Speech and Language Processing

by Jurafsky, Martin

14 Dependency Parsing

Dependency grammars are another family of formalisms distinct from constituency-based approaches. The syntactic structure of a sentence is described in terms of its words and a set of directed binary grammatical relations among them.

A typed dependency structure is one in which the words' labels are drawn from a fixed inventory of relations, and includes a root node.

Dependency grammars's main advantage lies in their ability to deal with free word order languages. Additionally, the dependency relations provide an approximation to the semantic relationship between the words.

14.1 Dependency Relations

The traditional notion of grammatical relations is the basis for dependency relations, consisting of a head and a dependent. Dependency grammar allows us to classify the relations (or functions) in terms of the role of the dependent.

14.2 Dependency Formalisms

In their most general form, dependency structures are just directed graphs. Vertices typically correspond to words, but they might also correspond to morphemes or punctuation.

Sometimes, further constraints are imposed – common ones are connectivity, rootedness, acyclicity and planarity. We only consider rooted trees here. Thus, the following constraints are applicable:

- there is a root node with no incoming arcs.
- all other nodes have one incoming arc.
- there is a unique path from the root node to each vertex in V.

14.2.1 Projectivity

Projectivity is another constraint based on the order of words in the input. An arc is said to be projective if there is a path from the head to every word that lies between the head and the dependent; a tree is projective is all the arcs that make it up are projective.

In other words, a tree is projective if it can be drawn with no crossing arcs. Trees generated from phrase-structure treebanks are guaranteed to be projective by the nature of CFGs.

14.3 Dependency Treebanks

14.4 Transition-Based Dependency Parsing

Our first approach is motivated by shift-reduce parsing (a stack-based approach for PLs). It uses a CFG, a stack and a list of tokens. Input tokens are pushed onto the stack and the top two elements are matched against the rules in the grammar; if there is a match they are popped and the non-terminal producing them is pushed (reducing).

We modify this approach to replace the "reduce" action with an action asserting a head-dependent relation (either between the top element and the one below it, or vice versa).

A key element in this algorithm is the "configuration", consisting of a stack, an input buffer of tokens and a set of relations. Then the parsing process is nothing but a sequence of transitions through configurations. The goal is to reach a configuration where all words have been read and a tree has been synthesised.

This is essentially searching the configuration space, for which we all define transition operators (which produce new configurations from old ones). The initial configuration will have a stack with a ROOT node, the complete input word list, and an empty set of relations.

We define three operators:

- leftArc: assert a head-dependent relation between the word at the top of the stack and the one below it, remove the latter.
- rightArc: assert a head-dependent relation between the second word in the stack and the top one, remove the former.
- shift: move a word from the front of the input buffer to the top of the stack.

These operators constitute the arc standard approach to this parsing method.

There are, however, some restrictions on the application of these operators – both require two elements on the stack, and leftArc cannot be applied when ROOT is the second element. Thus we have the basic algorithm:

```
function DependencyParse(words) returns dependencytree
    state <- {[root], [words], []}
    while state not final
        t <- Oracle(state)
        state <- Apply(t,state)
    return state
```

The oracle provides the correct transition operator to use.

Three important things to be notes are: more than one sequence can lead to a correct parse, especially (but not only) in case of ambiguities; the oracle may not always be accurate, making a more thorough search necessary; and the arcs

generated are not labelled (they can be by parameterising and expanding the operators).

14.4.1 Creating an Oracle

Usually, supervised ML methods are used to train oracles.

Generating Training Data We use a training oracle to create a training set, given a corpus of gold standard (reference) parses. We can make it proceed as follows:

- choose leftArc if it produces a correct relation; else,
- choose rightArc if it produces a correct relation and all the dependents of the top word have been assigned; else
- choose shift.

Features We need to extract features from configurations to train classifiers, like morphosyntactic markings, parts of speech and so on. We usually focus on the top levels of the stack and the words near the front of the buffer (along with the relations already existing among these words) for feature extraction.

Learning

14.4.2 Advanced Methos in Transition-Based Parsing

Alternative Transition System In the arc standard method, dependents are removed from the stack as soon as they are assigned to their heads, which means that many words cannot be assigned to their heads until all their dependents have passed through the stack. Usually this delay is not problematic, but it could be in some cases.The arc-eager system, on the other hand, allows words to be attached to their heads as soon as they are available; it defines the operators as:

- leftArc: assert a relation between the word at the front of the buffer and that at the top of the stack, pop the stack.
- rightArc: assert a relation between the word on top of the stack and that at the front of the buffer, push the latter on the stack.
- shift: push the word at the front of the input buffer on the stack.
- reduce: pop the stack.

Beam Search Beam search explores the configuration space using BFS combined with a heuristic filter to prune the search frontier (keeping it within a fixed beam width).

We apply all applicable operators to each state on an agenda and score the results, adding them to the frontier as long as it stays below the beam width. When it exceeds it, we only add new configurations that are better than the

worst one on the agenda. We continue this as long as there are non-final states on the agenda.

The score of a configuration will now depend on its entire history; for example,

$$\text{ConfigScore}(c_0) = 0.0$$

$$\text{ConfigScore}(c_i) = \text{ConfigScore}(c_{i-1}) + \text{Score}(t_i, c_{i-1})$$

Thus we get the beam search algorithm as:

```
function DependencyBeamParse(words, width) returns dependencytree
    state <- {[root], [words], [], 0.0}
    agenda <- <state>

    while agenda contains non-final states
        newagenda <- 0
        for each state \in agenda do
            for all {t | t \in ValidOps(state)} do
                child <- Apply(t, state)
                newagenda <- AddToBeam(child, newagenda, width)
            agenda <- newagenda
        return BestOf(agenda)

    function AddToBeam(state, agenda, width) returns updatedAgenda
        if Length(agenda) < width then
            agenda <- Insert(state, agenda)
        else if Score(state) > Score(WorstOf(agenda))
            agenda <- Remove(WorstOf(agenda))
            agenda <- Insert(state, agenda)
        return agenda
```

14.5 Graph-Based Dependency Parsing

Graph-based approaches search through the space of trees for a tree that maximises some score. The overall score for a tree is typically the sum of the scores of its edges.

Graph-based methods are motivated by their ability to produce non-projective trees and their accuracy in long-distance dependencies.

14.5.1 Parsing

This approach begins with a fully connected, weighted, directed graph whose vertices represent input words. An additional ROOT node has edges going to all other vertices, and the weights represent the score for each head-dependent relation. An MST of this graph emanating from ROOT represents the parse of this sentence.

We begin with greedy selection (picking the highest-weighted incoming node for each edge). If this does not lead to a tree, we execute a cleanup algorithm. Here, first we subtract the score of the maximum edge entering each node from the weights of all edges entering it. Thus all edges selected earlier have a weight of zero now.

Then a cycle is selected and collapsed into single new node; edges that enter or leave the cycle are altered to enter to leave this node. Edges in the cycle are dropped and those outside it are included. We then recursively apply the algorithm to find the MST of the new graph, after which we delete the edge (from the cycle) directed towards the single node representing the cycle.

```
function MaxSpanningTree(G = VE, root, score) returns spanningtree
    F <- []
    T' <- []
    score' <- []
    for eachv \in V do
        bestInEdge <- argmax_{(u,v) \in E} score(u,v)
        F <- F \cup bestInEdge
        for each (u,v) \in E do
            score'(u,v) <- score(u,v) - score[bestInEdge]
        if T = VF is a spanning tree return it
        else
            C <- a cycle in F
            G' <- Contract(G,C)
            T' <- MaxSpannignTree(G', root, score')
            T <- Expand(T', C)
            return T

function Contract(G, C) returns contractedgraph
function Expand(T, C) returns expandedgraph
```

This algorithm runs in $O(mn^2) = O(n^3)$ time, but it can be improved to $O(m + nlogn)$.

14.5.2 Features and Training

Edge-factored parsing models reduce the score for a tree to the sum of the scores of its edges. Each edge score is a weighted sum of features extracted from it.

14.6 Evaluation

The exact match (EM) metric is too pessimistic to be useful. Thus the most common methods are labelled and unlabelled attachment accuracy, where accuracy is the percentage of words in an input that are assigned to the correct head with the correct relation.

If we want to judge the performance w.r.t. a single relation, we use the notions

of precision (percentage of correct labellings out of all labellings) and recall (percentage of identified correct labellings out of all correct labellings).

14.7 Summary

1. In dependency-based approaches, the structure is described as a set of binary relations.
2. The relations capture the head-dependent relationships among words.
3. Dependency-based analysis provides information useful in information extraction, semantic parsing and question answering.
4. Transition-based parsing employs a greedy stack-based algorithm.
5. Graph-based parsing uses MSTs.
6. Both approaches make use of supervised ML techniques.
7. Treebanks provide the data needed for training.
8. Evaluation is based on labelled and unlabelled accuracy scores.