for a Large-Scale Natural Language System

Christer Samuelsson

Swedish Institute of Computer Science
Box 1263,
S-164 28 KISTA
Sweden
christer@sics.se

Manny Rayner

SRI International,
23 Millers Yard,
Cambridge,CB2 1RQ,
UK
manny@cam.sri.com

Abstract

This paper describes the application of explanation-based learning, a machine learning technique, to the SRI Core Language Engine, a large scale general purpose natural language analysis system. The idea is to bypass normal morphological, syntactic and (partly) semantic processing, for most input sentences, instead using a set of learned rules. Explanation-based learning is used to extract the learned rules automatically from sample sentences submitted by a user and thus tune the system for that particular user. By indexing the learned rules efficiently, it is possible to achieve dramatic speed-ups.

Performance measurements were carried out using a training set of 1500 sentences and a separate test set of 100 sentences, all from the ATIS corpus. A set of 680 learned rules was derived from the training set. These rules covered 90 percent of the test sentences and reduced the total processing time to a third. An overall speed-up of 50 percent was accomplished using a set of only 250 learned rules.

1. Introduction*

> "When interacting in natural language it is easy to fall into assuming that the range of sentences that can be appropriately processed will approximate what would be understood by a human being with a similar collection of data. Since this is not true, the user ends up adapting to a collection of idioms --- fixed patterns that experience has shown will work." [Winograd & Flores 85, p. 129].

Many users of natural language systems tend to phrase themselves in the same way most of the time. The optimization technique described in this paper is based on this observation: If one can speed up the processing for this limited set of "typical phrases", one will save a lot of computing time. The idea is that one can bypass normal processing for most input sentences, instead using a set of learned rules, and thus vastly speed up the processing. This will be done paying the price of a small overhead when no learned rule proves applicable. The set of learned rules is extracted automatically, using explanation-based learning (EBL), from training sentences given by a user. By learning the rules from real user interaction, the set of rules is tailored so as to capture the user's way of expressing himself. The hope is that a comparatively small set of language constructions will account for the majority of the sentences actually submitted to the system. Once the rules have been learned, it is important to store them in a way that minimizes the search for applicable ones at run-time, that is to index the learned rules so that quick access is guaranteed.

In this paper we focus on the experiments carried out at SRI Menlo Park on the ATIS corpus, a large size test corpus extracted from real user interaction. We have tried to make the paper as self-contained as possible. Due to space limitations, however, we can only sketch out the main principles, and must refer to previous articles for the details. This is especially the case in sections 3 and 4. In section 2, we offer a description of what explanation-based learning means in the context of natural language processing and in section 3 we address the important issue of how to assure quick access to the learned rules. The architecture of the EBL module is shown in section 4 and in section 5 we describe the set-up of the experiments and report the results. In section 6 we summarize our experiences and reflections on the subject of applying EBL to natural language processing and point to questions that deserve further attention.

This is the third time we have applied explanation-based learning to a natural language processing system. The target system this time was the SRI Core Language Engine (CLE), a large scale general purpose natural language analysis system developed by SRI Cambridge [Alshawi *et al.*, 89], [Alshawi, forthcoming]. In previous works, we have applied the EBL technique to the syntactic analysis phase of another large-scale NL system, [Rayner & Samuelsson, 90], and to syntactic and semantic processing in the CHAT-80 system, [Rayner & Samuelsson, 89]. Here, EBL was used to bypass morphological and syntactic analysis, and semantic analysis up to and including generation of quasi-logical forms (QLF's).

* This work was sponsored by SICS, the Swedish Institute of Computer Science, and the greater part of it was carried out while the second author was resident there.

2. What EBL means in NL processing

Explanation-based learning is a machine-learning technique, closely connected to other techniques like macro-operator learning, chunking, and partial evaluation; a phrase we have found useful for describing the method to logic programmers is *example-guided partial evaluation*; a phrase that may appeal to a natural-language person is *grammar rule chunking*. Explanation-based learning is well-described in a number of articles [Mitchell *et al.*, 86], [Dejong & Mooney, 86] and [Minton *et al.*, 89], to which we refer the reader who wants to understand the general principles; here, we will only summarize briefly what EBL means in the context of natural-language processing. In section 2.1, we offer an intuitive description of the explanation-based learning method used and in section 2.2 we give an illustrative example of it using a toy NL system.

2.1 Informal description of the EBL scheme

Our implementation is based on the scheme that has now become the standard model for "EBL in Prolog", as exemplified by the systems described in [Hirsch, 87], [van Harmélen & Bundy, 88] and [Kedar-Cabelli & McCarty, 87].

We learn rules from training examples. In this domain, a training example is a string that can be derived from the grammar, together with a trace of the derivation: in terms more commonly used by natural-language experts, a sentence and its associated parse-tree. The basic idea is first to define a class of operational goals; by this, we mean the sub-goals which will be allowed to appear on the right-hand-side of learned rules. Having done this, a successfully processed example is generalized by, notionally, constructing the derivation tree, and then chopping off all the branches rooted in operational goals, "un-instantiating" variables bound by these goals; the leaves in the new, "generalized" derivation will be the conditions in the learned rule (and thus by construction operational), and the new root will be a more general version of the goal corresponding to the root in the example. In the simplest (one-level) version of the method, operational goals will coincide with lexical ones as in the toy grammar example of section 2.2.

We have extended the model in three important ways. Firstly, we allow for multiple levels of operationality: the goals that are operational at one level (and thus appear on the right-hand side of the learned rules) are target concepts at the next level down (and thus appear on the left-hand side here). In the implemented system, there are three levels of operationality, corresponding to the goals "prove string X is a sentence", "prove string X is a noun phrase (NP)" and "prove string X is a word". Thus rules at the top level define sentences as concatenations of words and NP's, rules at the second level define NP's as concatenations of words, and rules at the lowest level define words as lists of morphemes. The choice of levels was fairly arbitrary, and it is certainly conceivable, especially at the highest level, to consider other possible choices for operational goals. We have already experimented, for example, with defining relative clauses as operational [Rayner and Samuelsson, 89]. Secondly, we carry out generalization in two stages, first running the problem in a "dirty" representation which has been optimized for efficiency, to get a trace, and then using the trace together with a "clean" version of the same domain theory (see section 4.2). This method makes it much easier to produce generalizations from complex domain theories. Thirdly, we demonstrate that the use of domain-specific indexing methods can provide a satisfactory solution to the "utility problem". This is explained in detail in section 3.

2.2 Example of a derivation

We turn now to an example of single-level explanation-based generalization.

We transform our grammar into a Horn-clause theory by encoding it as a Definite Clause Grammar [Pereira & Shieber 85]; this involves adding two extra arguments to each non-terminal, which represents the input string before and after the non-terminal is parsed. Diagram 1 shows a toy example of such a grammar, closely based on the TALK program in the above-mentioned book. The ^ symbol represents lambda abstraction.

```
s(S) --> np(Agr,VP^S), vp(Agr,VP).                          (1)
np(3-s,NP) --> pn(NP).                                      (2)
np(Agr,NP) --> det(Agr,N1^NP), n(Agr,N1).                   (3)
vp(Agr,X^S) --> tv(Agr,X^VP), np(_,VP^S).                   (4)
vp(Agr,VP) --> iv(Agr,VP).                                  (5)
pn((PN^VP)^VP) --> [PN], {lex(PN,pn)}.                      (6)
det((X^S1)^(X^S2)^quant(Det,X,S1,S2)) -->
                                [Det], {lex(Det,det)}.      (7)
n(Agr,X^[N,X]) --> [N], {lex(N,n(Agr))}.                    (8)
tv(Agr,X^Y^[TV,X,Y]) --> [TV], {lex(TV,tv(Agr))}.           (9)
iv(Agr,X^[IV,X]) --> [IV], {lex(IV,iv(Agr))}.               (10)

lex(john,pn).
lex(a,det).
lex(cat,n(3-s)).
lex(loves,tv(3-s)).
lex(sleeps,iv(3-s)).
```

Diagram 1, toy grammar and lexicon

This grammar can parse a few trivial sentences like *John sleeps* or *John loves a cat*, and associate with each a corresponding expression in first-order logic; we will now show what happens when the second sentence is generalized with respect to the lexicon.

We first construct the normal derivation tree for the original sentence, or to be more exact for the proposition

```
s(quant(a,A,[cat,A],[loves,john,A]),[john,loves,a,cat],[]).
```

(The first argument - the logical form - is to be read as "An A such that [cat,A] is also such that [loves,john,A].")

The derivation tree will be as shown in diagram 2, where each node has been marked with the number of the clause resolved on at that point. We then want to generalize away the lexical information present. To do this, we perform the same series of resolution steps, but this time omitting all resolutions with unit clauses of the type lex(_,_). Here the operational goals coincide with the calls to lexical predicates. This will yield us the conditional derivation tree in diagram 3, where the assumptions have been written in bold-face.

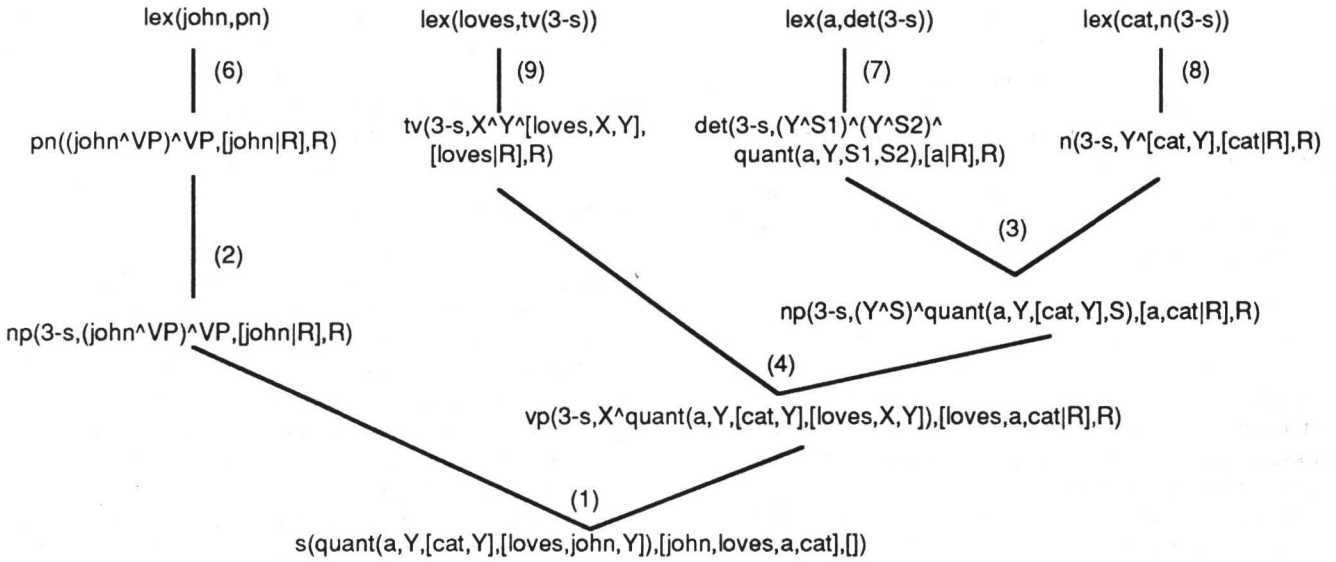

Diagram 2, derivation of "John loves a cat"

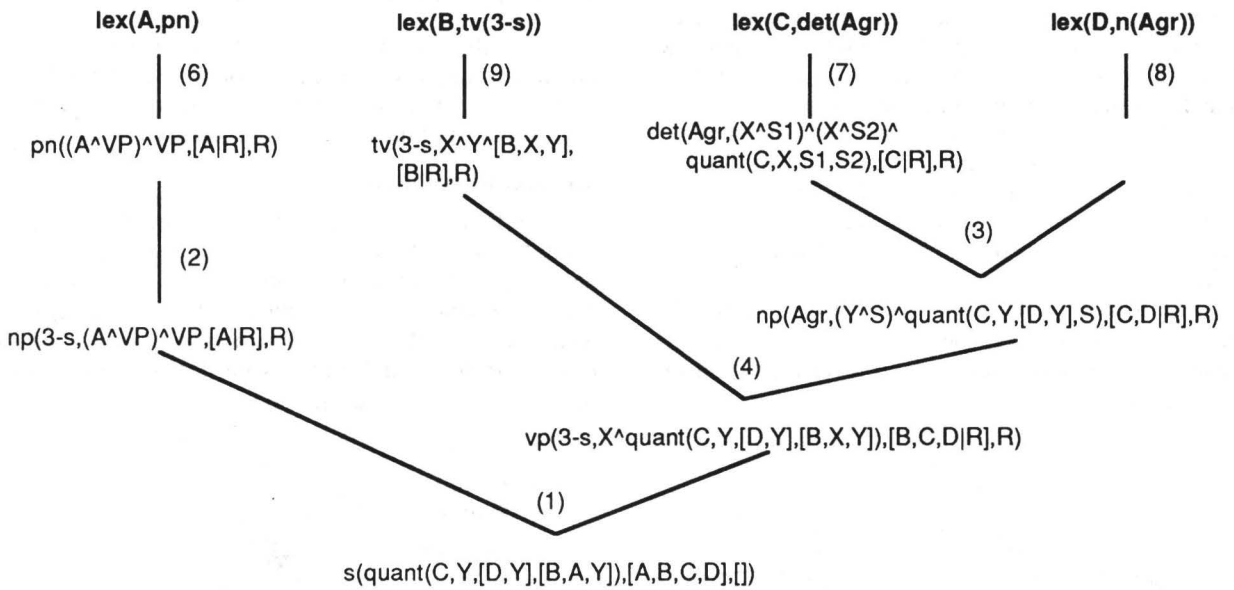

Diagram 3, generalized derivation tree

The suspended calls to operational goals are then collected to form the learned rule in diagram 4. Since the new tree represents a valid derivation for *any* values of the (meta-) variables A, B, C, D and Agr, it thus constitutes a proof of the learned rule.

```
s(quant(C,Y,[D,Y],[B,A,Y]),[A,B,C,D],[]) :-
        lex(A,pn),
        lex(B,tv(3-s)),
        lex(C,det(Agr)),
        lex(D,n(Agr)).
```

Diagram 4, generalized derived rule

Thus we are now in possession of a learned rule which can handle not only *John loves a cat*, but also many other sentences, for example *Mary hates the winter* and *Charlie has a Ferrari*, or could handle, had there been entries for these words in the lexicon.

3. Indexing the learned rules

Once we have derived a large set of learned rules, it will be too time consuming to simply try one after the other until we stumble upon the proper one. All computing time gained from using learned rules, instead of normal processing, will be devoured by costly linear search among the learned rules. This is the *utility problem* [Minton 88]. Our solution is to use a domain-specific indexing method to ensure quick access to the learned rules.

The obvious way to filter out most learned rules as inapplicable, is to inspect the syntactic categories of the

input string: all strings a particular learned rule can handle must match the sequence of lexical constraints in the rule.

In the following two sub-sections, we describe briefly two such indexing schemes. From timing studies we have concluded that the key indexing scheme is good for one-level rules and that the decision-tree indexing scheme is good for two-level rules. Therefore, we used the decision-tree indexing scheme for the sentence and noun phrase rules and the key indexing scheme for the morphological rules.

3.1 The decision-tree indexing method

The decision-tree indexing method stores the learned rules in a decision-tree, using a demand driven algorithm dispatching on lexical category to access them. Below we sketch out how the algorithm for finding a learned rule works; [Rayner & Samuelsson, 90] contains a more detailed, pseudo-code description of it.

To find a learned rule at run-time we do the following: We start at the root node in the decision-tree, and inspect the lexical category of the first input constituent. If the start node has an arc labelled with this category, we follow it to the child node. If it doesn't we fail. If we were successful, we then inspect the category of the next input word. If our current tree node has an arc labelled with this category, we follow it, and so we continue until all of the input words are consumed. For the last input word, we search for an arc to a leaf node. This node will be the reference to a learned rule.

It is important to realize that lexical ambiguity does not in general give rise to an exponentially growing search space; an incorrect guess of a word's lexical category will normally find no arc to dispatch on, and thus terminate immediately.

The rule derived from *John loves a cat* in section 2.2 will be indexed as rule42 in the diagram below. The other rules correspond to *John sleeps*, *John loves Mary* and *The cat sleeps* respectively.

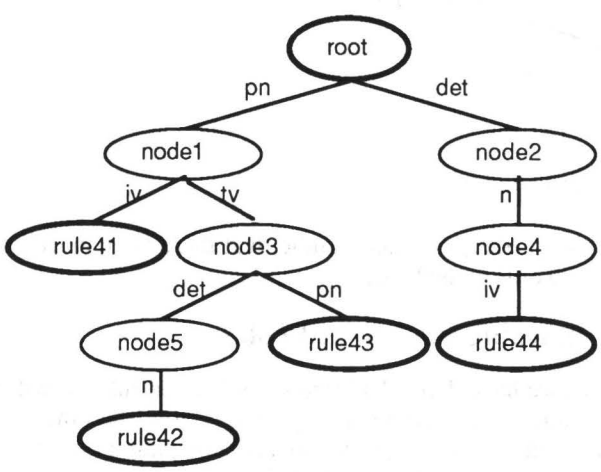

Diagram 6, one-level decision-tree indexing scheme

When using multiple rule-levels, we extend the scheme by including arcs labels with "pushes" to trees indexing the lower level rules. These lower level trees will have "pop" arcs back to the tree that called them. Apart from the slight modifications that these "pops" and "pushes" introduce, the scheme will work as in the one-level case

In our implementation we used one tree for top-level rules and another one for noun phrase rules. To avoid repeating work, it proved useful to include a well-formed sub-string table for the derived noun phrases.

3.2 The key indexing method

The key indexing scheme scheme first maps the lexical constraints of the learned rule to a sequence of key letters. These letters are chosen such that distinct lexical constraints are associated with distinct key letters. In our example, assume that pn goes to **e** (for *eigen-name*), tv goes to **v** for *verb*, det goes to **d** and n goes to **n**. Then, the sequence of letters is collapsed into an atomic key, in our case **evdn**. The learned rules are stored in a table indexed by such keys, utilizing the "first functor" indexing mechanism of most Prolog systems. A more thorough description and analysis of this indexing scheme can be found in [Rayner & Samuelsson, 89].

4. Design of the EBL module

We will now describe briefly how the ideas in the previous section were realized for the SRI Core Language Engine (CLE). It was interesting to note that the CLE supplied most of the primitives necessary for our purposes.

4.1 Overall architecture

The EBL module can naturally be divided into its compile-time and run-time parts. The learning component is the compile-time part. For convenience, we will sub-divide it into two smaller components. These are the *generalizer*, which performs the actual extraction of learned rules and the *rule compiler*, which indexes the learned rules to ensure quick access to them at run-time.

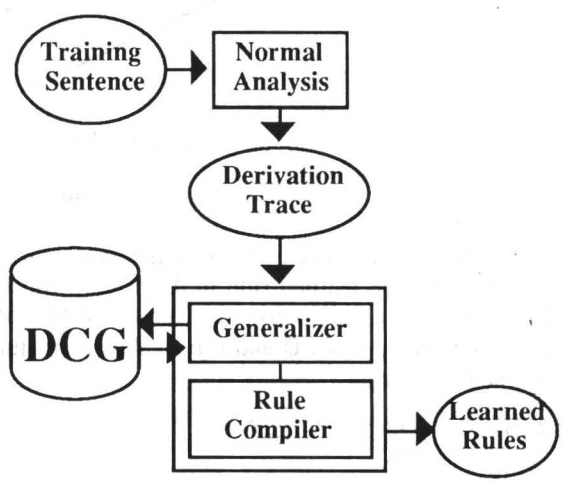

Diagram 8, the learning component.

The run-time component of the system is the *pattern matcher*, which processes input sentences using the set of learned rules or establishes that no applicable combination of learned rules exists.

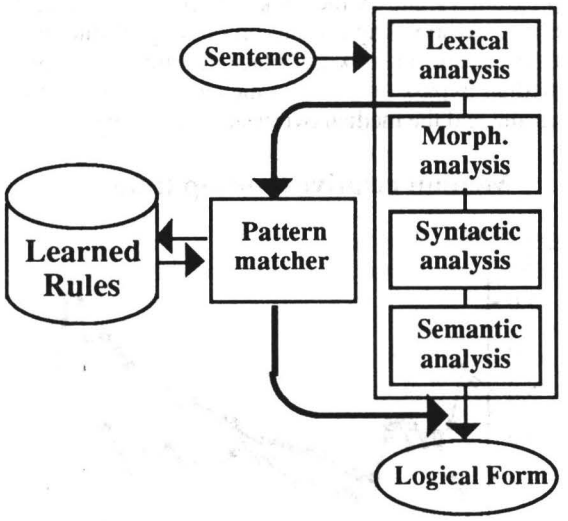

Diagram 9, the run-time component.

We now examine the components in turn.

4.2 The generalizer

The generalizer is a simple Prolog meta-interpreter and the generalization is from a computational perspective essentially the parsing of a sentence with a DCG. Usually, this means that care has to be taken to ensure that parsing efficiency is acceptably high, and even more importantly that infinite recursions are not caused by left-recursive grammar rules. However, the generalization is not driven simply by the target goal and, by recursion in the generalization algorithm, successive sets of sub-goals; it is guided by the derivation trace of a training example. Thus the parsing is actually deterministic. The sentence is first subjected to the normal analysis component to find a derivation trace, which is then used to guide the DCG parser used by the generalizer. The top-level is thus schematically:

```
learn_a_rule(Sentence,Rule,SubRules) :-
        ...
        normal_processing(Sentence,LogicalForm,Trace),
        generalize(Trace,Rule,SubRules),
        ...
```

The Trace is a tree of rule-ids encoding the derivation of LogicalForm from Sentence. Each rule-id refers to the DCG rule used at that resolution step. The DCG rule will have a LHS and a RHS. The generalizer re-does the resolutions of the derivation tree. With each rule-id it does one of the following: If the corresponding LHS is a non-operational goal, it expands it according to the DCG rule and calls itself recursively for each RHS goal. If the rule is a production rule for NP's, or if it is a morphological rule, it saves the LHS of the DCG rule to be added to the RHS of the main learned rule and returns to top-level to learn a sub-rule for the LHS goal of the DCG rule, guided by the remaining sub-tree rooted in this rule-id. If the DCG rule is a lexicon look-up, the LHS goal of the DCG rule is saved and added to the RHS of the learned rule. This is basically the same generalizer as the one presented in [van Harmélen & Bundy, 88].

4.3 Quick access to learned rules

If the pattern-matcher is to be able to use the learned rules efficiently, they must be indexed in order to ensure quick access to them. This is what the rule compiler does. In general, when adding a rule to the data base of learned rules, it first checks to see if the new rule is subsumed by any previously added rule, or if is identical to one, in both cases the new rule is not added. Also, it determines if the new rule subsumes some old rule already present among the learned rules, in the latter case the old rule is removed and the new one takes its place.

The pattern matching process is divided into two steps:

In the first step, words are built from word stems and affixes. The rules performing this task are accessed by key indexing, since a one-level rule consumes its entire input in one go.

In the second step, phrases are constructed from words. The rules for doing this are accessed using the decision-tree indexing scheme, as described in section 3.1.

5. Performance studies

The main results of this paper concern the relationship between the performance and the size of the set of learned rules, when both the rules and the test data are acquired from real user interaction. Such experiments were carried out at SRI Menlo Park using the ATIS corpus.

5.1 Set-up of the experiments

Here we describe the set-up of the experiments carried out at SRI Menlo Park. They were conducted to determine how the coverage, the rule access time and the overall performance gain depend of the number of learned rules.

The coverage is defined as the fraction of a test corpus successfully handled by the EBL module. The performance gain is simply the ratio of the total processing time for the test set using normal processing, divided by the total time using the learned rules.

In these experiments, we did not bypass morphological analysis. The main reason for this was that the morphology is multiplied out into the lexicon in the SRI Menlo Park version of the CLE.

We worked with the ATIS corpora, large sets of sentences extracted from user interaction using "Wizard of Oz" techniques. We from these collected a large set of sentences that the CLE could attribute with a semantic interpretation. We then split the set into two sub-sets. The larger, consisting of 1563 sentences, was used as a training set. The other, with a size of one hundred exactly, became the test set. The sets were split by randomly choosing 100 numbers. First we learned a set of rules from the training set. The 1563 sentences yielded 680 learned rules. We then applied the EBL run-time component, equipped with this set of learned rules, to the training set and ordered the rules according to their frequency of use, dividing them into groups of 20 rules each according to their usefulness. Thus rules which could be applied to many sentences (and therefore, intuitively, represented constructs "typical" of the domain) were added first.

When measuring the run-time performance, we started with no learned rules and added each group of rules in turn to the rule data base. Before adding a new score of rules we determined the coverage and measured the success- and failure times of the EBL module, using the test set. We then continued to add learned rules and repeated the measurements until all learned rules had been added. For comparison, we performed timing studies on normal processing. We could then extract the desired results from the experimental data. Training and performance measurements were carried out in "batch mode" (i.e no learned rules were added during training) primarily to make it simpler to get the typicality figures.

The timing studies were carried out using QUINTUS 2.5 running on a SPARC1+ workstation with a 32 MByte local disc. The Prolog work-space was set to the maximum of what the Unix system could provide, in order not to have rule space expansions corrupt the timing results. Garbage collect was enforced before, and disabled during, every measurement. Predicate calls of short duration were re-run a multitude of times, the entire CPU time measured and divided with the number of runs.

5.2 Experimental results

It is interesting to note that, at compile-time, 680 learned rules, which together amount to 470 kByte, could account for 1563 sentences. From the run-time part of the experiment we find that the coverage reaches 90 percent at 680 rules, as shown in diagram 10.

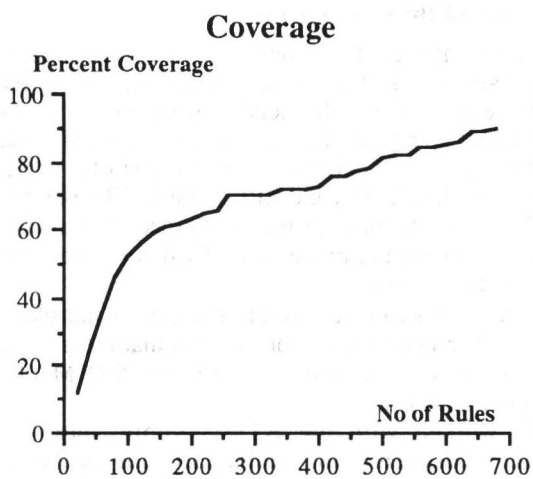

Diagram 10. The coverage as a function of the number of learned rules

The coverage swiftly rises to 60 percent for 150 learned rules and then seems to increase slowly, but steadily, with about 0.75 percent additional coverage for every ten rules. Presumably, rules which were sufficiently atypical would not be worth adding. However, diagram 12 shows quite clearly that performance was still going up even when the last of the 680 rules were added. It is very probable that even better figures could have been achieved by training with a larger set of examples, extracting a correspondingly larger set of rules.

Diagram 11 shows the look-up time normalized by the time for corresponding normal processing. White diamonds indicate successful look-up, black failure. With 680 rules, the median bypass time is 15 times less than that of normal processing and the median overhead is 5 percent.

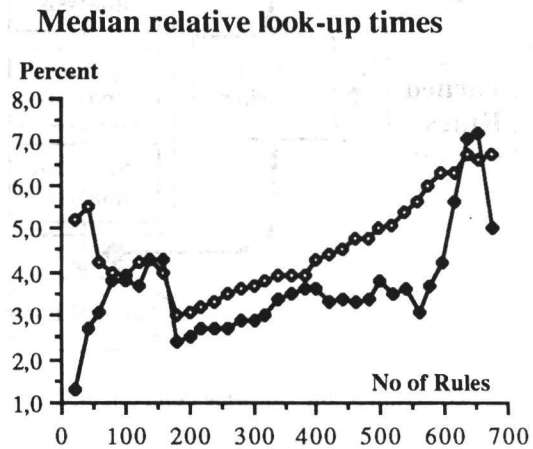

Diagram 11. The EBL look-up times as a function of the number of learned rules

There is a close to linear increase in look-up time with the number of learned rules. The large apparent variation of the failure times after 600 rules is probably not statistically significant, since there are only a dozen or so outstanding sentences left in the test-set at this stage that are not already covered by learned rules.

The EBL look-up times are small compared to normal processing times - the median look-up times lie between 1 and 7.5 percent of normal processing time. This results in a substantial overall speed-up, as can be seen in diagram 12.

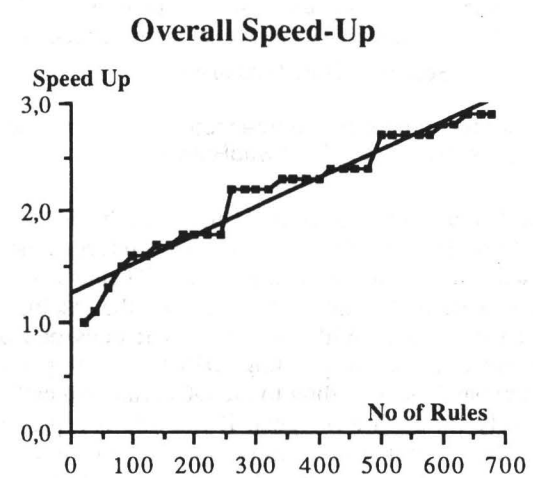

Diagram 12. The overall processing times as a function of the number of learned rules

The overall speed-up increases slightly sub-linearly in the number of learned rules. The system is twice as fast with 250 learned rules and with 680 rules the system runs three times faster.

6. Summary, conclusions and further directions

On the basis of our experiences this far, we think there are good reasons to take EBL seriously as a practical and generally applicable way of optimizing natural language systems; the speed-ups achieved are considerable at a low overhead. Even more importantly, it has been possible to apply the EBL method to all NL systems attempted this far, even though one had several characteristics undesirable from this point of view.

An important question, when dealing with some systems, is the extent to which it is possible to compress the generated rules; one idea is to keep only the derivation traces and redo the resolution steps at run-time, since generalization is deterministic and has been observed to be about as fast as pattern-matching. From this we conclude that most savings result from guiding the search. Since one is essentially trading space for time, this may be an economical compromise.

Another, rather obvious, thing to do is to use the learned rules "backwards", that is for paraphrasing, by constructing an indexing scheme for logical forms.

Two interesting software engineering challenges are to integrate the EBL scheme more closely with the target system to allow the normal analysis component to use partial results from the EBL module and vice versa, and to allow incremental adaption of the system by letting the learning component run as a background process. One will then also have to deal with the problems connected with user interaction and utility analysis.

Finally, we mention briefly a line of research that we have just begun to investigate, namely to incorporate the learned rules into a probabilistic language model of the kind used by speech recognition systems. Although our work to date is still only at a preliminary stage, it appears that this idea may potentially be very promising.

Acknowledgements

The performance measurements described in section 5 could not have been carried out without the help and support of many people at SRI Menlo Park. We are most grateful for being given this opportunity. In particular, we would very much like to thank Robert C. Moore, Douglas B. Moran and John Dowding for all their help and for all the time and effort they spent making these experiments possible. We would also like to thank Hiyan Alshawi of SRI Cambridge for valuable advice.

References

[Alshawi, forthcoming] Alshawi, H. (ed.). *The Core Language Engine*, MIT Press, Cambridge, Mass., to appear 1991.

[Alshawi *et al.*, 89] Alshawi, H., Carter, D.M., van Eijck, J., Moore, R.C., Moran, D.B., Pereira, F.N.C., Pulman, S.G and Smith, A.G. "Research Program in Natural Language Processing: Final Report", SRI Technical Report, 1989.

[Dejong & Mooney, 86] DeJong, G. F. and Mooney, R. "Explanation-Based Generalization: A Alternative View", *Machine Learning*, 1(2) pp. 145-176, 1986.

[van Harmélen & Bundy, 88] van Harmélen, F. and Bundy, A. "Explanation-Based Generalization = Partial Evaluation (Research Note)", *Artificial Intelligence* 36, pp. 401-412, 1988.

[Hirsh, 87] Hirsh, H., "Explanation-Based Generalization in a Logic-Programming Environment", *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 221-227, Milano, 1987.

[Kedar-Cabelli & McCarty, 87] Kedar-Cabelli, S. T., and McCarty, L. "Explanation-Based Generalization as Resolution Theorem Proving", *Proceedings of the Fourth International Workshop on Machine Learning*, Irving, CA, Morgan Kaufmann, 1987.

[Minton, 88] Minton, S., "Quantitative Results Concerning the Utility of Explanation-Based Learning", *Proceedings of the Seventh AAAI*, pp. 564-569, Saint Paul, Minnesota, 1988.

[Minton *et al.*, 89] Minton, S., Carbonell, J.G., Knoblock, C.A., Kuokka, D.R., Etzioni, O. & Gil, Y., "Explanation-Based Learning: A Problem-Solving Perspective", *Artificial Intelligence*, (40):11-62, 1989.

[Mitchell *et al.*, 86] Mitchell, T., Keller, R. and Kedar-Cabelli, S. "Explanation-Based Generalization: A Unifying View", *Machine Learning*, 1(1), pp. 47-80, 1986.

[Pereira & Shieber, 85] Pereira, F.N.C. and Shieber, S. *Prolog and Natural Language Understanding*, CSLI Lecture Notes, University of Chicago Press, 1985.

[Rayner, 88] Rayner, M. "Applying Explanation-Based Generalization to Natural-Language Processing", *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1988.

[Rayner & Samuelsson, 89] Rayner, M. and Samuelsson, C., "Applying Explanation-Based Generalization to Natural-Language Processing (Part 2)", SICS Research Report R89015, 1989.

[Rayner & Samuelsson, 90] Rayner, M. and Samuelsson, C., "Using Explanation-Based Learning to Increase Performance in a Large-Scale NL Query System", *Proceedings of the Third DARPA Workshop on Speech and Natural Language*, Morgan Kaufmann, 1990.

[Winograd & Flores, 85] *Understanding Computers and Cognitions*, Winograd, T. and Flores, F., Norwood NJ: Ablex, 1985.