Natural Language Processing

Lecture 6
Dependency parsing

2021

The dependency parsing task

Syntactic parsing (refresher)

Syntactic theories aim to characterize

> "the set of rules or principles that govern how words are put together to form phrases, well formed sequences of words." [Koopman et al., 2013, 1]

The most important "well formed sequences" in this context are *sentences*: the central goal of syntactic theories for a given language is to find structural rules or principles that characterize/delineate well formed sentences of the language in question.

Syntactic parsing cont.

A sentence is well formed if it has a *structural description* or *syntactic parse* which satisfies the syntactic constraints of the theory in question. Syntactic well formedness doesn't guarantee coherence or meaningfulness. To use Chomsky's famous example:

Colorless green ideas sleep furiously.

is syntactically well formed but nonsensical, while

Furiously sleep ideas green colorless.

is not even well formed.

Dependency grammars (refresher)

Dependency grammars treat the *dependency relation* between words as fundamental.
The precise criteria vary from theory to theory, but typically a d word depends on a h word (equivalently, h heads d) in a sentence if

- d modifies the meaning of h, makes it more specific, e.g. *eats* $\Rightarrow$ *eats bread*, *eats slowly* etc.
- and there is an asymmetric relationship of omissibility between them: d can be omitted from the sentence keeping h but not vice versa.

Dependency grammars cont.

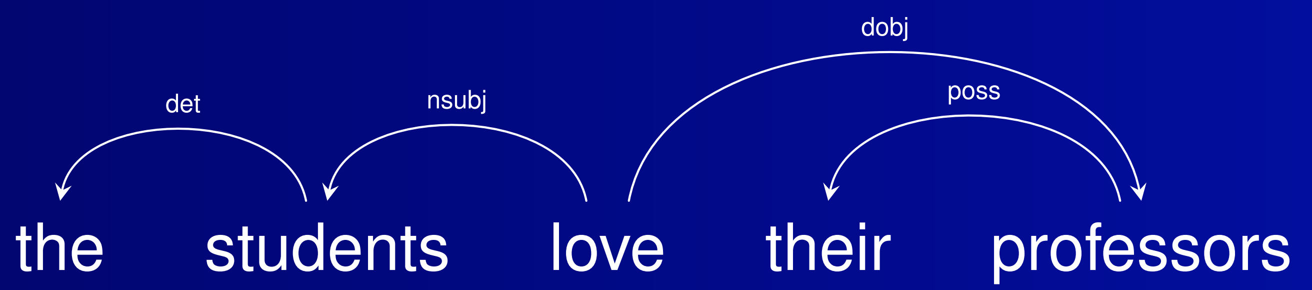
Dependency grammars impose important global constraints on the dependency relations within a well formed sentence, e.g.,

- There is exactly one independent word (the root of the sentence).
- All other words depend directly on exactly one word.

As a consequence of the constraints, the direct dependency graph of a sentence is a tree.
Most dependency grammars work with *typed direct dependencies*: there is finite list of direct dependency types with specific constraints on when they can hold.

Dependency grammars cont.

A dependency parse tree of the earlier example:



Compared to the constituency tree, it contains fewer nodes (one per word), but the edges are labeled with the corresponding dependency types.

Projectivity

An important (not always satisfied) requirement on dependency parse trees is *projectivity*:

> If a w word depends directly on h and a w' word lies between them in the sentence's word order, then the head of this w' is either w or h, or another word between them.

Less formally, the projectivity condition states that dependencies are *nested*, there cannot be *crossing* dependencies between words.

Projectivity cont.

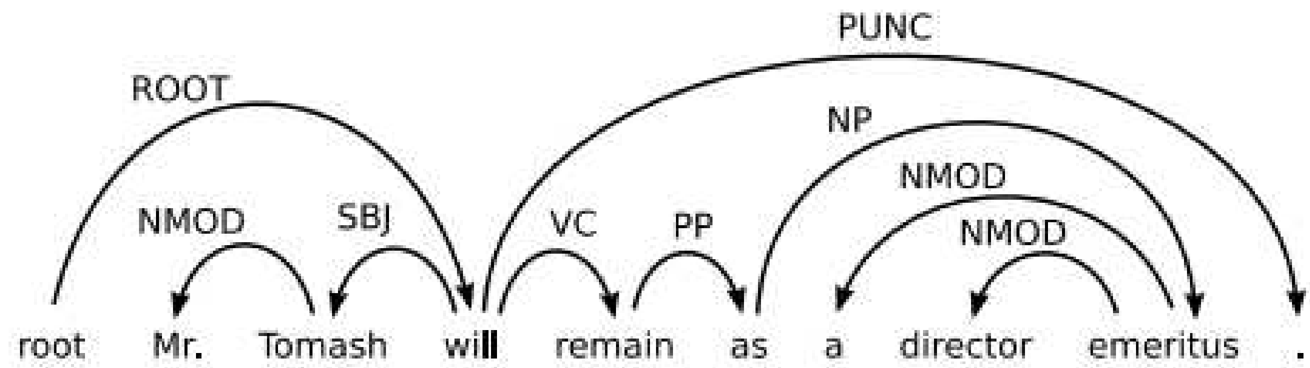


Figure 1: A projective dependency graph.

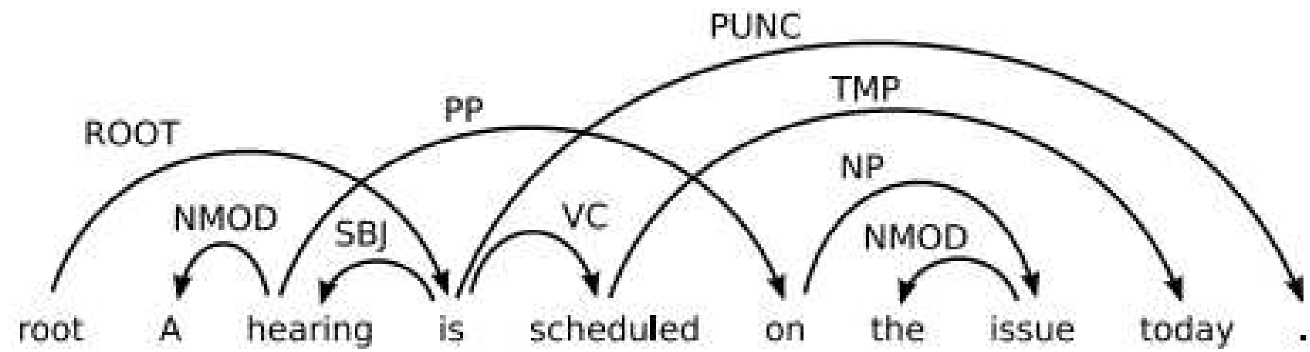


Figure 2: Non-projective dependency graph.

(Figure from Language Log: Non-projective flavor.)

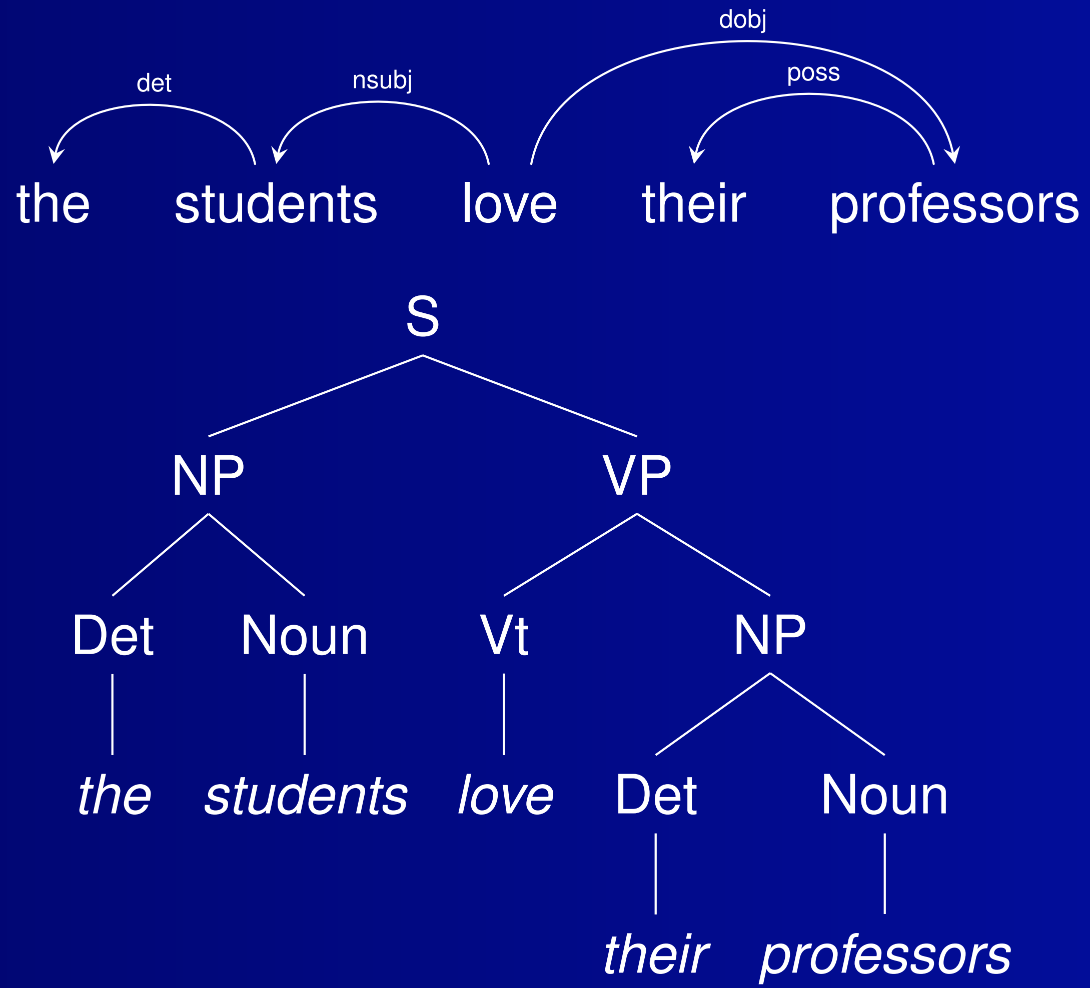
The advantages of dependency grammars

Dependency grammars are becoming the dominant syntactic theory used in NLP, since

- dependency trees are in many respect simpler structures than phrase structure parse trees (e.g., have only one node per word);
- the predicate-argument analysis of sentences provided by dependency graphs is a very good starting point for event or frame-oriented semantic analysis.

Usability for semantic representation

Compare, for event-semantic aspects



The dependency parsing task

Given a syntactic theory, the parsing task is to assign syntactic structure to input sentences which satisfy the constraints/conditions of the theory. For dependency grammars, this means assigning a *dependency structure*:

- identifying direct dependencies between words of the sentence,
- in such a way that together they constitute a *dependecy tree* which satisfies all of the the theory's constraints.

The dependency parsing task cont.

In modern NLP practice, the dependency grammar underlying a parsing task is typically specified implicitly, using a so called *treebank*, that is, a dataset consisting of sentences annotated with their parse trees.

This makes parsing a *structured supervised learning task*: given a training set consisting of a large number of $\langle \text{sentence}, \text{parse tree} \rangle$ pairs, learn to predict the parse tree of unseen sentences.

Performance metrics

For dependency grammar parsers, the most commonly used evaluation metrics are

- *UAS (Unlabeled Attachment Score):* The percentage of words that are attached to the correct head.
- *LAS (Labeled Attachment Score):* The percentage of words that are attached to the correct head with the correct dependency label.

Parsing algorithms

Like most sequence tagging approaches, dependency parsing algorithms use the strategy of breaking down the prediction task into individual decisions over elements of the structure. In this case,

- the individual decisions are about individual dependencies between words, and
- the central problem is to ensure that the individual decisions lead to a coherent dependency tree.

Dependency parsers typically use either a

- *transition-based*, or
- *graph-based* approach.

Transition-based parsing

The transition-based approach

The algorithm is based on a formal model of a parsing process which moves from left to right in the sentence to be parsed and at every step chooses one of the following actions:

- "assign the current word as the head of some previously seen word,
- assign some previously seen word as the head of the current word,
- or postpone doing anything with the current word, adding it to a store for later processing."[1]

[1] Jurafsky and Martin [2019, ch. 15].

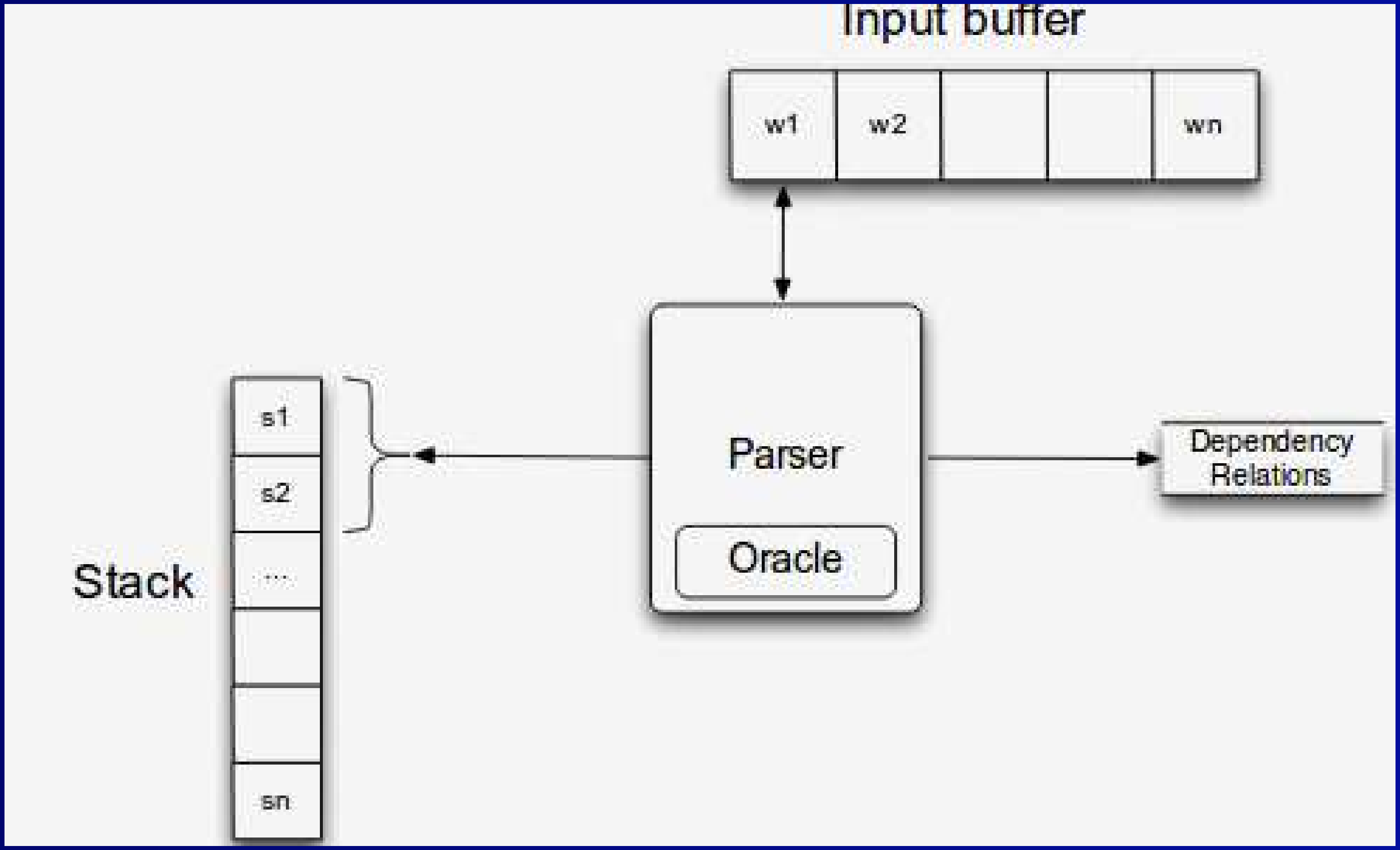
The transition-based approach

The formal model of this process consists of the following component:

- a *buffer*, in which the unprocessed tokens of the input are contained;
- a *stack* containing tokens for current operation and storing postponed elements;
- a *dependency graph*, which is being built for the input sentence.

Model configuration

The model is in a certain *configuration* at every step of the process:



(Figure from Jurafsky and Martin [2019, ch. 15].)

Initial configuration

The parsing process starts with the special initial configuration in which

- the buffer contains all words of the input,
- the stack contains the single root node of the dependency graph,
- and the dependency graph is empty (contains no dependency edges).

Parsing process

At every step, one of the permitted configuration manipulating actions (configuration transitions) are performed. The permitted actions vary; a very simple set of actions is used in the so called *arc standard* approach:

- *left arc with label l*: add edge $s_2 \xleftarrow{l} s_1$ to the graph and remove s_2 (s_2 cannot be the root element);
- *right arc with label l*: add edge $s_2 \xrightarrow{l} s_1$ to the graph and remove s_1 (s_1 cannot be the root element);
- *shift*: remove the first word w_1 from the buffer and put it on the top of the stack.

The process ends when a configuration is reached in which none of the actions can be performed.

Parsing process cont.

The process is guaranteed to end after a finite number of steps, in a configuration in which the buffer is empty and the created dependency graph is a well-formed dependency tree for the whole input:

- it ends because at every step we decrease the "collective token distance from the dep. graph" $2 \cdot \#(\text{tokens in buffer}) + \#(\text{tokens in stack})$;
- the buffer must be empty because otherwise the shift action would be available, and the stack can contain only the root element for similar reasons;
- each input token has exactly one head in the graph;
- there cannot be a *circle* in the graph.

Parsing process cont.

An example run from Jurafsky and Martin [2019, ch. 16]:

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Figure 14.7 Trace of a transition-based parse.

Choosing the right action

How does a parser decide which action to choose? The model has to act as a *classifier over possible configurations*: if there are n labels, then there will be $2n + 1$ actions/classes.

To have training data for this classifier, dependency treebank annotations have to be turned into supervised datasets containing

$$\langle \text{parser configuration}, \text{correct action} \rangle$$

pairs, i.e., treebanks have to be turned into datasets about the actions of a "*parsing oracle*", which always chooses the right action.

Converting a parse tree “to oracle actions”

Given the correct parse tree, the configurations and actions of the *oracle* can be reconstructed using a straightforward algorithm:

- (obviously) start with the stack containing only the root and a buffer with the full input;
- choose the *left arc* action with the correct label if it leads to a correct edge,
- else choose the *right arc* action with the correct label if (i) it leads to a correct edge (ii) all dependencies with s_1 as head were already added to to the dependency graph;
- otherwise choose shift.

Alternative action/transitions sets

Arc-standard is not the only transition system used for transition-based parsers – an important alternative is *arc-eager*, which can radically simplify some derivations. Arc-eager has the following actions:

- *Right-arc*: add edge $s_1 \xrightarrow{l} w_1$ and move w_1 to the top of the stack.
- *Left-arc*: add edge $s_1 \xleftarrow{l} w_1$ and remove w_1 from the buffer. Precondition: s_1 does not have a head yet.
- *Shift*: move w_1 to the top of the stack.
- *Reduce*: remove s_1 from the stack. Precondition: s_1 already has a head.

The problem of non-projectivity

Arc-standard and arc-eager transitions can produce only projective trees, but most treebanks contain a sizeable amount of non-projective sentences:

Language	Trees	Arcs
Arabic [Hajič et al. 2004]	11.2%	0.4%
Basque [Aduriz et al. 2003]	26.2%	2.9%
Czech [Hajič et al. 2001]	23.2%	1.9%
Danish [Kromann 2003]	15.6%	1.0%
Greek [Prokopidis et al. 2005]	20.3%	1.1%
Russian [Boguslavsky et al. 2000]	10.6%	0.9%
Slovene [Džeroski et al. 2006]	22.2%	1.9%
Turkish [Oflazer et al. 2003]	11.6%	1.5%

(Table from Nivre [2013].)

Non-projectivity: solutions

- Use transition systems that can create (a certain amount of) non-projective edges.
- *Pseudo-projective parsing*:
  - find a φ mapping between all relevant (projective + non-projective) trees and projective ones;
  - for training, the training set is "projectivized" using φ, and the parser is trained on the transformed dataset;
  - for prediction/inference, φ^{-1} is applied to the parser's output to get the final (possibly non-projective) result.[2]

[2] See, e.g., Nivre and Nilsson [2005] for details.

Classifier features

Proper feature extraction from configurations is important for performance. Traditional (e.g., perceptron-based) solutions used complex, expert-engineered feature templates, e.g.,

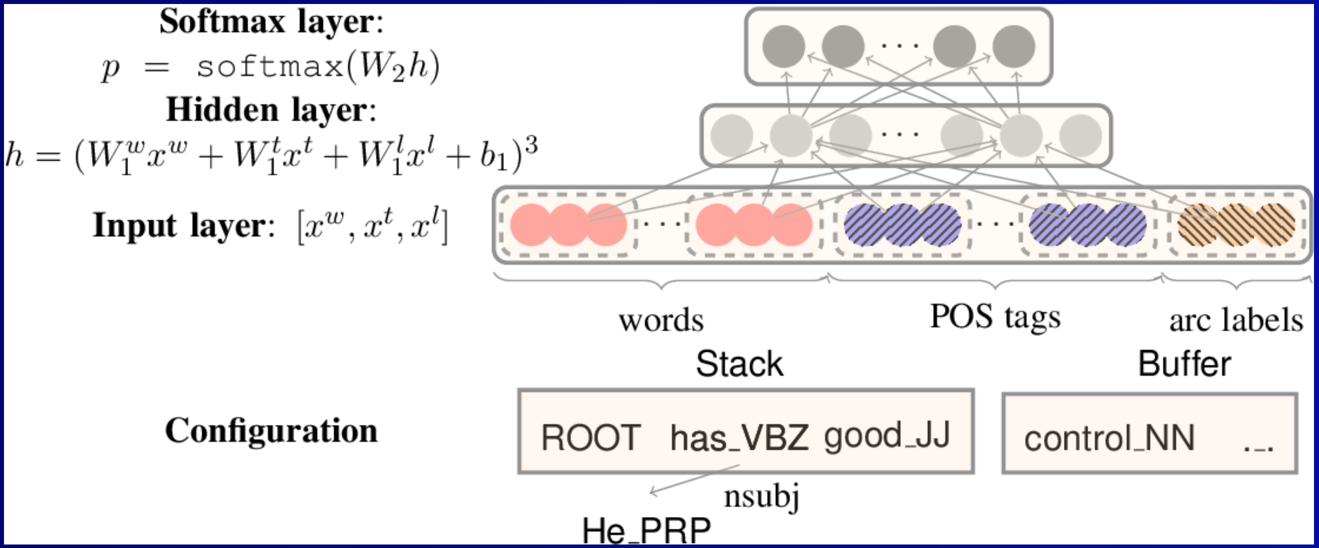
Type	Features		
Unigram	s_t	$T(s_t)$	$s_t \circ T(s_t)$
	s_{t-1}	$T(s_{t-1})$	$s_{t-1} \circ T(s_{t-1})$
	w_i	$T(w_i)$	$w_i \circ T(w_i)$
Bigram	$s_t \circ s_{t-1}$	$T(s_t) \circ T(s_{t-1})$	$T(s_t) \circ T(w_i)$
	$T(s_t) \circ s_{t-1} \circ T(s_{t-1})$	$s_t \circ s_{t-1} \circ T(s_{t-1})$	$s_t \circ T(s_t) \circ T(s_{t-1})$
	$s_t \circ T(s_t) \circ s_{t-1}$	$s_t \circ T(s_t) \circ s_{t-1} \circ T(s_{t-1})$	
Trigram	$T(s_t) \circ T(w_i) \circ T(w_{i+1})$	$T(s_{t-1}) \circ T(s_t) \circ T(w_i)$	$T(s_{t-2}) \circ T(s_{t-1}) \circ T(s_t)$
	$s_t \circ T(w_i) \circ T(w_{i+1})$	$T(s_{t-1}) \circ s_t \circ T(w_i)$	
Modifier	$T(s_{t-1}) \circ T(lc(s_{t-1})) \circ T(s_t)$	$T(s_{t-1}) \circ T(rc(s_{t-1})) \circ T(s_t)$	$T(s_{t-1}) \circ T(s_t) \circ T(lc(s_t))$
	$T(s_{t-1}) \circ T(s_t) \circ T(rc(s_t))$	$T(s_{t-1}) \circ T(lc(s_{t-1})) \circ s_t$	$T(s_{t-1}) \circ T(rc(s_{t-1})) \circ s_t$
	$T(s_{t-1}) \circ s_t \circ T(lc(s_t))$		

Table 2: Feature templates of the baseline parser. s_t, s_{t-1} denote the top and next to top words on the stack; w_i and w_{i+1} denote the current and next words on the queue. $T(\cdot)$ denotes the POS tag of a given word, and $lc(\cdot)$ and $rc(\cdot)$ represent the leftmost and rightmost child. Symbol $\circ$ denotes feature conjunction. Each of these templates is further conjoined with the 3 actions shift, reduce_L, and reduce_R.

(Table from Huang et al. [2009].)

Classifier features cont.

As in other areas, the problems with manual feature engineering and data sparsity led to the development of deep learning solutions, which rely on *embeddings* for classification. The Stanford neural dependency parser is a simple but representative example:



(Figure from from Chen and Manning [2014].)

Architectures

The used model architectures are the typical classification architectures used in NLP:

- Before the emergence of DL-based parsers, mainly linear models were used (with weighted perceptron or SVM as the learning algorithm), but k-NN-based solutions also existed.
- In deep learning, CNN and LSTM-based models were dominant before the appearance of transformer-based solutions, which rely heavily on pretrained contextual embeddings such as BERT.

Graph-based parsing

The graph-based approach

Two contrasting ways of scoring parse trees:

- The *transition-based* approach transforms the problem of scoring a dependency-graph into scoring the *steps* of a somewhat complicated *graph building process*.
- *Graph-based* parsers, in contrast, score directly the graphs themselves and try to find the dependency graph with the maximal score:

$$\hat{g} = \operatorname*{argmax}_{g \in G} S(g)$$

The graph-based approach cont.

A simple but surprisingly well performing approach is to

- score all the possible edges individually (this requires scoring $n(n-1)l$ directed edges if there are n tokens and l labels), and then
- find the (correctly directed) tree with the largest sum total score.

The assumption is simply that

$$S(g) = \sum_{e \in g} S(e).$$

This way of scoring a graph is called the *edge-* or *arc-factored* approach.

Finding the tree with the maximal score

A brute-force search over all possible graphs would be obviously unfeasible. Fortunately, there are relatively fast algorithms for finding the maximally scoring tree (the so-called *maximum spanning tree*).

A frequently used algorithm is the *Chu–Liu–Edmonds algorithm*, which has time complexity $\mathcal{O}(n^3 l)$ for n input tokens and l possible labels, what can be reduced to $\mathcal{O}(n^2 l)$ by storing the edge scores in a special data structure, a so-called Fibonacci-heap.

Edge scoring features

Graph-based dependency parsers are *regressors*: they have to produce scores for the possible edges between the input tokens. The used feature templates are analogous to those in transition-based parsers:

- the dependent and its affixes, POS etc.;
- the head and its affixes, POS etc;
- the edge label;
- the relationship between the head and the dependent in the sentence, e.g. their distance;
- for neural architectures, embeddings for the nodes and the label of the edge.

Architectures

Analogously to the transition-based case, both classic ML and neural graph-based parsers have been developed over the years, the highest performing parsers using self-attention layers.

An important aspect of some of the recent architectures, introduced by a paper by Dozat and Manning [2016], is that they use different sets of embeddings for the head and dependent representations of the same words.

Transition- vs graph-based parsing

There are important trade-offs between the two approaches.

Time complexity: the time-complexity of parsing n tokens with l possible edge labels is

- typically $\mathcal{O}(n)$ for transition-based parsers, while
- graph-based parsers precompute scores for all possible edges, so they start with an $\mathcal{O}(n^2 l)$ operation, and the time of finding the maximum spanning tree is added to this. Even if we treat finding labels as a separate task the $\mathcal{O}(n^2)$ complexity is inescapable.

Transition- vs graph-based parsing cont.

Non-projectivity: as we have seen, non-projectivity is a serious problem for the most wide-spread transition systems which needs special treatment. Graph-based approaches don not suffer from this problem.

Performance: Transition-based systems tend to have problems with long-distance dependencies, graph-based models do not have this performance issue. As a consequence, the dependency parser leader boards are dominated by graph-based systems.

References

References

Danqi Chen and Christopher D Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014. URL https://nlp.stanford.edu/pubs/emnlp2014-depparser.pdf.

Timothy Dozat and Christopher D Manning. Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734*, 2016. URL https://arxiv.org/pdf/1611.01734.pdf.

Liang Huang, Wenbin Jiang, and Qun Liu. Bilingually-constrained (monolingual) shift-reduce parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1222–1231, 2009. URL https://dl.acm.org/doi/10.5555/1699648.1699668.

Daniel Jurafsky and James H Martin. Speech and language processing (3rd ed. draft). 2019. URL https://web.stanford.edu/~jurafsky/slp3/.

Hilda Koopman, Dominique Sportiche, and Edward Stabler. *An introduction to syntactic analysis and theory*. John Wiley & Sons, 2013. URL https://linguistics.ucla.edu/people/stabler/isat.pdf.

References cont.

Joakim Nivre and Jens Nilsson. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 99–106, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics. doi: 10.3115/1219840.1219853. URL https://www.aclweb.org/anthology/P05-1013.

Joakim Nivre. Beyond MaltParser. Presentation. 2013. URL https://bit.ly/3q2jm86.