# Projective Dependency Parsing with Perceptron

**Xavier Carreras, Mihai Surdeanu** and **Lluís Màrquez**
TALP Research Centre – Software Department (LSI)
Technical University of Catalonia (UPC)
Campus Nord - Edifici Omega, Jordi Girona Salgado 1–3, E-08034 Barcelona
{carreras,surdeanu,lluism}@lsi.upc.edu

## Abstract

We describe an online learning dependency parser for the CoNLL-X Shared Task, based on the bottom-up projective algorithm of Eisner (2000). We experiment with a large feature set that models: the tokens involved in dependencies and their immediate context, the surface-text distance between tokens, and the syntactic context dominated by each dependency. In experiments, the treatment of multilingual information was totally blind.

## 1 Introduction

We describe a learning system for the CoNLL-X Shared Task on multilingual dependency parsing (Buchholz et al., 2006), for 13 different languages.

Our system is a bottom-up projective dependency parser, based on the cubic-time algorithm by Eisner (1996; 2000). The parser uses a learning function that scores all possible labeled dependencies. This function is trained globally with online Perceptron, by parsing training sentences and correcting its parameters based on the parsing mistakes. The features used to score, while based on the previous work in dependency parsing (McDonald et al., 2005), introduce some novel concepts such as better codification of context and surface distances, and runtime information from dependencies previously parsed.

Regarding experimentation, the treatment of multilingual data has been totally blind, with no special processing or features that depend on the language. Considering its simplicity, our system achieves moderate but encouraging results, with an overall labeled attachment accuracy of 74.72% on the CoNLL-X test set.

## 2 Parsing and Learning Algorithms

This section describes the three main components of the dependency parsing: the parsing model, the parsing algorithm, and the learning algorithm.

### 2.1 Model

Let $1, \ldots, L$ be the dependency labels, defined beforehand. Let $x$ be a sentence of $n$ words, $x_1 \ldots x_n$. Finally, let $\mathcal{Y}(x)$ be the space of well-formed dependency trees for $x$. A dependency tree $y \in \mathcal{Y}(x)$ is a set of $n$ dependencies of the form $[h, m, l]$, where $h$ is the index of the head word ($0 \leq h \leq n$, where 0 means root), $m$ is the index of the modifier word ($1 \leq m \leq n$), and $l$ is the dependency label ($1 \leq l \leq L$). Each word of $x$ participates as a modifier in exactly one dependency of $y$.

Our dependency parser, $\mathrm{dp}$, returns the maximum scored dependency tree for a sentence $x$:

$$\mathrm{dp}(x, \mathbf{w}) = \operatorname*{arg\,max}_{y \in \mathcal{Y}(x)} \sum_{[h,m,l] \in y} \mathrm{sco}([h, m, l], x, y, \mathbf{w})$$

In the formula, $\mathbf{w}$ is the weight vector of the parser, that is, the set of parameters used to score dependencies during the parsing process. It is formed by a concatenation of L weight vectors, one for each dependency label, $\mathbf{w} = (\mathbf{w}^1, \ldots, \mathbf{w}^l, \ldots, \mathbf{w}^L)$. We assume a feature extraction function, $\phi$, that represents an unlabeled dependency $[h, m]$ in a vector of $D$ features. Each of the $\mathbf{w}^l$ has $D$ parameters or dimensions, one for each feature. Thus, the global weight vector **w** maintains $L \times D$ parameters. The scoring function is defined as follows:

$$\mathrm{sco}([h, m, l], x, y, \mathbf{w}) = \phi(h, m, x, y) \cdot \mathbf{w}^l$$

Note that the scoring of a dependency makes use of $y$, the tree that contains the dependency. As described next, at scoring time $y$ just contains the dependencies found between $h$ and $m$.

### 2.2 Parsing Algorithm

We use the cubic-time algorithm for dependency parsing proposed by Eisner (1996; 2000). This parsing algorithm assumes that trees are projective, that is, dependencies never cross in a tree. While this assumption clearly does not hold in the CoNLL-X data (only Chinese trees are actually 100% projective), we chose this algorithm for simplicity. As it will be shown, the percentage of non-projective dependencies is not very high, and clearly the error rates we obtain are caused by other major factors.

The parser is a bottom-up dynamic programming algorithm that visits sentence spans of increasing length. In a given span, from word $s$ to word $e$, it completes two partial dependency trees that cover all words within the span: one rooted at $s$ and the other rooted at $e$. This is done in two steps. First, the optimal dependency structure internal to the span is chosen, by combining partial solutions from internal spans. This structure is completed with a dependency covering the whole span, in two ways: from $s$ to $e$, and from $e$ to $s$. In each case, the scoring function is used to select the dependency label that maximizes the score.

We take advantage of this two-step processing to introduce features for the scoring function that represent *some* of the internal dependencies of the span (see Section 3 for details). It has to be noted that the parsing algorithm we use does not score dependencies on top of every possible internal structure. Thus, by conditioning on features extracted from $y$ we are making the search approximative.

### 2.3 Perceptron Learning

As learning algorithm, we use Perceptron tailored for structured scenarios, proposed by Collins (2002). In recent years, Perceptron has been used in a number of Natural Language Learning works, such as in partial parsing (Carreras et al., 2005) or even dependency parsing (McDonald et al., 2005).

```
$\mathbf{w} = \mathbf{0}$
for $t = 1$ to $T$
    foreach training example $(x, y)$ do
        $\hat{y} = \mathrm{dp}(x, \mathbf{w})$
        foreach $[h, m, l] \in y \backslash \hat{y}$ do
            $\mathbf{w}^l = \mathbf{w}^l + \phi(h, m, x, \hat{y})$
        foreach $[h, m, l] \in \hat{y} \backslash y$ do
            $\mathbf{w}^l = \mathbf{w}^l - \phi(h, m, x, \hat{y})$
return $\mathbf{w}$
```

Figure 1: Pseudocode of the Perceptron Algorithm. $T$ is a parameter that indicates the number of epochs that the algorithm cycles the training set.

Perceptron is an online learning algorithm that learns by correcting mistakes made by the parser when visiting training sentences. The algorithm is extremely simple, and its cost in time and memory is independent from the size of the training corpora. In terms of efficiency, though, the parsing algorithm must be run at every training sentence.

Our system uses the regular Perceptron working in primal form. Figure 1 sketches the code. Given the number of languages and dependency types in the CoNLL-X exercise, we found prohibitive to work with a dual version of Perceptron, that would allow the use of a kernel function to expand features.

## 3 Features

The feature extraction function, $\phi(h, m, x, y)$, represents in a feature vector a dependency from word positions $m$ to $h$, in the context of a sentence $x$ and a dependency tree $y$. As usual in discriminative learning, we work with binary indicator features: if a certain feature is observed in an instance, the value of that feature is 1; otherwise, the value is 0. For convenience, we describe $\phi$ as a composition of several base feature extraction functions. Each extracts a number of disjoint features. The feature extraction function $\phi(h, m, x, y)$ is calculated as:

$$\phi_{token}(x, h, \text{``}head\text{''}) + \phi_{tctx}(x, h, \text{``}head\text{''}) + \phi_{token}(x, m, \text{``}mod\text{''}) + \phi_{tctx}(x, m, \text{``}mod\text{''}) + \phi_{dep}(x, mmd_{h,m}) + \phi_{dctx}(x, mmd_{h,m}) + \phi_{dist}(x, mmd_{h,m}) + \phi_{runtime}(x, y, h, m, d_{h,m})$$

where $\phi_{token}$ extracts context-independent token features, $\phi_{tctx}$ computes context-based token features, $\phi_{dep}$ computes context-independent depen-

| $\phi_{\mathbf{token}}(\mathbf{x}, \mathbf{i}, \mathbf{type})$ |
|---|
| $type \cdot w(x_i)$ |
| $type \cdot l(x_i)$ |
| $type \cdot cp(x_i)$ |
| $type \cdot fp(x_i)$ |
| foreach($ms$): $type \cdot ms(x_i)$ |
| $type \cdot w(x_i) \cdot cp(x_i)$ |
| foreach($ms$): $type \cdot w(x_i) \cdot ms(x_i)$ |
| **$\phi_{\mathbf{tctx}}(\mathbf{x}, \mathbf{i}, \mathbf{type})$** |
| $\phi_{token}(x, i-1, type \cdot string(i-1))$ |
| $\phi_{token}(x, i-2, type \cdot string(i-2))$ |
| $\phi_{token}(x, i+1, type \cdot string(i+1))$ |
| $\phi_{token}(x, i+2, type \cdot string(i+2))$ |
| $type \cdot cp(x_i) \cdot cp(x_{i-1})$ |
| $type \cdot cp(x_i) \cdot cp(x_{i-1}) \cdot cp(x_{i-2})$ |
| $type \cdot cp(x_i) \cdot cp(x_{i+1})$ |
| $type \cdot cp(x_i) \cdot cp(x_{i+1}) \cdot cp(x_{i+2})$ |

Table 1: Token features, both context-independent ($\phi_{token}$) and context-based ($\phi_{tctx}$). $type$ - token type, i.e. "head" or "mod", $w$ - token word, $l$ - token lemma, $cp$ - token coarse part-of-speech (POS) tag, $fp$ - token fine-grained POS tag, $ms$ - token morpho-syntactic feature. The $\cdot$ operator stands for string concatenation.

| $\phi_{\mathbf{dep}}(\mathbf{x}, \mathbf{i}, \mathbf{j}, \mathbf{dir})$ |
|---|
| $dir \cdot w(x_i) \cdot cp(x_i) \cdot w(x_j) \cdot cp(x_j)$ |
| $dir \cdot cp(x_i) \cdot w(x_j) \cdot cp(x_j)$ |
| $dir \cdot w(x_i) \cdot w(x_j) \cdot cp(x_j)$ |
| $dir \cdot w(x_i) \cdot cp(x_i) \cdot cp(x_j)$ |
| $dir \cdot w(x_i) \cdot cp(x_i) \cdot w(x_j)$ |
| $dir \cdot w(x_i) \cdot w(x_j)$ |
| $dir \cdot cp(x_i) \cdot cp(x_j)$ |
| **$\phi_{\mathbf{dctx}}(\mathbf{x}, \mathbf{i}, \mathbf{j}, \mathbf{dir})$** |
| $dir \cdot cp(x_i) \cdot cp(x_{i+1}) \cdot cp(x_{j-1}) \cdot cp(x_j)$ |
| $dir \cdot cp(x_{i-1}) \cdot cp(x_i) \cdot cp(x_{j-1}) \cdot cp(x_j)$ |
| $dir \cdot cp(x_i) \cdot cp(x_{i+1}) \cdot cp(x_j) \cdot cp(x_{j+1})$ |
| $dir \cdot cp(x_{i-1}) \cdot cp(x_i) \cdot cp(x_j) \cdot cp(x_{j+1})$ |

Table 2: Dependency features, both context-independent ($\phi_{dep}$) and context-based ($\phi_{dctx}$), between two points $i$ and $j$, $i < j$. $dir$ - dependency direction: left to right or right to left.

dency features, $\phi_{dctx}$ extracts contextual dependency features, $\phi_{dist}$ calculates surface-distance features between the two tokens, and finally, $\phi_{runtime}$ computes dynamic features at runtime based on the dependencies previously built for the given interval during the bottom-up parsing. $mmd_{h,m}$ is a shorthand for a triple of numbers: $\min(h, m)$, $\max(h, m)$ and $d_{h,m}$ (a sign indicating the direction, i.e., $+1$ if $m < h$, and $-1$ otherwise).

We detail the token features in Table 1, the dependency features in Table 2, and the surface-distance features in Table 3. Most of these features are inspired by previous work in dependency parsing (McDonald et al., 2005; Collins, 1999). What is important for the work presented here is that we construct explicit feature combinations (see above tables) because we configured our linear predictors in primal form, in order to keep training times reasonable.

| $\phi_{\mathbf{dist}}(\mathbf{x}, \mathbf{i}, \mathbf{j}, \mathbf{dir})$ |
|---|
| foreach(k $\in (i, j)$): $dir \cdot cp(x_i) \cdot cp(x_k) \cdot cp(x_j)$ |
| number of tokens between $i$ and $j$ |
| number of verbs between $i$ and $j$ |
| number of coordinations between $i$ and $j$ |
| number of punctuations signs between $i$ and $j$ |

Table 3: Surface distance features between points $i$ and $j$. Numeric features are discretized using "binning" to a small number of intervals.

| $\phi_{\mathbf{runtime}}(\mathbf{x}, \mathbf{y}, \mathbf{h}, \mathbf{m}, \mathbf{dir})$ |
|---|
| let $l_1, \ldots, l_S$ be the labels of dependencies in $y$ that attach to $h$ and are found from $m$ to $h$. |
| foreach $i$, $1 \le i \le S$ : $dir \cdot cp(x_h) \cdot cp(x_m) \cdot l_i$ |
| if $S \ge 1$ , $dir \cdot cp(x_h) \cdot cp(x_m) \cdot l_1$ |
| if $S \ge 2$ , $dir \cdot cp(x_h) \cdot cp(x_m) \cdot l_1 \cdot l_2$ |
| if $S \ge 3$ , $dir \cdot cp(x_h) \cdot cp(x_m) \cdot l_1 \cdot l_2 \cdot l_3$ |
| if $S \ge 4$ , $dir \cdot cp(x_h) \cdot cp(x_m) \cdot l_1 \cdot l_2 \cdot l_3 \cdot l_4$ |
| if $S = 0$ , $dir \cdot cp(x_h) \cdot cp(x_m) \cdot null$ |
| if $0 < S \le 4$ , $dir \cdot cp(x_h) \cdot cp(x_m) \cdot regular$ |
| if $S > 4$ , $dir \cdot cp(x_h) \cdot cp(x_m) \cdot big$ |

Table 4: Runtime features of $y$ between $m$ and $h$.

While the features presented in Tables 1, 2, and 3 are straightforward exploitations of the training data, the runtime features ($\phi_{runtime}$) take a different, and to our knowledge novel in the proposed framework, approach: for a dependency from $m$ to $h$, they represent the dependencies found between $m$ and $h$ that attach also to $h$. They are described in detail in Table 4. As we have noted above, these features are possible because of the parsing scheme, which scores a dependency only after all dependencies spanned by it are scored.

## 4 Experiments and Results

We experimented on the 13 languages proposed in the CoNLL-X Shared Task (Hajič et al., 2004; Simov et al., 2005; Simov and Osenova, 2003; Chen et al., 2003; Böhmová et al., 2003; Kromann, 2003; van der Beek et al., 2002; Brants et al., 2002; Kawata and Bartels, 2000; Afonso et al., 2002; Džeroski et al., 2006; Civit and Martí, 2002; Nilsson et al., 2005; Oflazer et al., 2003; Atalay et al., 2003). Our approach to deal with many different languages was totally blind: we did not inspect the data to motivate language-specific features or processes.

We did feature filtering based on frequency counts. Our feature extraction patterns, that exploit both lexicalization and combination, generate millions of feature dimensions, even with small datasets. Our criterion was to use at most 500,000 different dimensions in each label weight vector. For each language, we generated all possible features, and then filtered out most of them according to the counts. Depending on the number of training sentences, our counts cut-offs vary from 3 to 15.

For each language, we held out from training data a portion of sentences (300, 500 or 1000 depending on the total number of sentences) and trained a model for up to 20 epochs in the rest of the data. We evaluated each model on the held out data for different number of training epochs, and selected the optimum point. Then, we retrained each model on the whole training set for the selected number of epochs.

Table 5 shows the attachment scores obtained by our system, both unlabeled (UAS) and labeled (LAS). The first column (GOLD) presents the LAS obtained with a perfect scoring function: the loss in accuracy is related to the projectivity assumption of our parsing algorithm. Dutch turns out to be the most non-projective language, with a loss in accuracy of 5.44%. In our opinion, the loss in other languages is relatively small, and is not a major limitation to achieve a high performance in the task. Our system achieves an overall LAS of 74.72%, with substantial variation from one language to another. Turkish, Arabic, Dutch, Slovene and Czech turn out to be the most difficult languages for our system, with accuracies below 70%. The easiest language is clearly Japanese, with a LAS of 88.13%, followed by Chinese, Portuguese, Bulgarian and German, all with LAS above 80%.

Table 6 shows the contribution of base feature extraction functions. For four languages, we trained models that increasingly incorporate base functions. It can be shown that all functions contribute to a better score. Contextual features ($\phi_3$) bring the system to the final order of performance, while distance ($\phi_4$) and runtime ($\phi$) features still yield substantial improvements.

## 5 Analysis and Conclusions

It is difficult to explain the difference in performance across languages. Nevertheless, we have identified four generic factors that we believe caused the most errors across all languages:

|  | GOLD | UAS | LAS |
|---|---|---|---|
| Bulgarian | 99.56 | 88.81 | **83.30** |
| Arabic | 99.76 | 72.65 | **60.94** |
| Chinese | 100.0 | 88.65 | **83.68** |
| Czech | 97.78 | 77.44 | **68.82** |
| Danish | 99.18 | 85.67 | **79.74** |
| Dutch | 94.56 | 71.39 | **67.25** |
| German | 98.84 | 85.90 | **82.41** |
| Japanese | 99.16 | 90.79 | **88.13** |
| Portuguese | 98.54 | 87.76 | **83.37** |
| Slovene | 98.38 | 77.72 | **68.43** |
| Spanish | 99.96 | 80.77 | **77.16** |
| Swedish | 99.64 | 85.54 | **78.65** |
| Turkish | 98.41 | 70.05 | **58.06** |
| Overall | 98.68 | 81.19 | **74.72** |

Table 5: Results of the system on test data. GOLD: labeled attachment score using gold scoring functions; the loss in accuracy is caused by the projectivity assumption made by the parser. UAS : unlabeled attachment score. LAS : labeled attachment score, the measure to compare systems in CoNLL-X. Bulgarian is excluded from overall scores.

|  | $\phi_1$ | $\phi_2$ | $\phi_3$ | $\phi_4$ | $\phi$ |
|---|---|---|---|---|---|
| Turkish | 33.02 | 48.00 | 55.33 | 57.16 | 58.06 |
| Spanish | 12.80 | 53.80 | 68.18 | 74.27 | 77.16 |
| Portuguese | 47.10 | 64.74 | 80.89 | 82.89 | 83.37 |
| Japanese | 38.78 | 78.13 | 86.87 | 88.27 | 88.13 |

Table 6: Labeled attachment scores at increasing feature configurations. $\phi_1$ uses only $\phi_{token}$ at the head and modifier. $\phi_2$ extends $\phi_1$ with $\phi_{dep}$. $\phi_3$ incorporates context features, namely $\phi_{tctx}$ at the head and modifier, and $\phi_{dctx}$. $\phi_4$ extends $\phi_3$ with $\phi_{dist}$. Finally, the final feature extraction function $\phi$ increases $\phi_4$ with $\phi_{runtime}$.

**Size of training sets**: the relation between the amount of training data and performance is strongly supported in learning theory. We saw the same relation in this evaluation: for Turkish, Arabic, and Slovene, languages with limited number of training sentences, our system obtains accuracies below 70%. However, one can not argue that the training size is the only cause of errors: Czech has the largest training set, and our accuracy is also below 70%.

**Modeling large distance dependencies**: even though we include features to model the distance between two dependency words ($\phi_{dist}$), our analysis indicates that these features fail to capture all the intricacies that exist in large-distance dependencies. Table 7 shows that, for the two languages analyzed, the system performance decreases sharply as the distance between dependency tokens increases.

|  | to root | 1 | 2 | $3-6$ | $>=7$ |
|---|---|---|---|---|---|
| Spanish | 83.04 | 93.44 | 86.46 | 69.97 | 61.48 |
| Portuguese | 90.81 | 96.49 | 90.79 | 74.76 | 69.01 |

Table 7: $F_{\beta=1}$ score related to dependency token distance.

**Modeling context**: many attachment decisions, e.g. prepositional attachment, depend on additional context outside of the two dependency tokens. To address this issue, we have included in our model features to capture context, both static ($\phi_{dctx}$ and $\phi_{tctx}$) and dynamic ($\phi_{runtime}$). Nevertheless, our error analysis indicates that our model is not rich enough to capture the context required to address complex dependencies. All the top 5 focus words with the majority of errors for Spanish and Portuguese – "y", "de", "a", "en", and "que" for Spanish, and "em", "de", "a", "e", and "para" for Portuguese – indicate complex dependencies such as prepositional attachments or coordinations.

**Projectivity assumption**: Dutch is the language with most crossing dependencies in this evaluation, and the accuracy we obtain is below 70%.

**On the Degree of Lexicalization** We conclude the error analysis of our model with a look at the degree of lexicalization in our model. A quick analysis of our model on the test data indicates that only 34.80% of the dependencies for Spanish and 42.94% of the dependencies for Portuguese are fully lexicalized, i.e. both the head and modifier words appear in the model feature set (see Table 8). There are two reasons that cause our model to be largely unlexicalized: (a) in order to keep training times reasonable we performed heavy filtering of all features based on their frequency, which eliminates many lexicalized features from the final model, and (b) due to the small size of most of the training corpora, most lexicalized features simply do not appear in the testing section. Considering these results, a reasonable question to ask is: how much are we losing because of this lack of lexical information? We give an approximate answer by analyzing the percentage of fully-lexicalized dependencies that are correctly parsed by our model. Assuming that our model scales well, the accuracy on fully-lexicalized dependencies is an indication for the gain (or loss) to be had from lexicalization. Our model parses fully-lexicalized dependencies with an

|  | Fully lexicalized | One token unlexicalized | Fully unlexicalized |
|---|---|---|---|
| Spanish | 34.80% | 54.77% | 10.43% |
| Portuguese | 42.94% | 49.26% | 7.80% |

Table 8: Degree of dependency lexicalization.

accuracy of 74.81% LAS for Spanish (2.35% *lower* than the overall score) and of 83.77% LAS for Portuguese (0.40% higher than the overall score). This analysis indicates that our model has limited gains (if any) from lexicalization.

In order to improve the quality of our dependency parser we will focus on previously reported issues that can be addressed by a parsing model: large-distance dependencies, better modeling of context, and non-projective parsing algorithms.

## Acknowledgements

This work was partially funded by the European Union Comission (PASCAL - IST-2002-506778) and Spanish Ministry of Science and Technology (TRANGRAM - TIN2004-07925-C03-02). Mihai Surdeanu was supported by a Ramon y Cajal fellowship of the latter institution.

## References

S. Buchholz, E. Marsi, A. Dubey, and Y. Krymolowski. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proc. of the Tenth Conf. on Computational Natural Language Learning (CoNLL-X)*. SIGNLL.

X. Carreras, Lluís Màrquez, and J. Castro. 2005. Filtering-ranking perceptron learning for partial parsing. *Machine Learning*, 1–3(60):41–71.

M. Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

M. Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proc. of EMNLP-2002*.

J. Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proc. of the 16th Intern. Conf. on Computational Linguistics (COLING)*.

J. Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In H. C. Bunt and A. Nijholt, editors, *New Developments in Natural Language Parsing*, pages 29–62. Kluwer Academic Publishers.

R. McDonald, K. Crammer, and F. Pereira. 2005. Online large-margin training of dependency parsers. In *Proc. of the 43rd Annual Meeting of the ACL*.