An Transition-based Dependency Tree Parser Based on Perceptron

Ran Xian
School of EECS
Peking University
`xianran@pku.edu.cn`

Abstract

In this report, I demonstrate how I build a dependency tree parser with perceptron (Rosenblatt and Frank, 1958) as the classifier. I use the shift reduce method which is quite typycal. In the shift reduce method, features are extracted from the context, or configuration, and a feature vector is built. Then feature vector is then passed to the perceptron to train a classifier. Perceptron is proved to be a very robust classifer which is efficient and convenient, as the features can be defined in a very flexible way.

1 Introduction

Transition-based dependency parsing use the well-known shift-reduce procedure to fulfill the parsing job. During the parsing process, an action is determined according to current condition or configuration. For detail, 3 kinds of action could happen in the shift-reduce process, which is shift, left, and right.

In the standard way, a stack and a queue are used to help organizing the shift-reduce process. The elements in queue and stack are nodes of dependency tree, more specifically, all nodes in the stack and the queue is one dependency tree. If the expected parse results is one single tree, then there will be only one root after some steps of shift, left or right actions.

The configuration mentioned before is the status of the queue and stack, we can extract features from the configuration, and use a classifier to determine what action to be done at given configuration. Here we use a perceptron classifier, mainly because it is easy to be implemented with sparse representation, hence features can be added freely without hacking a lot about the representation. The code of perceptron is adopted from my previous work of emnlpProj1, it's glad to see it works here!

2 The Framework

In this section, I will make a description of the framework, namely the principle of the queue and stack, and the 3 actions which manipulates them.

Initially, the stack has nothing but a root node. The queue has a whole sentence in it. A node is the element of either stack or queue. A node is constructed by words in a sentence, more specifically, the token of then word, the pos tag of the word, the index (position in the sentence) of the word. Note that root node does not corespondant to a word, this is made up by ourselves. The token and position of the root will be a special value (e.g root and 0).

Then an action is determined by the current status of the stack and queue.

- SHIFT action: move the node out from the front of the queue, and push it to the stack.
- LEFT action: make an arc from the top of the stack to the front of the queue. This means that the stack top is the head of the queue front, hence the node of the queue front is combined into the stack top, i.e the node is removed from the queue.
- RIGHT action: make an arc from the front of the queue to the top of the stack. This means that the queue front is the head of the stack top, hence the node of the stack top is combined into the queue front, i.e the node is removed from the stack. Then the left node is pushed to the front of the queue.

There is one important thing to mention that once we want to add an arc, we must make sure that the child node of the arc must be complete, that other nodes who has an arc to it must be correctly added as child of this node previously. This constraint is useful when we are going to determine the action from the training dataset.

Once we have only one node in the queue, the parsing job is over, and a denpendency tree is produced.

3 Training

Now we are going to train the model. The first thing we have to do is determing the correct action to take at each step. This is simple and straightforward. Each time we look at the stack top (we call it top) and queue front (we call it front), if top is a child of front, then the action is LEFT. If the front is the child of top, then the action is RIGHT. Otherwise the action is SHIFT. The process is over and over until there the stack is empty and there's only one node in the queue.

Second we have to extract features corresponding to each of the action. We will notate the top as 0-, front as 0+, the second node in the queue as 1, third as 2, etc. We will notate the token of a node as w, pos tag of a node as p. The naming convention of the feature is like position:name+position:name etc. So the feature for both the pos tag and token of the queue front will be 0-:wp

In my implentation I have following features:

- bias term.
- x:wp, the pos tag and token at position x, x can be 0-, 0+, 1, 2, 3.
- x:w, the token at position x, x can be 0-, 0+, 1, 2, 3.
- x:p, the pos tag at position x.
- 0-:wp+0+:wp, the pos tag & word combination at stack top, combined with pos tag & word combination at queue front.
- 0-:wp+0+:w
- 0-w+0+:wp
- 0-wp+0+:p
- 0-p:0+wp
- 0-w:0+w
- 0-p:0+p
- 0+p:1p:2p, combination of pos tag at position 0+, 1, and 2.
- 0-p:0+p:1p
- 0-p:0-lp:0+p, combination of pos tag at position 0-, the leftmost child of 0-, and 0+.
- 0-p:0-rp:1p, combination of pos tag at position 0-, the rightmost child of 0-, and 1.
- 0-p:0+p:0+lp, combination of pos tag at position 0-, 0+, and the liftmost child of 0+.
- not-yet, if there's more than one node in the stack and queue, this feature is added, indicate the paring job has not yet done.

4 Parsing

Now we have a model to predict action to take at each configuration. The parsing job quite trival, it is almost like the training process, with the exception that some tricks need to be applied. Namely three:

- When there's one node in the stack and more than one node in the queue, the action must be SHIFT. Otherwise the stack will be empty hence the end of the operation, but the queue has more than one node.
- When the stack has more than one node and the queue has one node, we should not have a SHIFT action. Either LEFT or RIGHT should be taken, and are determined by the score output by perceptron, i.e if the score of LEFT is higher, then action is LEFT, vice versa.
- If the stack and queue both has only one node, an LEFT action is taken which leads to then end of the parsing job.

These tricks are important, it influences the final result a lot.

5 Experiment

I use the trn.conll08 dataset to train the parser. The perceptron runs for 10 iterations, it is resonable acceptable that every iteration takes about 10 secs. Then I use the parser trained to parse the dev.conll08 dataset, the final result is:

UP : UR : UF : UCompl = 80.86 : 80.86 : 80.86 : 17.51

LP : LR : LF :LCompl = 80.86 : 80.86 : 80.86 : 17.51

References

Rosenblatt and Frank. 1958. *The perceptron: a probabilistic model for information storage and organization in the brain.*. American Psychological Association.

Zhang, Yue and Nivre, Joakim. 2011. *Transition-based dependency parsing with rich non-local features.*. Association for Computational Linguistics.

Nivre and Joakim 2005. *Dependency grammar and dependency parsing.*. MSI report.