Write-Up
A Transition-Based Semantic Dependency Parser
Fang Wang

The project is a two-list transition-based semantic dependency parser implemented from scratch with C++. It supports graph represented dependencies.

1. Method
I adopted a structured perceptron and a two-list transition system to realize my dependency parser. I adopted beam search algorithm when calculate each example.

Two-list Transition System
I adopted two-list transition system to allow non-projective dependencies. A strict definition could be found in (Nivre, 2008). Here, we give a description of this transition system.
Containers:
List-1: contains the tokens have not been calculated whether or not there would be an arc between them and the token on the top of Buffer. Initialized as empty.
List-2: contains the tokens have already been calculated whether or not there would be an arc between them and the token on the top of Buffer. The tokens in List-2 are in the middle of List-1 and Buffer. Initialized as empty.
Buffer: contains the tokens to come. Initialized as the complete token sequence. We the Buffer becomes empty, we obtain a finished configuration.
Transitions:
SHIFT: remove the first node of Buffer and all tokens before Buffer are assign to List-1. (List-2 is empty.)
RIGHT-ARC (label): adds a dependency edge from the head of List-1 to the first node in Buffer. And move the head of List-1 into List-2.
LEFT-ARC (label): adds a dependency edge from the first node in Buffer to head of List-1. And move the head of List-1 into List-2.
NO-ARC: move the head of List-1 into List-2.

Feature Engineering
Lexical features are extracted including the form, lemma, pos, and the combination of these items from tokens on specific positions of a configuration. Those positions contains the head of LIST-1, the start and the end of LIST-2, and the top of Buffer.
Structural features are build on a combination of already predicted dependencies and the tokens on specific positions. The (lest/right-most) child(ren) and parent of those tokens and the relation (the label) together with the token construct structural features.

The features used in the project are listed in the table below.

◦ represent combination: e.g. *token.form* ◦*pos* represents the combination of the word itself and its POS tag

Lexical Features	B0.form, B0.lemma, B0.pos, B0.form◦pos, X1.form, X1.lemma, X1.pos, X1.form◦pos, X2.lemma, X2.pos, X2.form◦pos, B1.lemma, B1.pos, B1.form◦pos, B2.lemma, B2.pos, B2.form◦pos, B3.lemma, B3.pos, B3.form◦pos, Yst.pos, Yst.form◦pos, Yed.pos, Yed.form◦pos, B0.form◦X1.form, B0.form◦X1.pos, B0.pos◦X1.form, B0.pos◦X1.pos, B0.form◦X1.pos◦dictance(B0, X1), B0.pos◦X1.form◦dictance(B0, X1), B0.pos◦X1.pos◦dictance(B0, X1)
Structural Features	B0.leftmost-child.label, B0.lemma◦leftmost-child.label, B0.pos◦leftmost-child.label, X1.leftmost-child.label, X1.lemma◦leftmost-child.label, X1.pos◦leftmost-child.label, X1.lemma◦rightmost-child.label, X1.pos◦rightmost-child.label, X1.rightmost-child.label, set{X1.head◦head.label}

More complex features could be used to earn better parsing result. As a work sample, we use quite limited features in this project (and it do work already). Please refer to (Xun Zhang, et al., 2015) to see well defined (and very complex) feature templates.

Incremental Dependency Parsing with the Perceptron Algorithm

When given the parameters of a structured perceptron, the parsing process contains following steps:

1) Obtain the initial configuration with the given sentence.
2) Extract features from current configuration and calculate scores for each transition according to the parameters of the perceptron. For transitions that earn top *k* scores under current configuration, get new configurations by applying transitions to current configuration. Then, we obtain *k* configurations with top *k* scores. Set accumulated scores for each configuration with the above scores we get. The accumulated score of a current configuration is the sum of scores of each transition in the transition sequence that transform the initial configuration to current configuration.
3) For the kept *k* configurations, calculate scores for each configuration with each transition according to the parameters of the perceptron. Among all those scores, keep top *k* and applying transition on corresponding configurations. Repeat the process until we get *k* finished configurations finally.
4) Select the configuration with highest score. Output the parsing result of the configuration.

Here *k* is the beam width.

To train a model is to estimate the parameters of the perceptron. Similar to above steps, we train the model with training examples. For each example, if the output exactly matches the golden parsing result, we do nothing. But once the output is different from the golden result, we update the parameters of the perceptron.

2. Training and Performance
In the directory

/Checkpoint/sample_unlabelled/

there is an unlabeled model trained on a subset containing 3258 sentences of DeepBank (DM) training data from SemEval 2015 Task 18. The model is obtained by training 20 epochs using a beam width of 32. And it is an unlabeled parsing model. The model achieved an unlabeled F-Score of 77.1% on DeepBank test deta of SemEval 2015 Task 18.

To run the project, please refer to HowToUse.pdf

References:

Nivre, Joakim. 2008. *Algorithms for deterministic incremental dependency parsing*. Computational Linguistics, 34:513–553.

Xun Zhang, Yantao Du, Weiwei Sun, and Xiaojun Wan. 2016. *Transition-based parsing for deep dependency structures.* Computational Linguistics, 42(3):353–389.