

Learning and Inference for Clause Identification

Xavier Carreras¹*, Lluís Màrquez¹, Vasin Punyakanok², and Dan Roth²

¹ TALP Research Center** – LSI Department
Universitat Politècnica de Catalunya
{carreras,lluism}@lsi.upc.es

² Department of Computer Science***
University of Illinois at Urbana-Champaign
{punyakan,danr}@cs.uiuc.edu

Abstract. This paper presents an approach to partial parsing of natural language sentences that makes global inference on top of the outcome of hierarchically learned local classifiers. The best decomposition of a sentence into clauses is chosen using a dynamic programming based scheme that takes into account previously identified partial solutions. This inference scheme applies learning at several levels—when identifying potential clauses and when scoring partial solutions. The classifiers are trained in a hierarchical fashion, building on previous classifications. The method presented significantly outperforms the best methods known so far for clause identification.

1 Introduction

Partial parsing is studied as an alternative to full-sentence parsing. Rather than producing a complete analysis of sentences, the alternative is to perform only partial analysis of the syntactic structures in a text [6, 1, 5]. There are several possible levels of partial parsing—from the identification of base noun phrases[9] to the identification of several kinds of “chunks” [1, 12] and to the identification of embedded clauses [1].

While earlier work in this direction concentrated on manual construction of rules, most of the recent work has been motivated by the observation that partial syntactic information can be extracted using local information—by examining the pattern itself, its nearby context and the local part-of-speech information. Thus, over the past few years there has been a lot of work on using statistical learning methods to recognize partial parsing patterns—syntactic phrases or words that participate in a syntactic relationship [9, 7, 8, 2, 12, 3]. Earlier learning works on partial parsing have used mostly local classifiers; each detects the beginning or end of a phrase of some type (noun phrase, verb phrase, etc.) or determines, for each word in the sentence, whether it belongs to a phrase

* Supported by a grant from the Catalan Research Department.

** This research is partially funded by the Spanish Research Department (TIC2000-0335-C03-02, TIC2000-1735-C02-02) and the EC (NAMIC IST-1999-12392).

*** Supported by NSF grants IIS-99-84168, ITR-IIS-00-85836 and an ONR MURI award.

or not. Recent work on this problem has achieved significant improvement by using global inference methods to combine the outcomes of these classifiers in a way that provides a coherent inference that satisfies some global constraints, for example, non-overlapping constraints [8, 7]. The work presented here can be viewed as an extension of this approach to a more involved partial parsing problem.

In this paper we study a deeper level of partial parsing, that of clause identification. A clause is a sequence of words in a sentence that contains a subject and a predicate [13]. The problem is to split a sentence into clauses, as in,

(Coach them in (handling complaints) (so that (they can
 resolve problems immediately)) .)

This problem has been found more difficult than simply detecting non-overlapping phrases in sentences [13]. Existing approaches to it use a large number of local classifiers to determine the beginning and end of clauses, as well as the embedding level of the clause.

The work presented here builds on the success of a phrase identification approach that uses inference on top of learned classifier; it develops a scheme that allows the use of global information to combine local classifiers for the finer and more difficult task of identifying embedded clauses. The approach is related also to methods used in bottom-up parsing methods [4]. The key difference is that, as in the inference with classifiers approach in [8], all the information sources used in the inference are derived from hierarchical classifiers that are applied within a recursive scheme. Specifically, the best decomposition of a sentence into clauses is chosen using a dynamic programming scheme that takes into account previously identified partial solutions. This scheme applies learning at several levels, when identifying beginnings and ends of potential clauses and when scoring partial solutions. The classifiers are trained from annotated data in a hierarchical fashion, built on previous classifications.

This work develops a general framework for clause identification that, while being more complex than previous approaches, is derived in a principled way, based on a clear formalism. In particular, the inference scheme can take several scoring functions that could be derived in different ways and make use of different information sources. We exemplify this by experimenting using three different scoring functions.

2 Clause Identification

Basic Definitions. Let w_i be the i -th word in a sentence. Let w_s^t denote the sentence fragment or sequence of words w_s, w_{s+1}, \dots, w_t and, in particular, let w_1^n represent a sentence. In this paper we do not consider clause types. In this setting, thus, a clause c is an element of the set $\mathcal{C} = \{(w_s, w_t) | 1 \leq s \leq t \leq n\}$. For brevity, from now on we will denote a clause simply using the indices of the words of the sentence, and therefore a clause will be an element of the set $\mathcal{C} = \{(s, t) | 1 \leq s \leq t \leq n\}$. Given two clauses $c_1 = (s_1, t_1)$ and $c_2 = (s_2, t_2)$, we

say that c_1 and c_2 are *equal*, denoted by $c_1 = c_2$, iff $s_1 = s_2$ and $t_1 = t_2$. We define that c_1 and c_2 *overlap* iff $s_1 < s_2 \leq t_1 < t_2$ or $s_2 < s_1 \leq t_2 < t_1$, and we note it as $c_1 \sim c_2$. Furthermore, we define that c_1 is *embedded* in c_2 iff $s_2 \leq s_1 \leq t_1 \leq t_2$ and $c_1 \neq c_2$, and we note it as $c_1 < c_2$. A clause split for a sentence is a coherent set of clauses of the sentence, that is, a subset of \mathcal{C} whose clauses do not overlap. Formally, a clause split can be seen as an element S of the set $\mathcal{S} = \{S \subseteq \mathcal{C} \mid \forall c_1, c_2 \in S, c_1 \not\sim c_2\}$.

We will refer to a clause without any embedded clause as a *base clause*, and to a clause which embeds other clauses as a *recursive clause*.

Goal and Evaluation Metrics. The goal of the clause identification problem is to predict a clause split S^P for a sentence which “guesses” the correct split S^C for the sentence. For evaluating the task in a set of N sentences, the usual precision (P), recall (R) and $F_{\beta=1}$ measures are used: ($|\cdot|$: number of elements in set.)

$$P = \frac{\sum_{i=1}^N |S_i^C \cap S_i^P|}{\sum_{i=1}^N |S_i^P|} \quad R = \frac{\sum_{i=1}^N |S_i^C \cap S_i^P|}{\sum_{i=1}^N |S_i^C|} \quad F_{\beta=1} = \frac{2 P R}{P + R}$$

Clause Identification in a Language Processor. A clause splitter is intended to be used after a part-of-speech (POS) tagger and a chunk parser. POS tags are the syntactic categories of words. Chunks are sequences of consecutive words in the sentence which form the basic syntactic phrases, subject to the constraints that chunks cannot overlap or have embedded chunks. In the example

```
( [Balcor]_NP , ( [which]_NP ( [has]_VP [interests]_NP [in]_PP
[real estate]_NP ) ) , [said]_VP ( [the position]_NP [is newly
created]_VP ) . )
```

the chunks are annotated together with its type between square brackets, while the clause split is annotated with parentheses. In a correct syntactic tree, clause boundaries are always at some chunk boundaries. However, in a real system chunk boundaries may be imperfect, so our formalization allows the violation of this constraint.

3 Inference Scheme

The decision scheme used for splitting a sentence into clauses includes two main tasks: 1) identifying single clauses in the sentence, that is, building the set \mathcal{C} ; and 2) selecting the clauses which form the optimal coherent split, that is, choosing the best element S in \mathcal{S} .

3.1 Identifying Clauses

The identification of clauses is done in two steps: the first, identifies candidate clauses in the sentence and the second, scores each candidate. We define an *S point* of the sentence as a word at which clauses may start, and similarly we

define an *E point* of the sentence as a word at which clauses may end. The first step consists of two functions, `spoint(w)` and `epoint(w)` which, given a word w , decide whether it is an S point and an E point, respectively. Each pair of S and E points, where E is not before S, is considered a clause candidate of the sentence. The S and E identification step reduces the space of clauses to be combined to form the solution, as a way to make the problem computationally feasible.

The second step scores each clause candidate of the sentence. This consists of a function `score(s, t)` which, given a clause candidate (s, t) , outputs a real number. Its sign is interpreted as an indication of whether the candidate is a clause (positive) or not; the magnitude of the score is interpreted as the confidence of the decision.

3.2 Selecting the Clause Split

Given a set of scored clauses in the sentence, a coherent subset S must be selected as the clause split for the sentence. Our criterion of optimality for a clause split is the maximization of the summation of the scores of the clauses in the split:

$$S^P = \arg \max_{S \in \mathcal{S}} \sum_{(s,t) \in S} \text{score}(s,t).$$

Given a matrix `BESTSPLIT[s,t]` that for each pair of words w_s and w_t stores the best split found in w_s^t , the best split for the whole sentence can be found at `BESTSPLIT[1,n]`. Using dynamic programming, the matrix can efficiently be filled by exploring the sentence bottom-up.

3.3 A General Algorithm

Although we have described the whole process as two separate tasks, we want to perform them together. The main reason is that when a clause candidate is considered, we want to take advantage of the clause structure that is possibly embedded inside the candidate. The idea is that, syntactically, a clause c_1 acts as an atomic constituent inside a clause c_2 which embeds c_1 so that, when considering c_2 , all the constituents which form c_1 can be reduced to a single constituent, making the structure of c_2 simpler (which may affect the scoring function).

The general algorithm is presented in Fig.1 as a recursive function. Two bi-dimensional matrices are maintained: `BESTSPLIT[s,t]` stores the optimal split found in w_s^t ; `SCORE[s,t]` stores the score for the clause candidate (s, t) . The call to the function `optimal_clause_split(1,n)` explores the whole sentence and stores the optimal clause split for the sentence in `BESTSPLIT[1,n]`.

The first block of the function ensures the completeness of the exploration by making two recursive calls on the sentence fragments, one without the word at the end and the other without the word at the beginning. By induction, after the recursive calls all the clause splits inside the current sentence fragment are identified. The second block of the function computes the optimal split for the

```

function optimal_clause_split (s, t)
  if (s ≠ t) then
    optimal_clause_split(s, t - 1)
    optimal_clause_split(s + 1, t)
  π := { BEST_SPLIT[s, r] ∪ BEST_SPLIT[r + 1, t] | s ≤ r < t }
  S* := arg maxS ∈ π ∑(k,l) ∈ S SCORE[k, l]
  if (spoint(s) and epoint(t)) then
    SCORE[s, t] := score(s, t)
    if (SCORE[s, t] > 0) then
      S* := S* ∪ {(s, t)}
    BEST_SPLIT[s, t] := S*
  end function

```

Fig. 1. General Algorithm for Clause Splitting

current sentence fragment. First, the optimal split is selected as the best union of two disjoint splits which cover the whole fragment. Then, the clause candidate for the current fragment is considered. If the `score` function classifies the current clause as positive, it is added to the optimal split. In the next section we will discuss several settings for this function.

The solution given by the algorithm is guaranteed to be coherent by construction. A clause split is constructed by joining two disjoint clause splits, and only a clause which embeds all the clauses in the split may be added.

Note that the algorithm, as described in Fig.1, repeats recursive calls. This recalculation is not needed and can be easily avoided by keeping track of the visited sentence fragments. It can also be noticed that a function call is relevant only if the fragment considered is bounded by an S point and an E point, and the algorithm can be adapted for avoiding unnecessary calls. In general, a sentence requires a function call for each clause candidate and there is a quadratic number of clause candidates over the n words in the sentence. The function requires a linear time for selecting the optimal split plus the cost of the scoring function. Thus, identifying a clause split in a sentence will take time $O(n^2(n + \text{cost}(\text{score})))$.

3.4 Scoring Functions

In this section we describe particular settings of the function `score`. Given a clause candidate, this function has to predict a score for the candidate being a clause in the sentence. It is defined as a composition of classifiers, each of which, given a clause candidate, output a real number that encodes its prediction (sign) and confidence (magnitude). Below we define such classifiers, and in Sect.4 we describe the learning process. Let (s, t) be the clause candidate to be scored. The variants are the following:

Plain Scoring. The clause structure is not considered. A clause is scored by a classifier, which we refer to as *plain*, which recognizes clauses as plain structures.

This scoring function is independent of the decision taken inside.

$$\text{score}(\mathbf{s}, \mathbf{t}) = \text{plain}(s, t)$$

The cost of this scoring function is the cost of the *plain* classifier.

Structured Scoring. In this setting, the score of a clause depends on its internal structure. This may require exploring all possible subclauses which, to guarantee optimal solutions, may be exponentially expensive. We present here two variations of scoring functions that provide some trade-off between the computational cost and the global optimality.

Let π be the set computed by the algorithm in Fig.1, which contains a linear number of splits, and let S^* be the optimal element in π . We define the cascade function $C(f_1, f_2, \dots, f_n)$ as a function which returns f_1 if $f_1 > 0$ and $C(f_2, \dots, f_n)$ otherwise, having $C() = -1$.

Best Split Scoring. The optimal split is considered for the scoring. The function is composed of three classifiers. A *base* classifier recognizes base clauses, i.e. clauses that do not embed other clauses. A *rec_C* classifier recognizes clauses assuming that the complete split of clauses inside the candidate is given. Finally, a *rec_P* classifier recognizes clauses assuming that only a partial split of clauses is given. When no clause has been identified inside the candidate, the function first applies the *base* classifier, and if it predicts FALSE, it applies the *rec_P* classifier, assuming that initial clauses were missed. When a split is given, the function cascades the three classifiers. The *rec_C* classifier may give an accurate prediction if the split is correct and complete. If it predicts FALSE, the classifier *rec_P* is applied, assuming that some clauses in the split were missed. If it also predicts FALSE, the candidate is tested as a base clause, despite the identified split. This function score depends both on the clauses identified inside and the choice of the optimal split. It is designed to overcome misses in the given split, but incorrect clauses in the split may damage the performance.

$$\text{score}(\mathbf{s}, \mathbf{t}) = \begin{cases} C(\text{base}(s, t), \text{rec}_P(s, t)) & \text{if } S^* = \emptyset \\ C(\text{rec}_C(s, t, S^*), \text{rec}_P(s, t, S^*), \text{base}(s, t)) & \text{otherwise} \end{cases}$$

The computational complexity of this scoring function is the cost of the involved classifiers.

Linear Average Scoring. All the splits in π are considered for scoring the candidate. The function uses the same classifiers as in the Best Split Scoring. The idea here is that the confidence of a clause depends on all the clause structures that can be embedded inside, not only on the optimal. Thus, the score of the clause is given by the function avg^+ which computes the average only over the scores which give positive evidence. As in the previous function, incorrect clauses in the splits may damage the performance.

$$\text{score}(\mathbf{s}, \mathbf{t}) = \begin{cases} C(\text{base}(s, t), \text{rec}_P(s, t)) & \text{if } \pi = \emptyset \\ C(\text{avg}_{S \in \pi}^+ C(\text{rec}_C(s, t, S), \text{rec}_P(s, t, S)), \text{base}(s, t)) & \text{otherwise} \end{cases}$$

This scoring function requires linear exploration of the structure, and hence its cost is n times the cost of the classifiers.

4 Learning the decisions

Here we describe the learning process of the functions involved in the system. When identifying candidates two classifiers are involved, *spoint* and *epoint*. The scoring functions use up to four classifiers, namely *plain*, *base*, *rec_C* and *rec_P*. We use AdaBoost with confidence rated predictions as the learning method.

4.1 AdaBoost

The purpose of boosting algorithms is to find a highly accurate classification rule by combining many *base* classifiers. In this work we use the generalized AdaBoost algorithm presented in [11] by Schapire and Singer. This algorithm has been applied, with significant success, to a number of problems in different research areas, including NLP tasks [10].

Let $(x_1, y_1), \dots, (x_m, y_m)$ be the set of m training examples, where each x_i belongs to an input space \mathcal{X} and $y_i \in \mathcal{Y} = \{+1, -1\}$ is the corresponding class label. AdaBoost learns a number T of base classifiers, each time presenting the base learning algorithm a different weighting over the examples. A base classifier is seen as a function $h : \mathcal{X} \rightarrow \mathbb{R}$. The output of each h_t is a real number whose sign is interpreted as the predicted class, and whose magnitude is the confidence in the prediction. The AdaBoost classifier is a weighted vote of the base classifiers, given by the expression $f(x) = \sum_{t=1}^T \alpha_t h_t(x)$, where α_t represents the weight of h_t inside the whole classifier. Again, the sign of $f(x)$ is the class of the prediction and the magnitude is its confidence.

The base classifiers we use are decision trees of fixed depth. The internal nodes of a decision tree test the value of Boolean predicate (e.g. “the first word of a clause candidate is *that*”). The leaves of a tree define a partition over the input space \mathcal{X} , and each leaf contains the prediction of the tree for the corresponding part of \mathcal{X} . We follow the criterion presented in [11] for growing base decision trees and computing the predictions in the leaves. A maximum depth is used as the stopping criterion.

4.2 Features

An entity to be classified is represented by a set of binary features encoding local and global information in the entity. Features are grouped into several types:

Word Window. A word window of context size n anchored in the word w_i encodes the words in the fragment w_{i-n}^{i+n} along with their position relative to the central word. For each word in the window, its POS forms a feature. For words whose POS are determiners, conjunctions, pronouns or verbs, the form is also a feature. When considered and available, features will also encode whether the words are S or E points.

Chunk Window. A chunk window of context size n anchored in the word w_i codifies the chunk containing the word w_i , the previous n chunks and the following n chunks. For each chunk in the window, a feature is formed with the chunk tag and the distance to the central chunk.

Patterns. A pattern represents the structure of a sentence fragment which is relevant for distinguishing clauses. The following elements are considered: a) Punctuation marks (’, ‘, (,), ,, ., :) and coordinate conjunctions; b) The word “that”; c) Relative pronouns; d) Verb phrases chunks; and e) CLAUSE constituents, already recognized. A pattern for the fragment w_i^j is a feature formed by concatenating the relevant elements inside the fragment.

Element Counts. Number of occurrences of relevant elements in a sentence fragment. Specifically, we consider the chunks which are verb phrases or relative pronouns, the word **that**, and the words whose POS is a punctuation mark. Given a sentence fragment, two features are generated for each element, one indicating the count of the element and the other indicating the existence of the element. If a clause split is given, elements inside clauses will not be counted.

4.3 Training the Classifiers

Classifiers in the decision scheme are used dynamically. Here we describe how to generate a static set of examples from a given set of annotated sentences.

For the S and E identification, each word in the sentence produces an example to be classified. Since clause boundaries, by definition, only appear at chunk boundaries, we consider only the words at the beginning of a chunk as examples for the *spoint* classifier and the words at the end of a chunk as examples for the *epoint* classifier. Consistently, when labeling, the words between chunk boundaries are never considered S or E points. The system works from left to right, by first using the S predictor for the whole sentence and then the E. An example at a word is represented with word and chunk windows, considering the S and E already predicted, and a pattern and counts features for the fragments of the sentence before and after the word.

The classifiers in the scoring function receive clause candidates as examples to be classified. The candidates are generated by the S and E identification so, clearly, the classifiers of the scoring functions depend on the performance of the S and E classifiers. In training, given a set of sentences, examples of candidates are generated with the correct set of S and E points plus a set of incorrect points which depends on the previously learned classifiers. Our criterion for selecting such incorrect points is to use negative examples which are closer to the decision boundary of the *spoint* and *epoint* classifiers.

Given a set of candidates, we generate and typify training examples into four positive labels (‘+1’-‘+4’) and two negative labels (‘-1’, ‘-2’) as follows:

- Each candidate which is a base clause generates one example of type ‘+1’.

- Each candidate which is a recursive clause generates: 1) one example of type ‘+2’, without considering its internal clause split; 2) one example of type ‘+3’, considering its complete clause split; and 3) k examples of type ‘+4’, each considering one of the k partial splits formed by removing clauses from the complete split for up to three levels deep.
- Each candidate which is not a clause generates: 1) one example of type ‘-1’, without considering any clauses inside; and 2) k examples of type ‘-2’ considering possible splits with the clauses inside the candidate generated as in examples of type ‘+4’.

For training, the *plain* classifier takes positive examples of type ‘+1’ and ‘+2’, and negative examples of type ‘-1’. The *base* classifier takes ‘+1’ positive examples and ‘-1’ negative examples. The *rec_C* takes ‘+3’ for positives and ‘-2’ for negatives. Finally, the *rec_P* takes positive examples of type ‘+2’, ‘+3’ and ‘+4’, and negative examples of both types.

In these classifiers, a candidate is represented by word and chunk windows anchored both in the S and E point of the candidate, a pattern codifying the structure of the candidate and counts of the relevant elements in the candidate. Note that when a clause split is considered within a candidate, clauses in the split are represented in the pattern as reduced elements and elements inside the clauses are not counted.

5 Experiments

In this section we describe the experiments we performed to evaluate the presented algorithm with its variations.

CoNLL 2001 Corpus. We used the Penn Treebank as data for training and testing the clause system, following the setting of the CoNLL 2001 shared task [13]. WSJ sections from 15 to 18 were used as training material (8,936 sentences), section 20 as development material (2,012 sentences), and 21 as test data (1,671 sentences).³ The data sets contain sentences with the words, the clause split solution, and automatically tagged POS tags and chunks.

Baseline: Open-Close. The best system presented in the CoNLL task, which we call the Open-Close [3], is used as the baseline for comparison. The scheme it follows first identifies the S and E points in a sentence. Then, an *open* classifier decides how many open brackets correspond to each S point. After that, for each open bracket a *close* classifier scores each E point as closing point for the bracket. A final procedure ensures the coherence of the solution by choosing the most confident decisions that form a correct split. The Open-Close scheme has a close relation to the Plain Scoring approach presented in this work, in the sense that the *close* and the *plain* classifiers score clause candidates in a similar way.

³ Corpus freely available at <http://lcg-www.iaa.ac.be/conll2001/clauses>.

Training classifiers. All the classifiers involved in the scheme were trained using base decision trees of depth 4 (four levels of predicates plus the leaves with the predictions). Initial experiments showed a great improvement in using depths around 4 rather than the usual decision stumps (depth 1). Only features with more than three occurrences in the training data were considered. Up to 4,000 trees were learned for each classifier, and the optimal number was selected as the one with the best $F_{\beta=1}$ measure on the development set.

Evaluating the scoring functions. In the first experiment we compared the performance of the three proposed scoring functions. The classifiers involved in the functions were learned without considering incorrect S and E points in the training set. Table 1 shows in the first three columns the results for each scoring function on the development set, together with the results of the Open-Close. The performance of the S and E points identification was 93.89% and 90.12% in F_1 , respectively.

Table 1. Results on the development set. Overall performance of the presented variants, using the predicted (left) or the correct (right) S and E points.

	predicted S E			correct S E		
	prec.	rec.	$F_{\beta=1}$	prec.	rec.	$F_{\beta=1}$
Open-Close	87.18%	82.48%	84.77%	98.12%	96.16%	97.13%
Plain Scoring	88.33%	83.92%	86.07%	95.04%	96.14%	95.59%
Best Split Scoring	89.10%	83.92%	86.44%	96.74%	96.90%	96.82%
Linear Average Scoring	87.60%	81.91%	84.66%	95.88%	95.74%	95.81%
Robust Best Split Scoring	92.53%	82.48%	87.22%	97.47%	90.29%	93.74%

Regarding the three results, the Best Split Scoring obtained the best rates. The Plain Scoring obtained the same recall but less precision. Our hypothesis is that considering reduced clauses simplifies the structures to be classified and yields more precise predictions. The Linear Average Scoring is significantly worse than the other variants. Thus, it seems that in this problem taking into account the optimal identified structure helps the decisions, but further explorations of non-optimal solutions confuses the decisions. Comparing to the best results in CoNLL, both the Best Split and Plain Scoring variants significantly outperformed the Open-Close method.

In order to show the bottleneck that the S and E identification introduced, we ran the systems considering the correct S and E points instead of using the predictions. The results are shown in the right side of Table 1. In this ideal setting, the performance is very good, clearly indicating that errors in the S and E layer significantly affect the general performance. The Best Split Scoring is again better than the other scoring variants. Here Open-Close achieves a very high precision, possibly due to a heuristics which opens a clause at each S point.

Robust Training. The false positive errors in the S and E identification produce clause candidates that have not been considered when training the scoring clas-

sifiers. In this experiment we retrained such classifiers generating better sets of negative examples, and exploring different sizes of negative examples. As described in Sect.4.3, such training examples were generated with the correct set of S and E points plus a $P\%$ of incorrect points, selecting those which were close to the decision boundary of the learned *spoint* and *epoint* classifiers, respectively. We used P values ranging from 0 to 100. In general, the higher the P the more precise were the classifiers we obtained. The Plain Scoring, despite the improvement in precision, did not improve the F measure because the recall rates dropped faster. The Linear Average Scoring slightly improved its F rate, but did not outperform the other variants on the default training. Finally, the Best Split Scoring obtained significant improvements: the best performance was achieved when adding a 20% of incorrect S predictions and 40% incorrect E predictions, giving an F rate of 87.22%. Table 1 (left) shows the results only for this improved model. Naturally, the performance using the correct S and E (Table 1 right) deteriorated when incorrect predictions were also used.

Table 2 presents the results obtained by the different scoring functions on the test set, together with the Open-Close results and the S and E performance. As observed in the systems tested in the CoNLL competition [13], the test set seems to be harder than the development set. Again, the Best Split Scoring performs significantly better than the other approaches, and the robust training of the function, with the setting tuned on the development, yields a significant improvement in precision and the F measure.

Table 2. Results on the test set, both for the S and E identification (above) and the general performance of the scoring functions (below).

	prec.	rec.	$F_{\beta=1}$
S points	93.96%	89.59%	91.72%
E points	90.04%	88.41%	89.22%
Open-Close	84.82%	73.28%	78.63%
Plain Scoring	85.25%	74.53%	79.53%
Best Split Scoring	86.44%	74.41%	79.98%
Linear Average Scoring	86.53%	72.54%	78.92%
Robust Best Split Scoring	90.18%	72.59%	80.44%

6 Conclusions and Future Work

We have presented a framework for the identification of embedded structure in sentences and investigated experimentally several instantiations of it. All the decisions involved in the scheme are derived using learned classifiers, and thus it is a scheme for doing inference with classifiers. We have shown that this approach improves over the top-performing clausing system. Moreover, we believe that the general framework developed here can be generalized to the identification of embedded structures in other structure learning problems, such as information

extraction problems and other natural language processing, and this is one of the important direction that we intend to explore in future work.

Several questions remain open with respect to the specific problem studied here. The key one is that of incorporating the chunk parsing stage into the framework rather than using its outputs. The idea is to maintain the ambiguity in the classification longer, perhaps until it can be resolved using other information sources, as our framework suggests. Other problems include investigating the use of additional linguistic knowledge, such as the type of the clause, and avoiding the significant bottleneck introduced by the S and E layer.

References

1. S. P. Abney. Parsing by chunks. In R. C. Berwick, S. P. Abney, and C. Tenny, editors, *Principle-based parsing: Computation and Psycholinguistics*, pages 257–278. Kluwer, Dordrecht, 1991.
2. S. Buchholz, J. Veenstra, and W. Daelemans. Cascaded grammatical relation assignment. In *EMNLP-VLC'99, the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, June 1999.
3. X. Carreras and L. Màrquez. Boosting trees for clause splitting. In *Proceedings of CoNLL-2001*, pages 73–75. Toulouse, France, 2001.
4. J. Goodman. Parsing algorithms and metrics. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 177–183, 1996.
5. G. Grefenstette. Evaluation techniques for automatic semantic extraction: comparing semantic and window based approaches. In *ACL'93 workshop on the Acquisition of Lexical Knowledge from Text*, 1993.
6. Z. S. Harris. Co-occurrence and transformation in linguistic structure. *Language*, 33(3):283–340, 1957.
7. M. Muñoz, V. Punyakanok, D. Roth, and D. Zimak. A learning approach to shallow parsing. In *EMNLP-VLC'99, the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, June 1999.
8. V. Punyakanok and D. Roth. The use of classifiers in sequential inference. In *NIPS-13; The 2000 Conference on Advances in Neural Information Processing Systems*, 2001.
9. L. A. Ramshaw and M. P. Marcus. Text chunking using transformation-based learning. In *Proceedings of the Third Annual Workshop on Very Large Corpora*, 1995.
10. R. Schapire. The Boosting Approach To Machine Learning: An Overview. In *Proceedings of the MSRI Workshop on Nonlinear Estimation and Classification*, 2002.
11. R. E. Schapire and Y. Singer. Improved Boosting Algorithms Using Confidence-rated Predictions. *Machine Learning*, 37(3):297–336, 1999.
12. E. F. Tjong Kim Sang and S. Buchholz. Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of CoNLL-2000 and LLL-2000*, pages 127–132, 2000.
13. E. F. Tjong Kim Sang and H. Déjean. Introduction to the CoNLL-2001 shared task: Clause identification. In W. Daelemans and R. Zajac, editors, *Proceedings of CoNLL-2001*, pages 53–57. Toulouse, France, 2001.