NP Chunking using ILP

Stasinos Konstantopoulos
Alfa-Informatica, Rijks*universiteit* Groningen
konstant@let.rug.nl

Abstract

This is to report the results of approaching the problem of NP chunking using Inductive Logic Programming techniques. The problem, as defined in (Ramshaw and Marcus, 1995), is the machine learning of rules that recognise non-recursive, base NPs in text annotated with part-of-speech tags, by tagging each word as being 'inside' or 'outside' an NP. (Consecutive NPs are appropriately treated.)

The same input data as in the original experiment is used here, but the machine learning technique is Inductive Logic Programming, and more specifically the Progol algorithm. The problem is formulated as the machine learning of a Prolog predicate that will accept a part-of-speech tagged word and its context as input and associate it with the appropriate syntactic tag.

1 Introduction

Text *chunking* amounts to identifying non-overlapping constituents in a sentence, without assigning internal structure to them. It is useful either as a preprocessing stage for full parsing or in the context of shallow parsing for information retrieval.

A *BaseNP* is a bottom level, non-recursive Noun Phrase (NP) including all the NP elements up to and including the head noun. This way relative clause and prepositional phrase post-modifiers are excluded and recursion is avoided. For example, consider the following snippet taken from the Penn TreeBank, with the BaseNPs shown in brackets:

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

Syntactic Tags can be used to mark a non-overlapping partitioning of a sentence, such as chunking. Since any given word can only be in at most one chunk, it is enough to mark each word with a syntactic tag, denoting the kind of chunk it is in or marking it as not being in any chunk. For the task of BaseNP chunking, it is enough to assign each word the tag `i` if inside and `o` if outside a BaseNP. A third tag `b` is used to tag the first word of a Base-NP; this is necessary in order to distinguish between two adjacent Base-NPs and a single longer one. According to this the example above would be tagged as follows:

> Confidence/b in/o the/b pound/i is/o widely/o expected/o to/o take/o another/b sharp/i dive/i if/o trade/b figures/i for/o September/b, due/o for/o release/b tomorrow/b...

The advantage of reformulating the problem in this manner is that techniques developed for part-of-speech tagging can be readily transferred and applied to chunking. Such an experiment is described in (Ramshaw and Marcus, 1995). The learning methodology used there is Transformation-Based Error-Driven Learning, originally used to automatically construct part-of-speech taggers. (Brill, 1995)

In this paper *Inductive Logic Programming* (ILP) is used as the learning method instead. The general setup of the experiment remains the same, i.e. one where chunking is viewed as a tagging rather than a bracketing problem. This is not only to compare ILP with Transformation-Based Error-Driven Learning, but also the performance of ILP on a tagging task as opposed to directly bracketing (by, for example, inducing a DCG).

2 Inductive Logic Programming

Inductive Logic Programming (ILP) is a Machine Learning discipline. It is logic programming in the sense that the target concept to be learned is a logic programme, i.e. a set of Horn clauses. It is inductive because the core operator used is that of *induction*.

Induction can be seen as the reverse of deduction. For example from the clauses

1. All humans die.
2. Sigurd is human.

the fact that Sigurd will eventually die can be deduced. Inversely, induction uses *background knowledge* (e.g. *Sigurd is human*) and a set of observations or *training data* (e.g. *Sigurd died*) to search for a *hypothesis* that, in conjunction with the background knowledge, can deduce the data. In this case all of *Sigurd dies*, *All humans die* or *Everybody dies* are hypotheses that satisfy this requirement, but as is intuitively obvious, they are not all equally useful or interesting.

As formally defined in (Muggleton and De Raedt, 1994, p. 635), induction constructs a hypothesis H with the following properties:

$$\begin{array}{ll} \textbf{Prior Satisfiability} & B \wedge D^- \not\models \square \\ \textbf{Posterior Satisfiab.} & B \wedge H \wedge D^- \not\models \square \\ \textbf{Prior Necessity} & B \not\models D^+ \\ \textbf{Post. Sufficiency} & B \wedge H \models D^+ \end{array} \quad (1)$$

where B and $D = D^+ \cup D^-$ are logic programmes that represent the background knowledge and the training data. D, in particular, is partitioned in the subset D^+ of clauses that succeed when queried with a positive examples and D^- that fail when queried with negative examples.[1]

The background knowledge and the data can been seen as Prolog clauses. It consists of a database of ground facts as well as clauses representing prior knowledge. So, for example, one might have:

```
human(sigurd). human(gunnar).
aesir(thor).   aesir(odin).
god(X) :- aesir(X).
```

[1] Since D is meant to be queried only about the training examples that are available, failure denotes an explicit negative example and not the inability to prove an example positive.

representing the background knowledge. The data, on the other hand, is usually restricted to ground facts in what is called the *example setting*. The terms 'examples' and 'data' will be used interchangeably to denote ground facts representing the evidence available:

```
dies(sigurd).
false :- dies(thor).
```

Since the two prior requirements of (1) hold in this example, the objective is to identify the *target predicate* that satisfies the posterior ones. Such a predicate is not necessarily unique, as can be easily demonstrated by the following three possible solutions:

1. Sigurd will die:
   ```
   dies(sigurd).
   ```
2. All humans will die:
   ```
   dies(X) :- human(X).
   ```
3. Everybody but Thor will die:
   ```
   dies(X) :- \+ X=thor.
   ```

The set of all such hypotheses that fit the formal requirements constitute a *hypothesis space* from which the most appropriate answer will be chosen.

The basis of this choice is how well it is estimated to perform when queried on data unseen during the training. From the three hypotheses above, the first one is *overfitting*, meaning that it describes the data too tightly and does not make any generalisations. It will, for example, fail to predict that Gunnar will also die. The last one, on the other hand, is *overgeneralising* and will wrongly predict that Odin will die. An ILP algorithm searches through the hypothesis space trying to strike a balance between overfitting and overgeneralising.

2.1 Background Knowledge and Bias

In the mortality prediction example above, one of the solutions was the following single-clause prolog predicate:

```
dies(X) :- human(X).
```

the body of which consists of a predicate declared as background knowledge. Although some ILP algorithms attempt background theory refinement and predicate invention, e.g. Merlin (Boström, 1996; Boström, 1998) and

Claudien (De Raedt and Dehaspe, 1996), the most straightforward ones employ only the predicates defined as background knowledge to construct the clauses of the target predicate. The background knowledge should, therefore, represent all the facts known about the domain of the concept being learned. The learning task is thus reformulated into that of deciding which background predicates are relevant to the target concept; and how they should be configured in the clauses that comprise the hypothesis being constructed.

One way to influence the outcome of the learning process is to restrict the background knowledge to a subset of all facts known about the domain. Since it is represented in a declarative formalism, it is easy to isolate the parts that represent the knowledge we wish to make accessible to the learning process. This way only the relations and attributes that are a priori deemed interesting will be explored.

This restricted background, however, still allows for a variety of solutions that correctly classify the data. Besides, in some cases it might be desirable to impose restrictions on the hypothesis space without reducing the background knowledge available. To restrict, in other words, the ways the background predicates are configured to form the hypothesis, but not the set of predicates that can be used. For this purpose *bias* is used to describe the form of the learned clauses.

Bias can be expressed as either a preference or a hard constraint. *Preference bias* is realised through the *evaluation function* that quantifies each learned clause's usefulness. So one might, for example, choose an evaluation function that favours shorter clauses over longer ones, all other things being equal. *Syntactic bias,* on the other hand, imposes constraints on the form of the learned clauses. It is enforced by employing a *pruning algorithm* that restricts the hypothesis space by deciding whether the path taken through the hypothesis space should be further pursued or not. It is used to direct the search away from uninteresting clauses, for example ones that are longer that a predetermined maximum length. (See section 2.2 below for more on clause evaluation and pruning.)

ILP allows background knowledge and bias to be expressed in the same terms in which popular symbolic representations for linguistic knowledge and algorithms are expressed. This is of relevance to all aspects of machine learning of natural language:

- In the context of *language learning*, in which it has frequently argued that some innate principles are needed in order to account for language acquisition (as argued by, for example, Noam Chomsky and Steven Pinker), background knowledge can serve to embody the innateness hypothesis more specifically.[2]
- Syntactic bias in general allows for the restriction of the hypothesis space within the limits of one particular meta-theory or theoretical framework. From the point of view of *theoretical linguistics* in particular, it makes it possible to confine the search to one particular formalism. This is not to argue for a 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. The qualitative difference that ILP makes is the ability to express those in an explicit and symbolic formalism (Prolog clauses) that is — for reasons independent from its being employed by ILP – considered to be particularly suitable for knowledge representation.

2.2 Searching the Hypothesis Space

The strategy around which ILP algorithms are built is an adaption of the general *sequential covering* technique, that iterates through steps

[2]It should be emphasised that our interest in modelling some aspects of human acquisition does not extend to every aspect of the psychological problem. We do not aim at simulating children's acquisition, but only claim that the ILP paradigm provides a framework in which putatively innate principles could be readily expressed.

of increasingly wider coverage until some termination criterion —typically depending on the performance of the current solution— is satisfied.

In its ILP incarnation each such iteration constructs a new clause that covers some positive examples that are not explained by the current theory. The individual clauses are constructed by searching through the hypothesis space. This requires that the hypothesis space is (at least partially) ordered and a transversal operator is defined. Both completeness and efficiency are desirable qualities of the search algorithm chosen, but generally not simultaneously achievable. For example an operator that alphabetically enumerates the whole space, imposes a total ordering and guarantees completeness, but is not efficient. ILP algorithms typically structure the hypothesis space along the general–specific axis instead. This means that the search proceeds from a maximally specific clause towards a more general one or vice versa. The generic specific-to-general ILP algorithm can then be outlined as follows:

1. Pick a positive example.
2. Search between the positive example and the maximally general clause by:
   (a) Applying the *generalisation operator* to the clause under consideration.
   (b) Checking if the resulting clause satisfies all syntactic bias constrains and if yes,
   (c) Evaluating it using the *evaluation function.*

   and iterating until the clause under consideration is evaluated as 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.

where the major choices to be made are the evaluation function, the termination criterion and the (generalising or specialising) traverse operator.

The evaluation function is used to measure a constructed clause's 'usefulness'. Its primary function is to achieve the 'golden section' between overgeneralising and overfitting, while at the same time applying preference bias, as mentioned above. Let P be the number of positive examples covered by a clause and N the number of negative ones (incorrectly) covered by that clause. Then one might use as evaluation function a simple *coverage* function $P-N$, the *Laplace function* $\frac{P+1}{P+N+2}$ or some function that also applies preference bias towards shorter clauses, e.g. $P-N-L+1$, 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 De Raedt, 1994, p. 651) or the m-estimate (Dzeroski and Bratko, 1992). The former facilitates learning from positive examples only; the latter is targeted to handling noisy data, i.e. data such that there is no theory satisfying eq. (1) either because they break the prior requirements or because there is no interesting hypothesis that satisfies the posterior ones. (In the sense that the hypothesis that simply enumerates all positive examples is not interesting.)

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.

Any deductive inference rule mapping a conjunction of clauses T onto a conjunction of clauses D such that $T \models D$, can be seen as a specialisation rule, from the point of view where T is some theory that can be used to explain not only D, but other datasets as well. Inversely, any inductive inference rule mapping D onto T is a generalisation rule, since the result is a theory that can be used to explain D, but not only D, and so is more general than D. This is consistent with viewing induction as the inverse of deduction, as put forward in the beginning of this section.

In this sense, given two clauses T and D such that $T \models D$, then we can say that T is more general than D. This way *inverse entailment* can be used as the generalisation operator required for the search. In practice this means picking

a deductive inference operator, such as *resolution* for example, and defining an operator that performs the inverse inference step. (See (Muggleton, 1995) and (Mitchell, 1997, chapt. 10) for the *inverse resolution* operator so derived.)

A simpler and more computationally-efficient inference rule, however, is that of θ-subsumption. Let A and B be clauses expressed as sets of literals; if there is a substitution θ such that $A\theta \subseteq B$, then A θ-subsumes B.

θ-subsumption is efficient, but not as complete as resolution. If A θ-subsumes B, then $A \vdash B$ as well, but the reverse is not true. Consider, for example, the following clauses:

(A)
```
human(X) :- father_of(Y,X),
            human(Y).
```

(B)
```
human(X) :- father_of(Y,X),
            father_of(Z,Y), human(Z).
```

Although $A \vdash B$ (by simply applying A twice), there is no variable substitution θ such that $A\theta \subseteq B$.

Using θ-subsumption to search (in either direction) between a given clause $\{H, \neg B_1, \neg B_2, \ldots, \neg B_n\}$ and the most general clause $\{H\}$ (where H contains no ground terms) effectively means searching through the space of all possible subsets of $\{\neg B_i\}$ for the best body for the clause being constructed. There is, however, one complication: there are several intermediate generalisations between keeping a ground body literal and dropping it. Any number of its argument terms may be replaced by a new variable. The range of the variables introduced must also be restricted by having it appear in some other literal or even a new literal introduced for this purpose.

The search can be organised as a generate-and-test search between the most general clause (only variable arguments in the head and no body literals) and the *least general generalisation* (lgg) of an example. The head of the lgg is the original example where all ground terms have been replaced by variables. The body consists of ground and unground literals, such that the lgg is a syntactically valid clause that explains the example. One way to derive the lgg is by repeatedly applying the inverse resolution operator to the example until all ground terms are replaced by variables, the range of which is appropriately restricted by the body literals.

The objective of the search is to find the subset of the body literals found in the lgg, such that a clause with only those literals in its body scores best according to the evaluation function. The search space can be traversed using any search strategy, preferably the one performing the most informed search based on the background knowledge available. Since the clauses so generated are not guaranteed to be syntactically valid, they need to be tested against the syntactic bias. This is performed by a *pruning algorithm* that forces the search strategy to backtrack whenever a clause is generated that does not conform to the syntactic bias specifications.

3 Syntactic Tagging with Aleph

Aleph is a machine learner developed at Oxford University, implementing the Progol algorithm (Muggleton, 1995). It performs a general-to-specific search between the empty-bodied clause and the lgg, as described above. The task of syntactic tagging has been set up for Aleph by declaring the target theory to be a prolog predicate that relates a word and its context to a syntactic tag. The words to the left are part-of-speech and syntactically tagged whereas the right context is only part-of-speech tagged. Once constructed, such a predicate can be used in a left-to-right pass along a sentence that has been pre-processed by a part-of-speech tagger.

Let us assume that the relevant context extends three words to the left and three words to the right of the word being tagged. The target predicate would then be:

```
tag/4 ( +left_context, +word,
        +right_context, ?syntactic_tag ).
```

The contexts are lists of terms, each term representing a word. The left context terms hold the part-of-speech tag, the actual word-form and the syntactic tag. The right context terms, on the other hand, only carry part-of-speech tags and word-forms. The word to be tagged is also represented as a term holding its part-of-speech tag and word-form. Examples of these two kinds of word terms are:

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

where the two part-of-speech tags stand for 'common noun, singular' and 'adverb'. The tagset used here is the one in the Penn Tree-Bank — from which the dataset was extracted — described at length in (Santorini, 1990). The syntactic tag is simply one of `b`, `i` or `o`, as explained in section 1. In this manner one example can be constructed from each word in the dataset:

```
tag( [w(confidence,nn,i),w(in,in,o)],
     w(the,dt),
     [w(pound,nn),w(is,vbz),
      w(widely,rb)], b ).
```

Negative data is constructed by simply flipping the syntactic tag in a positive example. For this purpose, `b` and `i` 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, one of `b` or `i` is chosen at random. This way the positive example above would yield the following negative one:

```
tag( [w(confidence,nn,i),w(in,in,o)],
     w(the,dt),
     [w(pound,nn),w(is,vbz),
      w(widely,rb)], o ).
```

The background knowledge consists of methods to access the context lists. The

```
prev1_s/2 ( ?List, ?SyntTag )
prev2_s/2 ( ?List, ?SyntTag )
prev3_s/2 ( ?List, ?SyntTag )

prev1_t/2 ( ?List, ?PoSTag )
prev2_t/2 ( ?List, ?PoSTag )
prev3_t/2 ( ?List, ?PoSTag )
```

predicates associate syntactic- and part-of-speech tags with the first, second and third word in the `List` argument, which represents the left context. Only part-of-speech tags are available for the right-hand-side, though, so only the `next?_t/2` collection is defined:

```
next1_t/2 ( ?List, ?PoSTag )
next2_t/2 ( ?List, ?PoSTag )
next3_t/2 ( ?List, ?PoSTag )
```

Positive Examples

	Predicted	Rejected	Percentage
Baseline	76500	17873	81.1%
Theory	86040	8333	91.2%

Negative Examples

	Predicted	Rejected	Percentage
Baseline	4240	90133	95.5%
Theory	4308	90065	95.4%

Table 1: Syntactic Tagging Results

A `naive/2` tagging predicate is also defined, which simply matches each part-of-speech to its most common syntactic tag:

```
naive/2 ( ?PoSTag, ?SyntTag )
```

Its definition is based on frequencies derived from the training set. When used as the baseline theory it scores remarkably well, especially w.r.t. accuracy, (see results, below) and so it is deemed to hold important information.

4 Results

The setup described above was trained on a data set of 922 examples. The evaluation function used was the Laplace function mentioned in section 2.2. The theory learned consisted of 59 clauses. Thirty-four of those cover most of the examples and capture generalisations. The remaining 25 cover only one example each and are exact replications of positive examples that where too exceptional to be covered by any other clause.

The precision and recall of the learned predicate when tested on unseen positive and negative examples, can be seen in table 1. The baseline is the performance of the 'naïve' predicate:

```
tag( _, P, _, S ) :- naive( P, S ).
```

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 accurate figures can be obtained by testing against input pre-processed through a part-of-speech tagger, rather than using the tags provided in the corpus.

It should also be stressed that these predicates are *relations* and not functions, meaning that a predicate might succeed with more that

one syntactic tag matching a particular context. This way backtracking will allow more than one syntactic tags to be tried, hoping that the syntactic tags on the right context will disambiguate. But one should note that the percentages given on table 1 have to be interpreted accordingly, i.e. the figures on the first part of the table mean that for 91.2% of the testing data the predicate succeeded with the correct tag, but it might have also succeeded with an incorrect one; the figures on the second part indicate how often that happened. They can be seen as metrics relating to recall and precision respectively, but are not immediately comparable to the recall and precision of a tagging function.

Using the learned predicate to bracket sentences will yield multiple 'parses'. This is a rather undesirable effect, since it means that parse selection must be employed and some of the gain in speed and simplicity of not using a full parser will be lost. Biasing the algorithm towards learning a function rather than a relation is likely to produce more useful results, despite the fact that the recall rating would drop. (See also section 5 below on the difficulties of implementing bias with the current setup.)

5 Conclusions and Further Work

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 (or even possible) to clearly identify the role of each clause, 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 theory framework rules like, for example, "each XP must include a head X" cannot be easily represented as syntactic bias in the current setup. As already argued in section 2, 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 very choice of ILP for the task is questionable. Therefore, in order to further pursue the application of ILP on syntactic tagging, it would be necessary to develop methodologies or even automated tools to simplify the process of transforming known cross-linguistic generalisations to syntactic bias appropriate for this domain.

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 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.

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 much more interesting task of top-level NP chunking, since top-level NPs will inevitably contain smaller ones in preposition phrases or relative clauses.

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

> [[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:

> [BaseNP/nn in BaseNP/nn] is widely expected to take BaseNP/nn if [BaseNP/nns for BaseNP/nnp]...

This will, of course, only work for a limited, pre-determined recursion length and is prone to over-generalise (since some aspects of the BaseNP that could be relevant are hidden from

the top-layer tagger) but may be sufficient for the purposes of information retrieval or as a preprocessing stage before full parsing.

6 The TMR-LCG Project.

This work is part of the EU funded *Learning Computational Grammars* project (LCG). The objective is the better understanding of the way various machine learning techniques perform on a shared task. This task is recognising the boundaries and internal structure of NPs.

LCG participants investigate how a variety of machine learning approaches can be applied to the shared task, make comparisons and draw conclusions on their applicability to the domain of learning natural language syntax.

References

Henrik Boström. 1996. Theory-guided induction of logic programs by inference of regular languages. In *Proceedings of the 13th International Conference on Machine Learning*, pages 46–53. Morgan Kaufmann.

Henrik Boström. 1998. Predicate invention and learning from positive examples only. In *Proceedings of the Tenth European Conference on Machine Learning*, pages 226–237. Springer Verlag.

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

Luc De Raedt and Luc Dehaspe. 1996. Clausal discovery. Technical Report CW 238, Department of Computing Science, K.U.Leuven, Leuven.

Sasho Dzeroski and I. Bratko. 1992. Handling noise in Inductive Logic Programming. In *Proceedings of the Second International Workshop on Inductive Logic Programming*, Japan. Institute for New Generation Computing.

Tom Mitchell. 1997. *Machine Learning*. McGraw Hill, second edition.

Stephen Muggleton and Luc De Raedt. 1994. Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, 19(20):629–679. Updated version of technical report CW 178, May 1993, Department of Computing Science, K.U. Leuven.

Stephen H. Muggleton. 1995. Inverse entailment and Progol. *New Generation Computing*, 13:245–286.

Lance A. Ramshaw and Mitchell P. Marcus. 1995. Text chunking using transformation-based learning. In *Proceedings of the Third ACL Workshop on Very Large Corpora*. Association for Computational Linguistics.

Beatrice Santorini. 1990. Part-of-speech tagging guidelines for the Penn Treebank project. Technical report, The Penn Treebank Project. 3rd Revision, 2nd Printing (Feb. 1995).