Using ILP to learn local linguistic structures
Konstantopoulos, Stasinos Themistokleous

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2003

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 25-04-2020
Chapter 4

Shallow Parsing

This chapter deals with applying Inductive Logic Programming (ILP) to the task of chunking, a form of shallow parsing explained in Sections 4.1 and 4.2. Then an overview of previous machine-learning approaches to chunking is given (Section 4.3), while the remainder of the chapter describes the ILP chunking experiment.

4.1 Full vs. Shallow Parsing

Syntax is the study of grammar relations between words and other units within the sentence. (The Concise Oxford Dictionary of Linguistics, [41])

The syntax of an utterance is, thus, the way in which words combine to form grammatical phrases and sentences and the way in which the semantics of the individual words combine to give rise to the semantics of phrases and sentences. It is, in other words, the structure hidden behind the (flat) utterance heard and seen on the surface. At least since the early work of Chomsky [12], it is the fundamental assumption of linguistics that this structure has the form of a tree, where the terminal symbols are the actual word-forms and the non-terminal symbols abstractions of words or multi-word phrases. So, for example, the phrase ‘confidence in the pound’ would be assigned the structure shown in Figure 4.1 to mean that ‘the’ and ‘pound’ combine to make a noun phrase (N1) component, before combining with the preposition ‘in’ to form a preposition phrase (PP), and so on.

The rules according to which words and lower-level phrases of a given language may combine to form a higher-level phrase constitute the grammar of the language. Grammars are typically expressed as Context-Free Grammars (CFG) or in a formalism that extends CFG with feature unifications, e.g. Definite Clause Grammars (DCG), or an even stronger formalisms, such

73
as Head-driven Head Structure Grammar (HPDG).\footnote{Shieber \cite{76} gives an overview of various unification-based grammar formalisms.} In the ‘confidence in the pound’ example here, the CFG fragment that would assign the correct structure, would look like this:

\[
\begin{align*}
\text{NP} & \rightarrow \text{N1 PP} & \text{Prep} & \rightarrow \text{in} \\
\text{PP} & \rightarrow \text{Prep N1} & \text{Det} & \rightarrow \text{the} \\
\text{N1} & \rightarrow \text{Det N} & \text{N} & \rightarrow \text{confidence} \\
\text{N1} & \rightarrow \text{N} & \text{N} & \rightarrow \text{pound}
\end{align*}
\]

A DCG would also allow for features to percolate up the tree:

\[
\begin{align*}
\text{NP(Det)} & \rightarrow \text{N1(Det) PP} & \text{Det(def)} & \rightarrow \text{the} \\
\text{PP} & \rightarrow \text{Prep N1(\_)} & \text{Det(indef)} & \rightarrow \text{a} \\
\text{N1(Det)} & \rightarrow \text{Det(Det) N} & \text{N} & \rightarrow \text{confidence} \\
\text{N1(\text{none})} & \rightarrow \text{N} & \text{N} & \rightarrow \text{pound}
\end{align*}
\]

in this case marking a noun phrase as definite, indefinite, or lacking a determiner. A DCG is immediately expressible in First-Order Horn clauses (Prolog clauses) with the use of difference lists.

Parsing is the task of assigning syntactic structure to a sentence by a programme — the parser — implementing the grammar rules according to which words and phrases combine to form higher-order phrases. It should be noted at this point that linguistic theories differ in the constituents and rules which they postulate, and this difference must be reflected in the structures which parsers assign to strings of words. In other words, parsers must not
only recognise all the grammatical sentences of a language and only those, but they must also do so by building the correct structure.

*Shallow parsing* is the extraction of some features from some (speech or text) material without performing a full parse to discover its complete structure. The fragments of the structure that it does retrieve tend to be flat. As an example, consider a shallow parser that retrieves the first nominal projections (N1) of a sentence, without further analysing their contents. The full-parsing tree of Figure 4.1 would then be reduced to a tree with only two (non-terminal) levels:

(1)  [Confidence] in [the pound]

where the brackets denote a sub-tree with an N1 label as its non-terminal node.

Shallow parsing 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 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.

Shallow parsing can also be used as a first stage for full parsing where, for example, a part-of-speech tagger might be used to assist a parser to reduce the number of parses to be considered. The information gain in this case comes for employing different paradigms for the tagger and the parser (typically a stochastic tagger and a DCG parser) so that the full parser can employ the information or relations that the stochastic model is better fitted to discover.

Stochastic models perform very well in shallow parsing tasks, and are very fast to train and apply. Furthermore — and since a shallow parser is required to assign only minimal structure to the ‘parsed’ material — their non-symbolic and non (or hardly) human-readable representation is less of an issue than it would have been for a full parser. In other words, in full parsing the process itself (that is, the series of rules applied) through which a sentence is recognised is important and that gives a big advantage to symbolic rule systems. In shallow parsing, on the other hand, the application of the rules is not part of the answer, which makes stochastic models more attractive.

Despite that, a symbolic, easier to manipulate representation would always be useful if it were as accurate and efficient as a stochastic one. To the array of generic (that is, not task-specific) arguments in favour of ILP
(ease of incorporating prior knowledge and enforcing a theoretical framework, human-readable resulting theories), one might add the ease of incorporating the resulting shallow parser into a bigger parsing system, since parsers’ grammars are typically expressed in Horn clauses.

4.2 Chunking

Text chunking is a form of shallow parsing that amounts to identifying non-recursive, non-overlapping constituent chunks in a sentence. As an example, consider the following snippet taken from the Penn TreeBank [40]:

(2)  [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.

4.2.1 What is a chunk?

There is no general definition of a chunk, but the general guidelines given above are used to define chunks for each language (or even task). Abney [1] introduces\textsuperscript{2} 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’.

This definition is based on English and would not be applicable in languages that would, for example, have adjectives follow the nouns they modify and not necessarily precede them as is the case in English. So, for example, in Spanish the noun phrase:

\textsuperscript{2}Muñoz et al. [56] quote an article by Harris [28] as the one introducing the notion of chunking as early as 1957, but that is not accurate.
4.2. Chunking

(3) los derechos humanos
   the rights human-PL

is — intuitively — a chunk, although it includes a content word (the adjective
‘humanos’) that is not between a function word and the content word it
selects (‘los’ and ‘derechos’, respectively).

The intuition that they are the bottom phrases of the full parse is also
not an adequate definition of a chunk for various reasons:

• On the practical level — and since chunking is to be used as a prepro-
cessing stage before full parsing — it has to be performed on plain (or at
most part-of-speech-tagged) text, so it cannot be defined in terms of the
full parse. And, even if it were to be defined simply as the bottom-level
phrases, this would still have to be ‘translated’ into a language-specific
definition applicable to flat text with the grammar of the language in
mind.

• Chunks are sometimes defined in surprising ways due to practical con-
considerations. Ramshaw and Marcus [69], for example, bundle the apo-
strophe s in phrases like ‘Ramshaw’s paper’ in the second noun’s chunk,
so that this fragment would be chunked as:

(4) [ Ramshaw ] [ ’s paper ]

While the authors provide no further motivation for this choice other
that it flattens the recursive structure ‘in a useful way’ (Section 2.2),
one can assume that preliminary experiments have shown this approach
to be advantageous.

• Some syntactic phenomena might lead to surprising or counter-intuitive
results if the bottom-level rule is adhered to religiously. In German,
for example, one might get NPs like:

(5) [NP Die [PP in Pfund] vertauende Leute ]
    [NP The [PP in the Pound] trusting people ]

where one might argue that the most interesting or useful way to chunk
the NP headed by ‘Leute’ is to have it include its determiner, but the
base-phrase-only definition would imply that ‘Leute’ is a chunk by its
own.

It is, therefore, the case that individual chunking experiments (sometimes
even on the same language) are based on different assumptions on what an
interesting chunk is and there is no general and clear definition of a chunk.
4.2.2 Noun Phrase Chunks

In analogy to phrase chunks in general, Base Noun Phrases (BaseNP) are 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:

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

BaseNPs 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.

4.3 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 straightforward way is bracketing, where the result is bracketed text with the restriction that brackets cannot be embedded. In Abney’s paper [1] 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 computational machinery as heavy as a CFG to assign one-level structure, as is the case with chunking. The advantage of a CFG over a Finite State Machine (FSM)\(^3\) is that it is able to assign more complex, multiple-layered structure to the strings it recognises, as opposed to the single layer of output of a finite state transducer. It is immediately obvious that this advantage is of no interest to the task of chunking, since we are only interested in a single layer of structure anyway.

This suggests that a more computationally efficient way to approach chunking is by representing chunks not as a one-level tree structure over the phrase, but as a syntactic tag associated with each word. These tags can be either open/close bracket tags, or inside/ outside tags.

---

\(^3\)In the cases where CFGs are used to analyse regular languages. This whole discussion is obviously not pertinent to the analysis of context free — or more complex — languages that are not representable by FSMs in the first place.
4.3. Chunking as Tagging

4.3.1 Bracket Tagging

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

(7) 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:

(8) ... due for release/B tomorrow/E+B ...

One such system is described by Muñoz et al. [56, 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 [84] where the learner employed is TiMBL (a memory-based learner [30]), except that its less sophisticated bracket matcher simply discards all inconsistent brackets, aiming for precision against recall.

4.3.2 Inside/Outside Tagging

The alternative approach is to make use of the fact that each word can belong to at most 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:

(9) Confidence/B in/E the/B pound/I is/E widely/O expected/O to/O take/O another/B

where words are marked as being inside (I) or outside (O) a chunk as well as opening or closing one. Words tagged as B are implicitly also marked as being inside a chunk, and words tagged as E as being outside all chunks. In fact, one could retrieve the chunks by simply marking the words with I/O tags, since the edges will be obvious:

(10) Confidence/I in/O the/I pound/I is/O widely/O expected/O to/O take/O another/I
and only using B tags to separate adjacent chunks:

\[(11) \quad \ldots \text{due for release/}I \text{ tomorrow/}B \ldots\]

This last tagging schema was introduced by Ramshaw and Marcus [69] in order to apply Transformation-Based Error-Driven Learning — a machine learning technique originally used by Brill [8] to construct part-of-speech taggers — to the problem of chunking.

### 4.3.3 Comparison

Tjong Kim Sang and Veenstra [84] and Muñoz et al. [56] have compared the two approaches 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 is 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 \( F_\beta \)-score, which are the metrics typically used in information retrieval and machine learning. \emph{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. \emph{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 \( F_\beta \)-score is defined as

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

where \( P \) is precision, \( R \) is recall, and \( \beta \) 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. [56] report 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 [84, 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 \( (F_{\beta=1} = 92.4) \).
4.4 Inducing a BaseNP Chunker

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 [84, 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-score is in all three cases around 92%.

4.4 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 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 Horn clause 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 2.4 above) to learn a BaseNP chunker from the corpus used by Ramshaw and Marcus [69], a derivative itself of the Penn TreeBank [40]. The work on shallow parsing presented here was originally published in the Proceedings of the Computational Linguistics in the Netherlands 1999 conference [33], and follow-up work in the proceedings of the 2002 edition of the same conference [36].

### 4.4.1 Experimental Setup

Based on the results described in Section 4.3 above, the Inside/Outside tagging scheme has been chosen, 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:

\[
\text{tagger/4 (+LeftContext, +Word, +RightContext, ?SyntacticTag)}
\]

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 syntactic 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 — where applicable — the syntactic tag. The word to be tagged is also a word term. Examples of these two kinds of word terms are:

\[
\text{w(confidence,nm,inp)}.
\]

\[
\text{w(widely,rb)}.
\]
4.4. Inducing a BaseNP Chunker

where the two part-of-speech tags stand for ‘common noun, singular’ and ‘adverb’, respectively. The tag-set used here is the one in the Penn TreeBank — from which the dataset was extracted — described at length by Santorini [72]. The syntactic tag is simply one of bnp, inp or o, as explained above.

To further demonstrate how the syntactic tagger works, consider the sterling-confidence example used above. The part-of-speech tagger would tag the beginning of that sentence as:

(12)  Confidence/nn in/in the/dt pound/nn is/vbz widely/rb expected/vbn to/to take/vb another/dt ...

When the tagger is applied to the first word it would be activated as:

\[
tagger([], \text{word(confidence,nn)}, \\
    \quad \text{[word(in,in), word(the,dt),...]}, \text{Tag})
\]

and would unify Tag with bnp. For the second word,

\[
tagger([\text{word(confidence,nn,bnp)}], \text{word(in,in)}, \\
    \quad \text{[word(the,dt), ...]}, \text{Tag})
\]

The tagger would unify Tag with o, and so on until the the whole sentence has been syntactically tagged.

The relation that this predicate is defining is meant to be a function, so that no backtracking is performed and the tagging is performed as a single, tractable pass over the sentence. This cannot be guaranteed by the learning process, in the sense that multiple activations of the learnt predicate are possible. This can be tackled in various ways; here a first-match-wins policy is enforced by the programme that employs the tagger/4 predicate to produce tagged text.

4.4.2 The Dataset

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

\[
tagger([\text{word(confidence,nn,bnp)}], \\
    \quad \text{word(in,in)}, \\
    \quad \text{[word(the,dt), word(pound,nn), ...]}, \\
    \quad \text{o})
\]

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:

\[
\text{tagger}([\text{word}(\text{confidence, nn, bnp})], \\
\quad \text{word}(\text{in, in}), \\
\quad [\text{word}(\text{the, dt}), \text{word}(\text{pound, nn}), \ldots ], \\
\quad \text{bnp})
\]

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.

### 4.4.3 List-Access Background Predicates

A general, theoretic overview of background knowledge has been given in Section 2.3.3, so this section will concentrate more on the way in which it has been employed for this experiment.

The two arguments of the $\text{tagger}/4$ 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.

Random access is implemented through the $\text{nth}/3$ predicate\(^4\) which is matching a list and an offset with the item found at that offset in the list. The list access methods built upon this predicate are $\text{get_ptag}/3$, $\text{get_stag}/3$ and $\text{get_wform}/3$ which retrieve the part-of-speech tag, syntactic tag and word form at the given offset, respectively.

\(^4\)Traditionally, Prolog lists are linked lists with random access implemented by recursively traversing the list. The Yap compiler implements random access arrays as well, but for the sake of demonstration linked lists will be used for both cases. If, however, the syntactic tagger were to make many references to items deep in the list, the lists would have to be substituted with arrays for efficiency. The conversion is trivial, so this has no impact on the experiment as a whole.
Linked list access is viewing the list as a head element and a tail list. The tail is a linked list itself, also consisting of a head and a tail, and so on through the list until the last element that has an empty list as a tail. This way the first element of list \( L \) is referred to as head(\( L \)), the second as head(tail(\( L \))), the third as head(tail(tail(\( L \)))) and so on. Prolog provides the head/2 and rest/2 relations, so that the above would be formulated as:

\[
\begin{align*}
\text{head}(L, \text{FirstElement}). \\
\text{rest}(L, \text{Rest}), \text{head}(\text{Rest}, \text{SecondElement}). \\
\text{rest}(L, \text{Rest1}), \text{rest}(\text{Rest1}, \text{Rest2}), \text{head}(\text{Rest2}, \text{ThirdElement}).
\end{align*}
\]

Building upon the head/2 predicate, the linked-list access methods consist of the head_pos/2, head_syt/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 example extracted from the third word of the ‘confidence in the pound’ sentence would be:

\[
\text{tagger}([\text{word(in, in, inp)}, \text{word(confidence, nn, bnp)}], \\
\text{w(the, dt)}, \\
[\text{word(pound, mn)}, \text{word(is, vbz)}, \ldots ], \\
\text{bnp}).
\]

This approach (when compared to random-access predicates) 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 there is a limit on the length of ‘chains’ of new\(^5\) variables. The following clause, for example:

\[
\text{tagger}(A, \text{word(mn, _), _, inp}) :- \\
\text{rest}(A, B), \text{rest}(B, C), C=\text{word(dt, _, b)}.
\]

is referring to a determiner two positions to the left, and introduces a ‘chain’ of input-output variables to do so. 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

\(^5\)Variables not appearing on the head, but introduced in the body by the learner.
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.

### 4.4.4 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:

```prolog
is_definite_NP(Context) :-
    current_phrase(Context, A), reverse(A, B), head(B, the).
```

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

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

### 4.4.5 Linguistic Background

Some part-of-speech tags carry some morphological and syntactic information as well as the word's part-of-speech in the strict sense of a lexical class. Nouns, for example, are tagged with one of four possible tags: singular common noun (nn), plural common noun (nns), singular proper noun (nnp) or plural proper noun (nmps). This lends itself to a classification based on two plus-minus features, PLURAL and PROPER. This information can be very simply encoded in the following relations:

```prolog
tag_plur(nn, -). tag_prop(nn, -).
tag_plur(nns, +). tag_prop(nns, -).
tag_plur(nnp, -). tag_prop(nnp, +).
tag_plur(nmps, +). tag_prop(nmps, +).
```
4.4. Inducing a BaseNP Chunker

<table>
<thead>
<tr>
<th>Part of Speech</th>
<th>Syntactic Tag</th>
<th>Part of Speech</th>
<th>Syntactic Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>determiners</td>
<td>bnp</td>
<td>nouns</td>
<td>inp</td>
</tr>
<tr>
<td>wh-determiners</td>
<td>bnp</td>
<td>adjectives</td>
<td>inp</td>
</tr>
<tr>
<td>existential ‘there’</td>
<td>bnp</td>
<td>comparative adj.</td>
<td>bnp</td>
</tr>
<tr>
<td>pre-determiners</td>
<td>bnp</td>
<td>superlative adj.</td>
<td>inp</td>
</tr>
<tr>
<td>apostrophe-s</td>
<td>bnp</td>
<td>cardinals</td>
<td>inp</td>
</tr>
<tr>
<td>pronouns</td>
<td>bnp</td>
<td>foreign words</td>
<td>inp</td>
</tr>
<tr>
<td>possessive pronouns</td>
<td>bnp</td>
<td>symbols ($, &amp; etc)</td>
<td>bnp</td>
</tr>
<tr>
<td>wh-pronoun</td>
<td>bnp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>possessive wh-pronoun</td>
<td>bnp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The Baseline PoS to Syntactic Tag Map

making it possible to refer to all possible subsets of noun tag. Adjectival and
erval tags are also broken down into similar morphosyntactic features.

Determiners are marked with a single part-of-speech tag and no distinction
is made between singular and plural, since no such distinction is made
in the original corpus either.

Furthermore it might be useful to be able to refer to any noun tag, or
any nominal (noun or adjective) tag, etc:

\[
\text{tag\_nominal(nn, +). tag\_verbal(nn, -). \% nouns (sg)}
\]
\[
\text{tag\_nominal(nns, +). tag\_verbal(nns, -). \% nouns (pl)}
\]
\[
\text{tag\_nominal(jj, +). tag\_verbal(jj, +). \% adjectives}
\]
\[
\text{tag\_nominal(prp, -). tag\_verbal(prp, -). \% prepositions}
\]

and so on, covering all the tags that are pertinent to any of the four major
lexical categories (nouns, verbs, adjectives and prepositions).

4.4.6 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 4.1, with the remaining tags being marked as being
outside a BaseNP, including part-of-speech tags that might appear in the
test data but were not encountered in the training data.

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:

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

### 4.4.7 Prior Bias

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 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.

The tagger/4 predicate is declared as:

```plaintext
:- mode(1, tagger(+wlist, w(+pos, +word), +wlist, -stag)).
```

The first argument is an upper bound of the *non-determinacy* (i.e. number of successful calls) of this particular calling form of the predicate. 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:

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

### 4.5 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 Laplace estimated accuracy (see Section 2.2.3).
4.5. Results and Conclusions

The resulting tagger consisted of 160 clauses, 11 of which constitute a substantial generalization 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 generalized 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 ‘naive’ theory taken as the baseline. (See Section 6.2 for comparison with similar work.)

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:

\[
\text{tagger}(A,w(B,C),D,E) : - \\
\text{head\_pos}(D,F), \text{head\_synt}(A,G), \text{rest}(D,H), \text{head\_pos}(H,F), \\
\text{rest}(H,I), \text{head\_pos}(I,J), \text{naive}(J,G), \text{naive}(B,E).
\]

\%
\%

\[
\text{[laplace estimate]} \ [0.95122]
\]

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

- \text{head\_pos}(D,F), \text{rest}(D,H), \text{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.

- \text{head\_synt}(A,G), \text{rest}(H,I), \text{head\_pos}(I,J), \text{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 into series of ground rules (in the same way that in finite domains Definite Clause 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, this setup 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 2.3.3 however, one of the strongest points of ILP is that it provides an intuitive formalism for declaring syntactic bias; so 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 Definite Clause Grammar (DCG) or some other more intuitively appealing formalism. Work in this direction would involve developing tools to extract the information in a theory thus constructed and reformulated in a more human-readable form.

4.5.1 Cascades of Chunkers

The other important limitation encountered stems not from ILP, but from syntactic tagging itself. Syntactic tagging as it stands, can only partition a sentence into non-overlapping chunks, thus making recursive theories unrepresentable. This renders the technique inapplicable to the task of top-level NP identification, since top-level NPs will inevitably contain smaller ones in preposition phrases or relative clauses.

One way in which recursion could be mimicked would be to break up the task in a bottom-up fashion. A separate tagger would then be induced
for each layer and parsing would proceed by repeated steps of recognising constituents and replacing them with a head node symbol. For example consider our original example,

(13) [[Confidence] in [the pound]] is widely expected to take [another sharp dive] if [[trade figures] for [September]]...

where higher-level NP bracketing is also marked. This could be syntactically tagged in two stages, the first one of which would be BaseNP chunking. Then all BaseNPs found are replaced by a label and in the new tagging problem these labels are treated as nouns:

(14) [BaseNP/nn in BaseNP/nn] is widely expected to take BaseNP/nn if [BaseNP/nn for BaseNP/nn]...

This approach, however, suffers from data sparseness at the higher levels, as well as from the fact that erroneous decisions made at the lower levels cannot be revisited as would be the case with a backtracking parser. In other words, by the time the tagging layers have reached the level of top-level NPs, the results will not be reliable enough to be useful for information extraction.

A different way of using such a cascade of chunkers is as a heuristic for a full parser [7], in a setup where the chunker’s suggestions are corroborated by the full parser and lower-level chunkings can be backtracked out of, if proven wrong at higher level. Such a setup, however, is restricting the shallow parser’s domain of applicability to that of a pre-processor (or, rather, co-processor) for the full parser, leaving out tasks where a full parse is not necessary.