Using Explanation-Based Learning to Increase Performance in a Large-Scale NL Query System

Manny Rayner, Christer Samuelsson
Swedish Institute of Computer Science
Box 1263
S-164 28 KISTA, Sweden

1. Introduction

Explanation-based learning (EBL) 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*. The basic ideas of the method are well-described in an overview article which recently appeared in *Artificial Intelligence* [1], to which we refer the reader who wants to understand the theoretical principles; here, we will only summarize briefly what EBL means in the context of natural-language processing. A detailed presentation can be found in [3] and [4].

What EBL tries to do in the context of NLP is exploit the well-known observation that users of an NL interface tend to ask the same types of question most of the time; lacking exact figures, it seems reasonable to guess that at least 80% of all questions posed to a given specific NL application will be accounted for by the 100 most common question-types. If one had some simple way of automatically identifying these "common" question-types, it would be possible to win a great deal of efficiency by by-passing the normal parsing mechanism in all but the hard cases.

Unfortunately, it is not feasible simply to add 100 extra rules to the grammar, since the common question-types vary depending on the application: a construction which occurs constantly in one domain may hardly exist in another. Something more sophisticated is required, which is capable of taking examples of common types of query and synthesizing the corresponding special rules. This is exactly what the EBL method offers. The normal route through the parser is extended with an *EBL bypass*, which contains special rules for efficient processing of common queries; these rules are not coded by the programmer, but rather are produced automatically by inspecting the solutions to previously posed queries of the same type. EBL can thus best be thought of as a way of automatically tuning an NL system to produce increased performance in a particular domain.

The EBL module can consequently be divided into its compile-time and run-time parts. The compile-time part extracts the learned rules from sample queries: the central component is the *generalizer*, which in our version is essentially a type of Prolog interpreter. The run-time part then applies the rules to input queries, the compile-time system having previously indexed them so as to make them readily accessible to some kind of table look-up facility. In [4], we demonstrated, using examples taken from an application of the EBL method to CHAT-80 [2], that table look-up methods of this kind can be implemented quite simply in Prolog with a minimal overhead.

In the current paper, we describe the results of experiments carried out at IBM Nordic Laboratories, where the EBL method was used on a large-scale NL query interface prototype. The EBL module learns a "two-level" set of special grammar rules; the top-level rules for S's treat NP's as primitive, and these are supplemented by a second set of rules for common NP's. Both types of rules are learned automatically in the way described above. In the remainder of the paper, we first give a brief overview of the IBM system, concentrating on the features that presented problems for the implementation of the EBL process; we then describe the architecture of the EBL module's compile-time and run-time components. In section 4, we present our experimental results, which indicate fairly unambiguously that the EBL method gives a real, and quite substantial, speed-up of the system as a whole; the final section contains our conclusions together with suggested directions for further research.

2. Relevant characteristics of the target NL system

The system used for our experiments was a large-scale NL query prototype, implemented in Prolog, which is intended to provide good coverage of a fairly large portion of English. The main components perform the tasks of parsing, semantic interpretation, paraphrasing and database query generation; since the first of these is both the "cleanest" and by far the most time-consuming, we decided only to attempt to apply EBL to this phase of the process. We will thus concentrate exclusively in the following description on the grammar formalism, grammar and parser. As explained in [4], the main difficulties derive from the fact that our implementation of the EBL method requires the grammar to be reduced to a set of Horn-clauses: in our earlier experiments with CHAT-80, this was fairly simple, and only involved some minor editing of the code. Here, however, the gap between the grammar and an equivalent "clean" version was non-trivial. This was much more important than the mere increase in its size (~1000 rules, as opposed to 150 for CHAT-80), which in fact caused no problems at all.

The two major hurdles with regard to the grammar formalism were its non-standard treatment of features and movement. The basic feature operation is not unification, but priority merge: movement is handled not by gap features, but rather by "non-restrictive" rules, in which more than one non-terminal can occur on the left-hand side of the rule as well as the right. Partly due to this, an unusual parsing mechanism is used, in which extra-logical predicates (especially "assert") play an integral part. To give the flavour of the formalism, the following is a slightly modified version of a typical non-restrictive rule, in this case intended to cover free relatives like the one in "John mentioned a book yesterday which you should read":

```
  s(2,prm=1,fpe(2)) &
temp_advp(1,trm=1,fpe(1)) ->
temp_advp(dng=0) & s(rel=1)
```

The rule reverses the sequence of temporal adverbial and relative clause, in effect transforming the sentence into "John mentioned a book which you should read yesterday". The "2" in the first argument position in the left-hand "s" indicates that its features are to be inherited from those in the second constituent on the right-hand side; "prm=1" means that the "prm" feature in the inherited set will if necessary be overridden and set to 1.

As we shall see in section 3.3, the parsing mechanism turns out to be irrelevant for our purposes; all that is significant is the grammar, viewed as a declarative description. We shall accordingly conclude our description of the target system at this point.

3. Design of the EBL module

3.1 Overall architecture

As explained above, the EBL module can naturally be divided into its compile-time and run-time components, which we will further describe in the following sections. For convenience, we will sub-divide the compile-time system into three smaller components. These are the *grammar pre-processor*, which converts the grammar into a suitable pure Horn-clause representation; the *generalizer*, which performs the actual extraction of learned rules; and the *simplifier*, which attempts to reduce them in size by removing unnecessary calls. We now examine each of these in turn.

3.2 The grammar pre-processor

This component performs the job of converting the original grammar into a pure DCG form, in which the first argument of each non-terminal contains a term encoding its derivation history; the motivation for this additional condition will be apparent in the next section. The only non-trivial part of the process, from our viewpoint, was dealing with unrestricted rules, since the other problems had already been taken care of by the normal grammar compiler. However, it turned out that this problem could also be solved simply, by first representing the unrestricted rules in Pereira's Extraposition Grammar (XG) format; using the XG compiler from [2], it is then straight-forward to turn the grammar into pure Horn-clauses. Conceptually, the XG compiler turns the unrestricted grammar into a DCG, where each non-terminal is given an extra pair of arguments (the "extraposition list"), to pass around the additional left-hand constituents. To give an example, the rule quoted at the end of section 2 is represented (again in a slightly edited form) as follows:

```
s(s(rule112,S,T),Feats_1,Sem_1,
  X_in,x(nogap,nonterminal,
         temp_advp(T,Feats_2,Sem_2),
         X_out)) ->
temp_advp(T,Feats_3,X_in,X_next),
{get_feature(Feats_3,dng,0)},
s(S,Feats_4,X_next,X_out),
{get_feature(Feats_4,rel,1),
 put_feature(Feats_3,prm,1,Feats_1),
 put_feature(Feats_4,trm,1,Feats_2)}.
```

The DCG produced can potentially contain left-recursive rules. However, we shall see in the next section that this causes no problems, since it is not used for normal, unrestricted parsing; the non-terminating branches in the search space can thus never be entered.

3.3 The generalizer

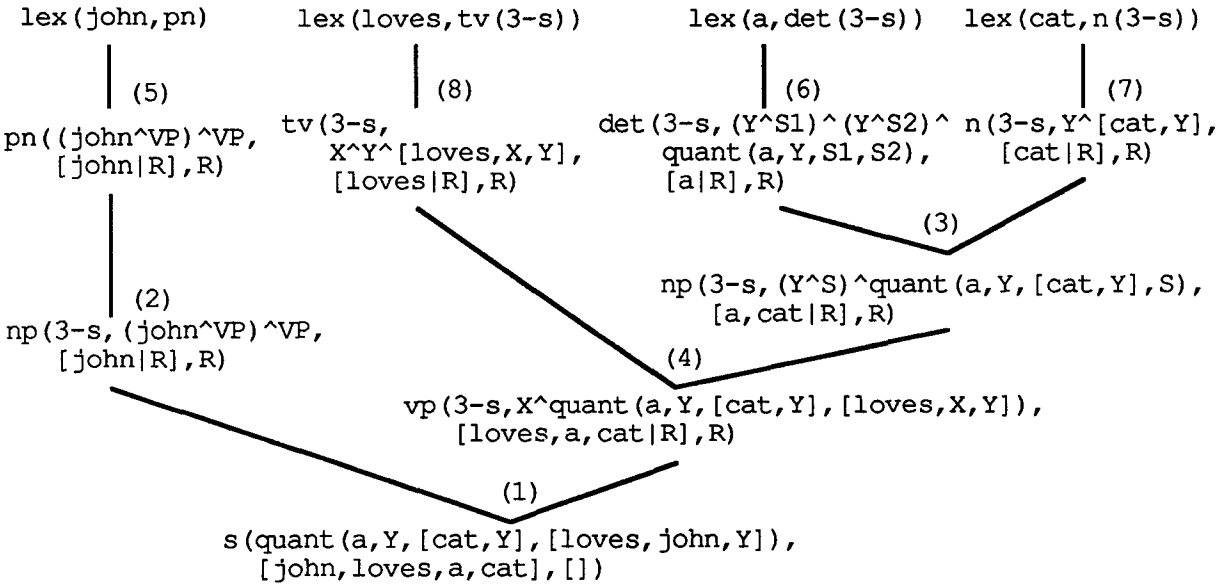
Since a detailed description of the generalizer can be found in [4], we will restrict ourselves here to an example and a brief overview. The basic idea is first to define the class of *operational goals*; by this, we mean the goals which will be allowed to appear on the right-hand-side of learned rules. Having done this, a successfully processed example is generalized by (notionally) constructing a derivation tree for it, and then chopping off all the branches rooted in operational goals; the leaves in the new, "generalized" derivation will be the conditions in the learned rule (and thus by construction operational), and the root will be a more general version of the goal corresponding to that in the example. In the simplest (one-level) version of the scheme, operational goals will coincide with lexical ones: thus generalization will be at the word level. An illustrative example is shown in diagram 1.

A slight refinement is to allow non-lexical operational goals, in particular ones corresponding to NP's. The basic method can now be applied recursively, first to the proof tree corresponding to the entire example, and then to each tree rooted in an operational NP goal; in the latter case, the operationality criterion is once again lexical. This results in the acquisition of two sets of rules, corresponding to the two different operationality criteria: the top-level rules construct S's from NP's and lexical items, and the second-level ones construct NP's from lexical items alone.

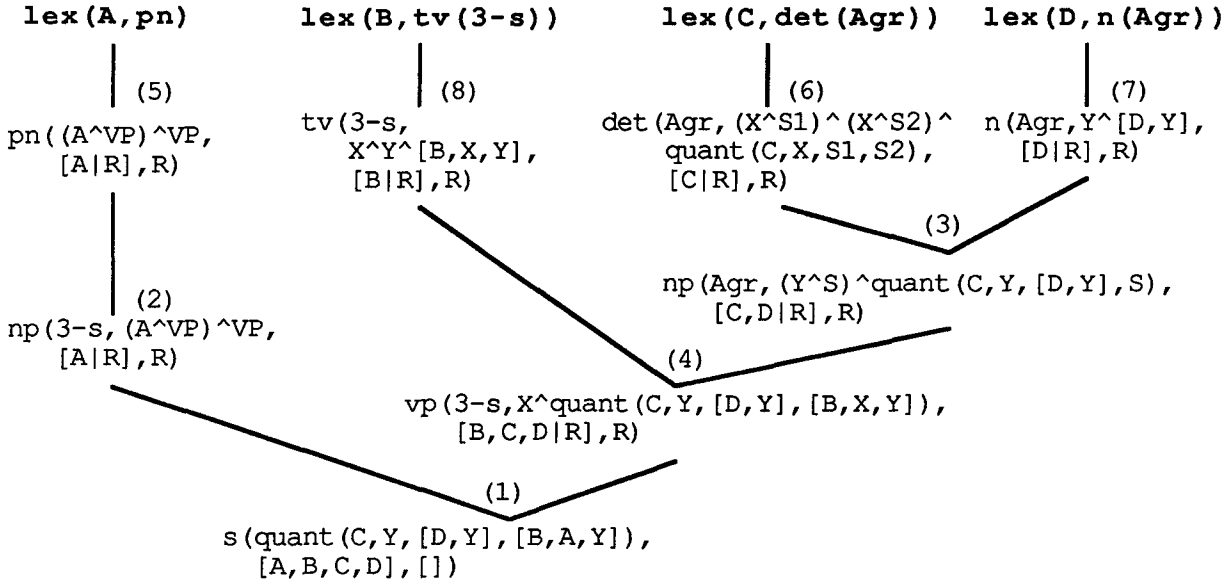
```
s(S) --> np(Agr,VP^S), vp(Agr,VP).                          (1)
np(3-s,NP) --> pn(NP).                                      (2)
np(Agr,NP) --> det(Agr,N1^NP), n(Agr,N1).                   (3)
vp(Agr,X^S) --> tv(Agr,X^VP), np(_,VP^S).                   (4)
pn((PN^VP)^VP) --> [PN], {lex(PN,pn)}.  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

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

Grammar and lexicon



Derivation of "John loves a cat"



Generalized derivation tree

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

Generalized derived rule

Diagram 1. Example application of EBL to a toy logic grammar.

The generalizer is basically a Prolog meta-interpreter, which means that generalization is from a computational perspective essentially the parsing of a query with a DCG; 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. Luckily, there is a simple and uniform way to solve this problem, by exploiting the fact that the first argument in each rule has been set up to hold the derivation history. The query is first run through the normal, "dirty" grammar, to find the intended instantiation of the derivation argument; this is then used to guide DCG parser used by the generalizer, effectively making the "parsing" deterministic. The top-level is thus schematically:

```
extract_rule(Query,Rule) :-
dirty_parse(s(Tree,_,_),Query),
generalize(clean_parse(s(Tree,_,_),
                       Query),
           Rule).
```

where the predicate names have their obvious meanings.

3.4 The simplifier

The purpose of this module is to attempt to reduce the size of learned rules, in particular calls to feature-manipulation primitives; these make up most of the body of typical rules with on average about 50 calls per rule. The basic mechanism is to take each feature-value, and trace its update history backwards through successive updates. Dividing feature-manipulation into "gets" and "puts", we can optimize in at least the following ways:

- Removing "gets" which can already be seen at compile-time to succeed. Since learned rules are compositions of normal ones, this case occurs when one component rule "gets" a feature that an earlier component has "put".

- Removing duplicate copies, when the same "get" occurs more than once in the rule.

- Reordering the rule body so that all structure-building takes place at the end: this ensures that structure will only be built if the rule succeeds.

If features were only used for syntax, it would also be possible to perform a further kind of optimization for S-level rules; having traced each "get" back through the chain of "puts" ending in the feature set it accesses, we could then remove the "puts" altogether. This would represent a very considerable reduction in average rule-size. Semantic processing in the target system is unfortunately not structured so as to allow this, but we think it likely that the method could be applicable in other, similar, contexts.

The following pseudo-code characterizes the simplification algorithm:

Phase 1

1. Combine "gets" and "puts" accessing the same feature set into groups. Replace each group with a corresponding call to `get_group` or `put_group`.
2. Collect all calls to structure-building routines.

Phase 2

Go through the body of the rule, passing an alist of annotations; this is used to replace or simplify calls to "get_group". The alist associates with each feature set a history of its derivation. This is one of

- `primitive(Constituent)` - the feature set is the one associated with `Constituent`.
- `update_from(Old_features,Update_set)` - the feature set was derived from `Old_features` by the chain of updates `Update_set`.

For each literal `L` in the rule body, do one of the following.

i) If `L` is of the form `put_group(Old,Updates,New)`, then add a suitable entry to the alist, constructed from `L` and the derivation history of `Old`.

ii) If `L` is of the form `get_group(Feature_set,Access_list)`, replace it with a literal of the form `get_group(Original,Access_list_1)`, where:

   a) `Original` is the base of the update chain that `Feature_set` belongs to.

   b) `Access_list_1` is derived from `Access_list` as follows: for each element `F=V`, if `F=V1` is in the list of updates, unify `V` with `V1` and throw away `F=V`.

iii) If `L` is of any other form, keep it unaltered.

Phase 3

1. Remove duplicate calls.
2. Re-expand calls to `get_group` and `put_group`.
3. Add structure-building calls to the end of the rule body.

3.5. The pattern-matcher

Since the learned rules acquired by the generalizer in effect comprise a specialized grammar, it would be possible to apply the normal parsing mechanism to them. However, this fails to exploit the grammar's unusually simple structure: the depth of a derivation-tree cannot exceed two, and NP is the only non-lexical category. Thinking about the problem in this way should make the pattern-matcher's construction easy to understand. The rules are compiled into a trie-structure, indexed by constituent category; this can either be "NP", or some lexical category. The pattern-matcher then locates potentially suitable rules by a kind of non-deterministic LR parsing method, driven by the trie-structure and otherwise optimized to exploit the peculiarities of the situation; a well-formed substring table is used to remember previously located NP's. Our tests indicate that this method is at least five times faster than the target system's normal parser.

The following pseudo-code characterizes the algorithm. Positions in the input string are marked from `0` to `*end*`;

trie-root denotes the root-node of the trie-structure; pointer marks the place we have reached in the input string, trie_node the current position in the rule trie, and nps the sequence of NP's so far located between 0 and pointer. We assume that lexical analysis has already been performed, so that we can discover by a suitable look-up operation whether or not there is an item of a given lexical category at a given location in the input string.

Pattern-matching algorithm

1. Set pointer to 0. Set trie-node to *trie-root*.
2. Set category to the lexical category of the item at pointer.
3. Non-deterministically do one of:
   a) If there is a trie arc from trie-node to next-node triggering on category then set trie-node to next-node. Bump pointer and go back to 2.
   b) If there is a trie arc from trie-node to next-node triggering on "NP", and there is an NP from pointer to next-pointer, set trie-node to next-node, set pointer to next-pointer, push the found NP onto nps, and go back to 2.
   c) If pointer = *end*, and trie-node is a leaf of the trie marked with a rule, then try to apply it to the whole input string, if necessary looking up NP's in sequence from nps.

The subroutine for finding NP's is similar, though slightly simpler; the variable and constant names correspond in the obvious way to those in the first algorithm.

To find an NP from pointer to next-pointer:

1. If the well-formed substring table records that NP's have been searched for at pointer, pick one non-deterministically and return, else
2. Set NP-pointer to pointer. Set NP-trie-node to *NP-trie-root*.
3. Set NP-category to the lexical category of the item at NP-pointer.
4. Non-deterministically do one of:
   a) Find a trie arc from NP-trie-node to NP-next-node triggering on category. Set NP-trie-node to NP-next-node. Bump NP-pointer and go back to 3.
   b) If there is a reduction rule at NP-trie-node, attempt to apply it to the segment of the input string joining pointer to NP-pointer, and record the result in the well-formed substring table. Then return.
   c) If NP-pointer = pointer and there are no alternatives left, record in the well-formed substring table that NP's have been searched for at pointer, and return with failure.

4. Results

A proper evaluation of performance gain due to the EBL bypass is impossible without a large statistical sample of typical user interactions with the target system; at this stage of the project, such data is unfortunately not available. Our preliminary performance measurements have been based on a corpus of 31 queries of distinct syntactic type, in length varying between 3 and 14 words; the histogram in diagram 2 summarizes the distribution of the speed-up factor over this set. The speed-up factor was defined as the ratio of EBL look-up to parsing for sentences where an applicable rule existed. It averaged slightly over 30, and as shown in the diagram exceeded 10 on all queries. The average look-up overhead on sentences for which no applicable rule existed was less than 3%. One of the few disappointments of the project was however the poor performance of the simplifier, which was unable to achieve better than an average 20% reduction in rule size; this appeared mainly to be due to the necessity to keep all feature sets for possible later use in semantic interpretation.

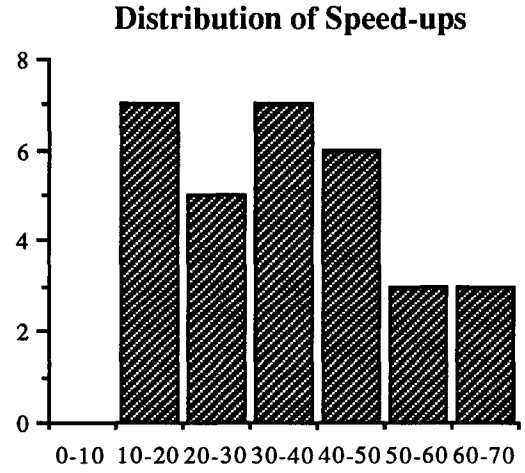


Diagram 2. Distribution of speed-ups due to EBL bypassing.

The following transcript of a short session with the system illustrates the EBL module in action. Input sentences are shown in bold-face, and comments in italics. Note that the glosses for acquired rules are only very approximate, and omit nearly all features.

```
EBL bypass initialized, no rules.
Does Iceland export fish?
Bypassing.
No match.
Adding a top level rule.
  "S -> does NP TV NP?"
Adding 2 second level rules.
  "NP -> Name" and
  "NP -> N:[mass=y]"

Is the Vip Club a small
organization?
Bypassing.
```

```
No match.
Adding a top level rule.
  "S -> is NP NP?"
Adding 2 second level rules.
  "NP -> the Name" and
  "NP -> DET ADJ N"

Who is a member of the Vip Club?
Bypassing.
No match.
Adding a top level rule.
  "S -> NP:[wh=y] is NP?"
Adding 2 second level rules.
  "NP -> PRO" and
  "NP -> DET N P DET NAME"

Is John a citizen of the United States?
Bypassing.
EBL look-up succeeded.
```

The top-level rule used is "S -> is NP NP?", from the second example; the second-level rules are "NP -> Name" from the first example, and "NP -> DET N P DET NAME" from the third.

5. Conclusions and further directions

On the basis of the experiments reported here, we think there are good reasons to take EBL seriously as a practical and generally applicable way of optimizing NL query systems; the speed-ups achieved were very considerable at a low overhead. Even more importantly, it was possible to apply the EBL method despite the target's having several characteristics undesirable from this point of view; our *a priori* guess at the beginning of the project was that, if it worked here, it would work on most systems. We plan soon to begin implementation of a similar module for a large unification grammar for Swedish, where it should be easy to cover both syntactic and semantic processing.

One thing that ought to be studied more is the dependence of access time on the number of learned rules when this number becomes large (over 500, say). It certainly seems reasonable to hope that the pattern-matching algorithm presented here will give approximately logarithmic behaviour, but this is really an empirical question, since it depends on the distribution of the common query-types in terms of their lexical categories. Another important question is the extent to which it is possible to compress the generated rules. Since we are essentially trading space for time, this is likely to define the limits of the method, since we will eventually simply run out of space to store more learned rules, even if we can index them efficiently.

In conclusion, it seems to us that application of the EBL method to Natural Language offers a fruitful field for continued investigation of both a practical and theoretical nature.

Acknowledgements

This project would have been impossible without the assistance of many people at SICS and IBM Nordic Laboratories. We would in particular like to thank Ivan Bretan, Carl Brown, Jane Brown, Mats Carlsson, Pär Dahlin, Mikael Eriksson, Per Kristiansson, Sten Orsvärn and Mohammad Sanamrad for their help and support.

References

1. 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** pp. 11-62, 1989

2. Pereira, F.N.C. *Logic for Natural Language Analysis*, SRI Technical Note No 275, 1983

3. Rayner, M. "Applying Explanation-Based Generalization to Natural-Language Processing". *Proc. Intl. Conference on Fifth Generation Computer Systems*, Tokyo, 1988

4. Rayner, M. and Samuelsson, C., "Applying Explanation-Based Generalization to Natural-Language Processing (Part 2)". SICS Research Report 89015, 1989