BaseNP Chunking using ILP

Stasinos Th. Konstantopoulos

Humanities Computing, University of Groningen

Abstract

This paper reports on the application of Inductive Logic Programming (ILP) to the task of BaseNP chunking. After ILP and NP Chunking are discussed, the experimental setup for using ILP to construct a BaseNP tagger in Prolog is described. Finally, the results are analysed quantitatively as well as qualitatively.

1 Introduction

Inductive logic programming (ILP) is a machine learning discipline that lies at the intersection between inductive machine learning and logic programming. It is inductive machine learning because the objective of ILP algorithms is the extraction of knowledge from observations. It is logic programming in the sense that the formalism in which the generated knowledge is represented is that of a logic programme.

Text *chunking* is one of the many ways to retrieve parts of the syntactic structure of a sentence. It amounts to identifying non-overlapping constituents in a sentence without assigning internal structure to them. *BaseNP chunking* is one such chunking task, where the chunks are bottom level, non-recursive noun phrases.

This paper deals with the application of ILP to the task of BaseNP chunking, and it builds upon an earlier attempt (Konstantopoulos 2000) to apply ILP to BaseNP chunking. The paper starts with a brief introduction to ILP and Aleph in section 2. Aleph is the ILP system used for the experiments described here. Section 3 proceeds to describe the task and section 4 the experimental setup. Finally, section 5 presents the results of the experiment and draws conclusions.

2 Inductive Logic Programming

ILP attempts to generate knowledge, a *hypothesis*, within the bounds of a given theoretical framework and prior world knowledge, the *background knowledge*. That is accomplished by an inductive algorithm, which can be seen as the reverse of a deductive one. In other words, the purpose of an ILP algorithm is to construct a hypothesis H such that

$$B \wedge H \models D$$

where B is the background knowledge and D the data. For all practical purposes, and for the rest of this paper, the ILP task is restricted to Prolog programmes; that is, logic programmes consisting of definite Horn clauses. Furthermore, the training data consists of true or false ground facts only (positive and negative *examples*, or simply positives and negatives).

2.1 Background Knowledge

The background knowledge is the prior domain knowledge available about the phenomenon we are trying to describe. It includes the *background predicates* as well as *semantic* and *syntactic bias*.

The background predicates are Prolog predicates that provide the set of already existing concepts upon which the constructed hypothesis will be based. In other words, the background predicates define the literals that will be used in the body of the hypothesised clauses.

Bias, on the other hand, imposes syntactic and semantic restrictions on the clauses that may be considered by the algorithm. *Syntactic bias* operates on the form of the clauses constructed and is used to enforce restrictions like, for example, 'these two literals cannot appear in the same clause'. *Semantic bias* deals with the semantic interpretation of variables, that is their type and whether they are used as input or output variables. See also section 4.6 for more on how bias is declared and used.

2.2 The Evaluation Function

The *evaluation function* is used to measure a constructed clause's 'usefulness', based on an estimation of how well it would perform when queried on data unseen during the training. The evaluation function should balance between overfitting and overgeneralising, by favouring neither clauses that achieve high precision at the expense of coverage, nor vice versa.

If P is the number of positive examples covered by a clause and N the number of negative ones, typical evaluation functions include simple *coverage* $P - N$, or coverage with preference bias towards shorter clauses ($P - N - L + 1$, where L is the number of literals in the clause). More complex approaches to evaluation include computing the *Bayesian probability* that a hypothesis is correct given the data (Muggleton and Raedt 1994, 651) or the *Laplace expected accuracy*. Laplace accuracy is a special case of the m-probability estimate (Cestnik 1990) for balanced positive and negative data and no preference towards more general or more specific clauses. This yields the Laplace accuracy formula, $(P + 1)/(P + N + 2)$

2.3 The Search

ILP algorithms will typically be employing a *sequential cover* strategy that iterates through steps of constructing a clause and appending it to the current hypothesis. Each such step increases the hypothesis' coverage, and the process continues until some termination criterion—typically depending on the performance of the current solution—is satisfied. Progol (Muggleton 1995) is such an ILP algorithm, and is the one that will be described here as a typical example. Aleph (Srinivasan 2002) is the ILP system used for the experiment described below, and is implementing the Progol algorithm.

The search for each clause is organised as a generate-and-test, general-to-specific search between the *top clause* and the *bottom clause*. The top clause is the maximally general, empty-bodied clause that accepts all instances as positive. The bottom clause is a maximally specific clause that accepts exactly one example. The bottom clause is constructed by picking a positive example and applying to it a generalisation operator—in Progol's case the *inverse resolution operator* (Muggleton and Buntine 1988, Mitchell 1997). The bottom clause is guaranteed to be semantically correct (that is, to comply with the semantic bias provided) due to the way it is constructed, which means that all the clauses visited during the search are also going to be semantically correct.

The general-to-specific search is using the θ-subsumption operator (Plotkin 1969, 1971) to structure the search space on the general-to-specific axis. This reduces the problem to a search through the powerset of the body literals of bottom clause. The search strategy can be any of the usual search strategies (depth, breadth, or best-first, random walks, and so on). The evaluation function is used as heuristics for strategies like best-first that rely on heuristics.

The Progol ILP algorithm can, then, be outlined as follows:

1. Pick a positive example and derive a bottom clause from it.
2. Search between the top and bottom clause by:
   (a) Applying the traversal operator to the clause under consideration.
   (b) Checking if the resulting clause satisfies the syntactic constrains and if yes,
   (c) Evaluating it using the heuristic.

   and iterating until the clause under consideration is evaluated to be useful enough.
3. Append the newly constructed clause to the theory and remove from the positive examples pool the ones covered by the updated theory.
4. if the termination criterion is not satisfied, re-iterate.

When learning from consistent data the *termination criterion* is that all positives are covered. When handling noisy data, though, the objective is a 'high enough' number of positives covered without covering 'too many' negatives. The exact parameter values vary between applications and domains and reflect a prior estimation of the noisiness of the data.

2.4 Why ILP?

The use of background knowledge and bias constitutes one of the strongest advantages of ILP. This is of relevance to the linguistic/theoretical as well as the computational aspect of machine learning of natural language:

- Syntactic bias in general allows for the restriction of the hypothesis space within the limits of a meta-theory or theoretical framework. From the point of view of *theoretical linguistics*, this makes it possible to confine the search to one particular formalism. This is not to argue for any particular choice, only that some choice needs to be made.

- From the perspective of *natural language engineering*, ILP offers an opportunity to capitalise on linguistic knowledge in order to reduce the computational cost of searching the hypothesis space. Many alternative learning schemes, by contrast, cannot make any use of existing knowledge.

This is, of course, not to argue that control over the feature set, constraint satisfaction and bias cannot be implemented in statistical or distributed-computation approaches to machine learning. But the qualitative difference that ILP makes, is the ability to express those in an explicit and symbolic formalism (Prolog clauses) that is considered—for reasons independent from its being employed by ILP— to be particularly suitable for knowledge representation.

3 Chunking

Text *chunking* is a form of shallow parsing that amounts to identifying non-recursive, non-overlapping constituents *chunks* in a sentence, without assigning internal structure to the chunks. As an example, consider the following snippet taken from the Penn TreeBank (Marcus et al. 1993):

> [Confidence] in [the pound] [is widely expected] [to take] [another sharp dive] ...if [trade figures] for [September], [due] [for release] [tomorrow]...

where the bottom phrases of the full parse are shown in brackets.

Chunking is much faster than full parsing and can be very useful for tasks where the complete structure is not necessary or where the information gain from accessing the complete structure is not enough to justify the cost of full parsing. Such tasks would include information retrieval, text classification or optical character recognition. In text classification, for example, it is usually enough to extract bottom-level noun phrases (here 'confidence' and 'the pound') and use them as keywords to classify the text, without paying attention to the relations among them.

Abney (1991) introduces the concept of chunks in the field of NLP and parsing, motivated from psychological as well as a phonological (prosodic) evidence presented in his paper. He defines chunks as non-recursive, non-overlapping constituents with exactly one *major head*, where a major head is a content word (as opposed to a function word) that is not between a function word and the content word it selects. To return to our pound-confidence example, *confidence* and *pound* are content words and they are both major heads, so that 'confidence in the pound' cannot be a chunk since it has two major heads. The function word *the* selects for *pound* making 'the pound' one chunk. Even if there were content-word modifiers to *pound* (as, for example, in 'confidence in the U.K. pound') they would not be major heads and would be inside the same chunk as *pound*.

In analogy to phrase chunks in general, *Base Noun Phrases* (BaseNP) are defined here to be bottom level, non-recursive Noun Phrases including all the NP elements up to and including the head noun. By this definition, relative clause and prepositional phrase post-modifiers are excluded and recursion is avoided. For example the Penn TreeBank snippet given above would include the following BaseNPs:

> [Confidence] in [the pound] is widely expected to take [another sharp dive] if [trade figures] for [September], due for [release] [tomorrow]...

BaseNP chunking is a particularly interesting chunking task, due to the importance of Noun Phrases in typical shallow parsing tasks, such as information extraction and text classification.

3.1 Chunking as Tagging

One of the most important characteristics of the approach used to perform chunking, is the way in which it represents chunks. The most straight-forward way is bracketing, where the result is bracketed text with the restriction that brackets cannot be embedded. In Abney's paper (Abney 1991) the chunker is described as a context-free grammar used to recognise individual chunks and identify their head. These chunks would then be composed into a sentence by an *attacher*.

Abney suggests using a context-free chunk recogniser, since it is a very natural way to assign structure (that is, identify brackets and heads) to a phrase. It might, however, not be necessary to use as heavy computational machinery as a CFG to assign one-level structure, as is the case with chunking. One way to reduce the computational complexity of the task is to use a *finite state transducer* that will assign a *syntactic tag* to each word.

3.1.1 Bracket Tagging

Brackets can be seen as *tags* that mark the words at beginning and the end of each chunk:

> Confidence/B in/E the/B pound is/E widely expected to take another/B

so that words can be marked as opening a chunk (that is, being immediately after the opening bracket) or closing it (being immediately after the closing bracket). Adjacent chunks have to be appropriately treated, by either allowing double tagging or introducing some special 'E+B' tag:

> ... due for release/B tomorrow/E+B ...

One such system is described by Muñoz et al. (1999, Section 3.3), where two independent predictors (in fact, networks of linear predictors) are trained to assign opening and closing 'tag candidates' with their associated confidence levels. A second pass over the tagged sentence is finding the consistent bracketing with the highest overall confidence level. A similar approach is described by Tjong Kim Sang and Veenstra (1999) where the learner employed is TiMBL (a memory-based learner), except that its less sophisticated bracket matcher simply discards all inconsistent brackets, aiming for precision against recall.

3.1.2 Inside/Outside Tagging

The alternative approach is to make use of the fact that each word can only belong to one chunk (since they neither overlap nor are embedded within each other) and tag each and every word instead of only the ones at the edges of the chunks. Words are, then, tagged with Inside/Outside tags:

> Confidence/I in/O the/I pound/I is/O widely/O expected/O to/O take/O another/I

and only using a special B tag to separate adjacent chunks:

> ... due for release/I tomorrow/B ...

This last tagging schema was introduced by Ramshaw and Marcus (1995) in order to apply Transformation-Based Error-Driven Learning—a machine learning technique originally used by Brill (1995) to construct part-of-speech taggers—to the problem of chunking.

3.2 Comparison

Tjong Kim Sang and Veenstra (1999) and Muñoz et al. (1999) have compared the bracket tagging and the inside/outside tagging schemes by using the same machine learning technique to learn taggers for each of the two tagging schemes and compare the results.

Qualitatively, the main difference between the two approaches to syntactic tagging described above have to do with the consistency of the resulting tagging. Bracketing is more prone to result in inconsistent assignments (i.e. unbalanced brackets) and thus requires more sophisticated post-processing to pick the brackets from among the 'bracket candidates' proposed by the tagger. Inside/Outside tagging, on the other hand, is more robust, since all possible taggings are valid.

The quantitative results are expressed in terms of *precision*, *recall*, and *F_β-score*, which are the metrics typically used in information retrieval and machine learning. *Precision* is the ratio of true positive predictions over all positive predictions. High precision means that the model is not too liberal with accepting an example as positive. *Recall*, on the other hand, is the percentage of the positives that are predicted as such. High recall means that the model is not too conservative about accepting examples. The *F_β-score* is defined as

$$F_\beta(P,R) = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

where P is precision, R is recall, and β is the parameter balancing the importance of the two.

The three metrics above, are then used to quantitatively compare the two tagging approaches. Both comparisons seem to suggest that the two approaches are equivalent. Muñoz et al. (1999) report a recall of 92.5% at 92.2% precision ($F_{\beta=1} = 92.4$) for Inside/Outside tagging and 93.1% recall, 92.4% precision ($F_{\beta=1} = 92.8$) for bracket tagging. In the case of the memory-based learner (Tjong Kim Sang and Veenstra 1999, Table 6), bracket tagging yielded 90.8% recall at 93.7% precision ($F_{\beta=1} = 92.2$) and I/O tagging 92.3% recall at 92.5% precision ($F_{\beta=1} = 92.4$)

One important factor is, of course, the correlation between invalid taggings and wrong taggings. In other words, in the case of bracket taggers for example, a very strict bracket-matching scheme is going to improve precision, since the chunker is taking fewer 'risks' when making a positive decision. The deciding factor on whether this is going to improve performance is the price paid in terms of recall, since the more security the chunker requires before answering positively, the fewer positive answers there will be.

This drop in recall will be lower if invalid bracketings are more likely to also be wrong bracketings, in which case strict bracket matching has a positive side-effect. Tjong Kim Sang and Veenstra (1999, Table 6) compare three bracketing schemes where one is stricter than the other two: the first assigns a bracketing only in the presence of an opening and a closing bracket of matching phrase type, whereas the other two allow mismatches. The quantitative results show that the precision gain does indeed balance the recall drop, to the effect that the $F_{\beta=1}$ score is in all three cases around 92%.

4 Inducing a BaseNP Chunker

The task of automatically constructing a chunker has been reformulated in the previous section as one of constructing a syntactic tagger, marking each word with a tag denoting the kind of chunk it is in or marking it as not being in any chunk. The basic motivation for this is that this way the task is formulated like a transduction task rather than a parsing task, reducing its complexity from that of a context-free language to that of a regular language. In other words, the most complex part of the space of all possible tagging schemes has been excluded from our set of potential tagging schemes and in exchange it is possible to implement the tagger with a Finite State Machine (FSM) instead of more complex computational machinery. This has a significant advantage since simpler machines are easier to learn as well as more efficient to apply.

It does, then, beg the question why would one apply a Unification Grammar-learning method like ILP to a task that can be tackled with much simpler FSM-learning approaches. It can argued however, that it will be interesting to experiment with inducing a chunker, for the sake of the formalism itself rather than the formalism's descriptive power. In other words, it might be interesting to try the more complex mechanism because of the brevity, intuitiveness, or readability of the rules and despite the fact that the language described by these rules is not complex enough to necessitate their usage. In many respects this is analogous to the advantages of using a very powerful grammar formalism like HPSG to describe natural language syntax, although there is barely an argument for its being even context free.

The chunker will nevertheless be formulated as a syntactic tagger, as described above, and not as a CFG. In other words the descriptive power that Horn clauses offer will be focused not on the structure assigned on the phrase being parsed, which will be minimal, but on the justification (that is, formal logic proof) provided for each chunking decision made.

Learning such a tagger can be fitted in the context of a single-predicate learning ILP system that does not perform predicate invention or background knowledge refinement: the target predicate is the relation between a word and its syntactic tag, and the examples can be easily extracted from a parsed or chunked corpus. This section describes employing such an ILP system (Aleph, see section 2.3 above) to learn a BaseNP chunker from a corpus of chunked text.

4.1 Experimental Setup

Based on the analysis in section 3.2 above, the Inside/Outside tagging scheme has been chosen for this experiment, since (a) there appears to be no performance advantage in choosing bracket tagging, and (b) Inside/Outside tagging has the advantage that it requires no post-processing. This is especially the case with a machine learner like Aleph, where there is no probability assigned to each tagging decision made, making it more difficult to apply any informed bracket balancing scheme.

The target concept (the syntactic tagger) is represented as a `tagger/4` predicate that relates a word and its context to a syntactic tag:

```
tagger/4 (+LeftContext, +Word, +RightContext,
          ?SyntacticTag)
```

where the input arguments are the word to be syntactically tagged and its context. The tagger is meant to be used in a left-to-right pass over a sentence that has already been part-of-speech-tagged, so that the left context carries chunk tags as well as part-of-speech tags, where the right context holds part-of-speech tags only.

The contexts are Prolog lists of word terms, and each such term encapsulates the word-form, the part-of-speech tag, and the syntactic tag. The word to be tagged is also a word term. Examples of of these two kinds of word terms are:

```
w(confidence,nn,inp).
w(widely,rb).
```

where the two part-of-speech tags stand for ‘common noun, singular’ and ‘adverb’. The tag-set used here is the one in the Penn TreeBank—from which the dataset was extracted—described at length by (Santorini 1990). The syntactic tag is simply one of `bnp`, `inp` or `o`, as explained above.

4.2 The Dataset

Given the above, one example can be constructed from each word in the dataset:

```
tagger( [word(confidence,nn,bnp)],
        word(in,in),
        [word(the,dt), word(pound,nn), ... ],
        o)
```

Negative data is constructed by simply flipping the syntactic tag in a positive example. For this purpose, bnp and inp tags are taken to be in the same 'class' of tags, i.e. the class of tags that mark words inside a BaseNP. Substituting one for the other might generate too many false negatives, which is avoided by always choosing a tag from a different class to generate a negative example with. When flipping an o tag the choice between bnp and inp is made so that the result is a valid tagging, i.e. no inp tag is put immediately after an o tag. This way the positive example above would yield the following negative one:

```
tagger([word(confidence,nn,bnp)],
        word(in,in),
        [word(the,dt), word(pound,nn), ... ],
        bnp)
```

The implications of this way of generating negative data is that there are no examples of inconsistent (with respect to the the tagging scheme) tagging in the negative data, but only examples of wrong tagging. In other words, the clauses constructed by the ILP system do not need to ensure that there will be no inp tag immediately following an o tag, since that can be easily checked and fixed on the tagged text.

The data is taken from the same corpus as for the chunking experiments of Ramshaw and Marcus (1995), a derivative itself of the Penn TreeBank (Marcus et al. 1993).

4.3 List-Access Background Predicates

The two arguments of the `tagger/4` predicate that specify the context are lists of word terms, so the background predicates must include methods for accessing and manipulating lists. There are two ways to look at a Prolog list; either as a random-access array the members of which can be accessed by their offset from the beginning of the array or as a linked list where each element is pointing to the next element in the list. The latter approach was used here, based on the `head/2` and `rest/2` relations provided by Prolog.

Building upon the `head/2` predicate, the linked-list access methods consist of the `head_pos/2`, `head_synt/2` and `head_wform/2` predicates, that retrieve the part-of-speech tag, syntactic tag or word form of the head element, respectively. The `rest/2` predicate is made available as is. It should be noted that the left context list is reversed, so that its head is the word closest to the focus word. So, for example, the data extracted from the third word of the 'confidence in the pound' sentence would be:

```
tagger([word(in,in,inp), word(confidence,nn,bnp)],
       w(the,dt),
       [word(pound,nn),word(is,vbz), ... ],
       bnp).
```

This approach (when compared viewing the list as a random-access array) constitutes implicit preference bias towards shorter dependencies, since the search is in the general-to-specific direction. This means that shorter (more general) clauses will the visited first, and the shortest clause that is good enough (according to the evaluation function) will be chosen. It also imposes a limit on the longest dependency, since Aleph allows to set a limit on the layers of new variables. The following clause, for example:

```
tagger(A, word(nn,_), _, inp) :-
   rest(A,B), rest(B,C), C=word(dt,_,b).
```

is referring to a determiner two positions to the left, and introduces a 'chain' of input-output variables to do so. This chain is built with two layers of new variables (B and C). In the experiment described here, a maximum of 7 layers of new variables is set, which imposes a prior limit on the word distance within the text to which a rule can refer. There is no prior motivation for picking any particular value for this limit, so it should be seen as a working assumption the validity of which needs to be confirmed at the end of the experiment by examining the longest `rest`-`head` chains that appear in the rules. It should be stressed however that allowing for longer chains has a dramatic effect on the size of the bottom clause and, subsequently, that of the search space.

4.4 List-Manipulation Background Predicates

Besides access to the elements of the context lists, the background includes the list manipulation predicates provided in the Prolog library, like `reverse/2` and `member/2`. Furthermore, the `current_phrase/2 (+Context, -Phrase)` predicate is provided, matching a context list with the BaseNP up to this point (or an empty list if the last context word is marked as being outside a BaseNP. This predicate is only applicable to left context lists, since it is using the syntactic tags already assigned to extract the BaseNP.

The resulting sub-lists can then be accessed in the same manner as the full lists themselves. Potentially interesting pieces of information that can be extracted in this way are, for example, whether the BaseNP under consideration is definite or not:

```
is_definite_NP(Context) :-
  current_phrase(Context, A), re-
verse(A, B), head(B, the).
```

Table 1: The Baseline PoS to Syntactic Tag Map

Part of Speech	Syntactic Tag	Part of Speech	Syntactic Tag
determiners	bnp	nouns	inp
wh-determiners	bnp	adjectives	inp
existential 'there'	bnp	comparative adj.	bnp
pre-determiners	bnp	superlative adj.	inp
apostrophe-s	bnp	cardinals	inp
pronouns	bnp	foreign words	inp
possessive pronouns	bnp	symbols ($, &, etc)	bnp
wh-pronoun	bnp		
possessive wh-pronoun	bnp		

or whether there is already an adjective in the BaseNP or not:

```
has_adjective(Context) :-
  current_phrase(Context, A), member(w(jj,_,_), A).
```

4.5 The Baseline Theory

A 'naïve' tagger can be easily derived from the training set, by simply matching each part-of-speech tag to its most likely syntactic tag. For this particular experiment, the part-of-speech tags that were matched against `bnp` or `inp` tags are listed in table 1, with the remaining tags being marked as being outside a BaseNP.[1]

When used as the baseline theory it scores remarkably well, especially with respect to accuracy, (see results below) which suggests that it is encoding interesting information that could be useful to the learner.

The baseline tagger is included in the background as the `naive/2` predicate, matching each part-of-speech tag against its most probable syntactic tag:

```
%% naive/2 (?PoSTag, ?SyntTag)
naive(nn, inp).
naive(dt, bnp).
```

and so on.

4.6 Semantic Prior Knowledge

The semantic bias is declared with *mode declarations* that specify the manner in which a predicate is meant to be used. The information each predicate's mode

[1] Including part-of-speech tags that might appear in the test data but were not encountered in the training data.

carries is its arguments' mode, type, and non-determinacy. *Mode* specifies a term as being an input variable, output variable or ground term. The *type* is a label each variable bears and much match the type of an argument before the variable is considered as a value for that argument. The *non-determinacy* of a predicate sets an upper bound on the number of times it can succeed; that is, the number of discreet variable substitutions that satisfy the predicate.

The `tagger/4` predicate is declared as:

```
:- mode(3, tagger(+wslist,w(+pos,+word),+wlist,-
stag)).
```

The first argument sets an upper bound on the number of successful calls of this particular calling form of the predicate, and is chosen to reflect the fact that the output variable can be bound to any of its three possible values for each instance of input variables. The second argument is specifying a form that the predicate calls may take. `+T` arguments are input variables of type T and `-T` output variables of type T.

A determinate predicate would be one that can succeed in one way only, for example:

```
:- mode(1, head_synt(+wslist,-stag)).
:- mode(1, head_pos(+wslist,-pos)).
:- mode(1, head_pos(+wlist, -pos)).
```

These examples also demonstrate how the variable types are being used to restrict the applicability of `head_synt/2` the left context only, since the the right context has not been syntactically tagged yet.

5 Results and Conclusions

The setup described above was used to train on a data set of 6338 positive and 6337 negative examples. The evaluation function used was the Laplace expected accuracy (section 2.2). The resulting tagger consisted of 160 clauses, 11 of which constitute a substantial generalisation and cover the vast majority of the positive examples. The remaining 149 are ground clauses that are simply verbatim re-iterating the outlying positive examples that could not be generalised in any useful way.

The constructed theory was then tested by syntactically tagging 2012 unseen sentences and calculating the BaseNP precision and recall rate of the syntactic tagger. The theory achieved a recall rate of 85.32% with 78.62% precision, improving the 75.38% recall with 75.01% precision of the 'naïve' theory taken as the baseline.

One thing to be noted about these results is that perfect part-of-speech tagging is assumed, which will generally not be the case. More moderate results are expected against input pre-processed through a part-of-speech tagger, rather than input extracted from the part-of-speech tagged corpus.

Regarding the more qualitative aspects of the resulting theory, some of the constructed rules are reasonable and intuitive, whereas others are not. One commonly recurring pattern is the conditional use of the `naive/2` predicate, so that the constructed theory is effectively specifying the contexts in which `naive/2` is correct and limiting its application to those cases.

Some of the most convoluted rules of this kind look like this one here:

```
tagger(A,w(B,C),D,E) :-
   head_pos(D,F), head_synt(A,G), rest(D,H), head_pos(H,F),
   rest(H,I), head_pos(I,J), naive(J,G), naive(B,E).
%% [laplace estimate] [0.95122]
```

which stipulates that the syntactic tag `E` should be what `naive/2` predicts, given that:

- `head_pos(D,F), rest(D,H), head_pos(H,F)`: the part-of-speech-tag of the two first words to the right is the same, no matter what it is.
- `head_synt(A,G), rest(H,I), head_pos(I,J), naive(J,G)`: the syntactic tag of the first word to the left is the same as the tag predicted by `naive/2` for the third word to the right, no matter what it is.

This rule (and others like it) are an example of a theory that is not representable in a formalism weaker than Horn clauses, due to its usage of variables. Although it is always possible to unroll such rules in series of ground rules (in the same way that in finite domains Unification Grammars can be re-written as longer CFGs), Horn clauses are more concise and readable. This last observation doesn't, however, mean that all the rules are necessarily intuitive or interesting, only that the formalism allows for potentially interesting rules.

One problem to be noted with the experiment conducted is the absence of syntactic bias. It is difficult to specify syntactic bias because of the way the data is represented: by breaking up the sentence bracketing task into that of tagging individual words, the theory constructed is, in a way, 'distributed'. In other words, it is not easy to identify clearly the role of each clause in identifying a BaseNP, and the bracketing is the effect of the interaction between clauses rather than the result of the application of the appropriate clause for each particular case.

From the above it is clear that rules enforcing a theoretical framework such as, for example, X-bar theory's 'each XP must include a head X' cannot be easily represented as syntactic bias in the current setup. In general, it has difficulties with rules that are not local, either horizontally (long-distance dependencies) or vertically (that is, ones that make reference to complex tree structures like X-bar theory does).

As it has already been argued in section 2.4, however, providing an intuitive formalism to declare syntactic bias is one of the strongest points of ILP and if it is not possible to take advantage of it, the motivation for choosing ILP for the task is undermined. Further experiments on the task of BaseNP chunking need to be focused on defining more complex and more linguistically informed background theories within this formulation of the problem, as well as devising other formulations that might be better fitted to the ILP framework.

A related issue is that the theory is not as human-readable as one might expect for a logic programme, making the task of qualitatively evaluating the result much more difficult that it would be if the theory was a DCG or some other less 'distributed' formalism.

References

Abney, S. (1991), Parsing by chunks, *in* R. Berwick, S. Abney and C. Tenny (eds), *Principle-Based Parsing*, Kluwer Academic Publishers, Dordrecht.

Brill, E. (1995), Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging, *Computational Linguistics* **21** (4), 543–66.

Cestnik, B. (1990), Estimating probabilities: A crucial task in machine learning, *European Conference on Artificial Intelligence*, pp. 147–149.

Konstantopoulos, S. T. (2000), NP chunking using ILP, *in* P. Monachesi (ed.), *Computational Linguistics in the Netherlands 1999*, Utrecht Institute of Linguistics OTS, Utrecht, pp. 109–116.

Marcus, M., Santorini, B. and Marcinkiewicz, M. A. (1993), Building a large annotated corpus of English: the Penn Treebank, *Computational Linguistics* **19** (2), 313–330.

Mitchell, T. (1997), *Machine Learning*, second edn, McGraw Hill, New York, chapter 10, Learning Sets of Rules.

Muggleton, S. (1995), Inverse entailment and Progol, *New Generation Computing* **13**, 245–286.

Muggleton, S. and Buntine, W. (1988), Machine invention of first-order predicates by inverting resolution, *Proceedings of the 5th International Conference on Machine Learning*, Morgan Kaufmann, San Francisco, pp. 339–352.

Muggleton, S. and Raedt, L. D. (1994), Inductive Logic Programming: Theory and methods, *Journal of Logic Programming* **19** (20), 629–679.

Muñoz, M., Punyakanok, V., Roth, D. and Zimak, D. (1999), A learning approach to shallow parsing, *Proceedings of EMNLP-WVLC '99*, College Park.

Plotkin, G. (1969), A note on inductive generalization, *in* B. Meltzer and D. Michie (eds), *Machine Intelligence*, Vol. 5, Edinburgh University Press, Edinburgh, pp. 153–163.

Plotkin, G. (1971), A further note on inductive generalization, *in* D. Michie, N. Collins and E. Dale (eds), *Machine Intelligence*, Vol. 6, Edinburgh University Press, Edinburgh, pp. 101–124.

Ramshaw, L. and Marcus, M. (1995), Text chunking using transformation-based learning, *Proceedings of the Third ACL Workshop on Very Large Corpora*, Cambridge, pp. 82–94.

Santorini, B. (1990), Part-of-speech tagging guidelines for the Penn Treebank project, *Technical report*, University of Pennsylvania, Philadelphia. 3rd Revision, 2nd Printing.

Srinivasan, A. (2002), Aleph, `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`. Last update: November 2002.

Tjong Kim Sang, E. and Veenstra, J. (1999), Representing text chunks, *Proceedings of the Ninth Conference of the European Chapter of the ACL*, Bergen, pp. 173–179.