Project 2: Shift-Reduce Parsing

Victoria Anugrah Lestari
Department of Computer Science
The University of Texas at Austin
`vlestari@cs.utexas.edu`

Abstract

(I collaborrated with Marko Vasic and discussed with Chin Wei Yeap and Ravindra Pai).
Shift-reduce parsing is a fast way to train a dependency parser. This assignment instructed us to train a dependency parser using two methods: greedy training and global training with beam search. I implemented my greedy model with logistic regression and achieved 79.39 with basic features on the development set. With additional features and Adagrad optimizer, I succedded to achieve 92.27 UAS on the development set. In addition, my beam model was implemented using perceptron learning and achieved 75.84 UAS with beam size 8, 5 epochs, and the Adagrad optimizer. However, additional features decreased the performance of the beam model.

1 Introduction

Parsing, which is a significant task in linguistics to determine the grammatical structure of a sentence, consists of two types: constituent-based parsing and dependency parsing. While constituent-based parsing follows a set of grammar rules based on part-of-speech (POS) tags, dependency parsing follows the relations of words (or lemmas) in a sentence, usually choosing the main verb as its head word.

There are two types of dependency parser: a graph-based parser and a transition-based parser. In this assignment, we are asked to build a transition-based parser using the method of shift-reduce parsing.

2 Data

We were provided with dependency tree datasets in the CONLL format. The datasets were split into three parts: training set, development set, and test set. For the English test sets, we were given both the blind test set and the regular test set (with the actual labels). For further experiments, we were also provided with Arabic, Chinese, and Japanese datasets.

3 Methods and Experiments

Training a model for shift-reduce parsing is considered a multiclass classification where the classes are from the set: {`shift (S)`, `left-arc (L)`, `right-arc (R)`}. Based on the definition by (Andor et al., 2016), given the input sentence x, the terms are described as follows:

- A state s is defined as the condition of stack, buffer, and a list of dependencies at a given time step.
- A set of states $S(x)$ contains all the possible states that can be generated from x.
- A special start state $s_0 \in S(x)$ is a condition where the stack only contains the root, the buffer is full with all the words in x, and the list of dependencies is empty.
- A set of possible actions $A(s,x)$ for all $s \in S(x)$.
- A transition function $t(s,d,x)$ that returns a new state s' for any action/decision $d \in A(s,x)$.

The term *action* is used interchangeably with the term *decision*. An action can only be either `S`, `L`, or `R`. Given a certain state, some actions are not possible. I listed the following conditions for the possible actions:

- If the stack only contains one element and the buffer contains more than one elements, the possible action is shift.
- If the buffer is empty and the stack contains two elements, which are the root and another element, the possible action is right-arc.
- If the buffer is empty but the stack contains more than two elements, the possible action is left-arc or right-arc.
- If the root already has a child and the root is one of the top two elements in the stack, the possible action is shift. (Root cannot have more than one child.)

The results of each experiment are briefly described in each subsection, but the overall comparison of all experiments is shown in Table 2.

3.1 Greedy Training

The greedy training method looks at the local best decision only as guided by the gold standard (also called oracle). The model never sees other states besides those in the set of gold states; it calculates the probability of each decision to determine the log likelihood, but the next state is always generated by the gold decision.

I used logistic regression for the greedy model. Inference is decided by $\text{argmax}_{y \in Y} P(y|x)$, with $Y = A(s, x)$. $P(y|x)$ is calculated by the softmax function:

$$P(y|x) = \frac{\exp(w^T f(x, y))}{\sum_{y' \in Y} \exp(w^T f(x, y'))}$$

Computing the gradient for training the model is as follows:

$$\frac{\partial}{\partial w_i} \mathcal{L}(x_j, y_j^*) = f(x_j, y_j^*) - \sum_{y \in Y} f_i(x_j, y) P(y|x_j)$$

The brief description of my greedy implementation is as follows: I extracted the features of the oracle and stored them in a feature cache. Then I started from the initial state and calculated the log probability for each possible action (illegal actions are given 0 probability), storing the log probability values in a list of length 3 (for the three actions). However, the next state is still generated by the gold decision at that particular point. Then I computed the gradient in every state and adjusted the weights accordingly.

I experimented with stochastic gradient descent (SGD) and Adagrad to train the weights. The learning rate for SGD is $\frac{0.1}{k}, 1 \leq k \leq$ number of epochs, while for Adagrad I used $\lambda = 1 - e7$.

First, I trained the greedy model on 1000 sentences and tested on the whole development set with 5 epochs. The greedy model with SGD yielded 73.30 UAS, while Adagrad yielded 71.95 UAS. In my program, Adagrad was considerably slower than SGD (probably because I did not give it proper setting).

Trained on the whole training set, the greedy model with SGD yielded 79.39 UAS while tested on the whole development set. I have not experimented with the greedy model trained on the whole training set with Adagrad.

3.2 Global Training

In contrast to the greedy method, the global method looks at all the potential new states that can be generated from a particular state. Since generating all states will be exponentially large, the global model uses beam search to only look at the top k states (k is the beam size).

I applied perceptron learning for the global method. I followed chapter 8.2 of (Goldberg, 2015) to understand about the scoring function for this method and the slide from lecture 10, page 24, using early updating. The brief description of my implementation is as follows: I started with the initial state and prepared a list of beams with size $2n + 1$, n denoting the length of the sentence (for indexing reasons, I need to add 1). I added the initial state and features into the first beam in the list, giving score 0. I inserted the features (using Counter) into the beam to keep track of the active features per state. The score of the next states are calculated:

$$\textbf{new\_score} = w^T f(x, y) + \textbf{old\_score}$$

Then I added the old features with the new features counter and put the new features, new state, and new score into the beam.

I experimented with both adding new features from unseen states during the training; however, setting the `add_to_indexer` variable to be False gave me better performance during training with

Table 1: Experiments for global training, b is beam size and e is the number of epochs.

Sentences	Configuration	UAS
1000	b = 10, e = 10, SGD	66.70
	b = 10, e = 5, ADA	70.76
All	b = 5, e = 5, SGD	67.60
	b = 7, e = 5, SGD	68.07
	b = 10, e = 10, SGD	69.05
	b = 8, e = 5, ADA	75.84

small datasets. Thus, my model only saw features that were extracted from the gold states.

Since training on the whole training set was too slow, I experimented with 1000 sentences at first, varying the beam size and number of epochs. Numerous experiments with SGD did not yield UAS 75; however, when I used Adagrad, the performance increased to above 70 for 1000 training sentences, beam size 10, and 5 epochs. I achieved above 75 UAS (75.84) on the whole training sentences with beam size 8 and 5 epochs.

Table 1 shows the results of my experiment. I concluded that with the same beam size, Adagrad optimizer worked better than adding more epochs.

4 Extensions

(Zhang and Nivre, 2011) served as my reference to add new features to the `extract_features` method. They mentioned adding several feature sets to improve the UAS; however, not all of those features increased the UAS of my greedy model. I noticed that adding third-order features did not improve the performance of my model. I found that adding too many word features also decreased my model's performance. I argued that it overfitted the training set so that it did not perform well on the development set, which contained many new words.

I only added single word features, word pair features, and dependency labels to my model. The dependency label features were the most significant in increasing the performance of my model. I believe this is caused by the importance of dependency labels in deciding to choose actions (since we are building a dependency parser). I added and removed features one by one, testing on small datasets at first (100 and 1000 sentences), but I did not record the results separately.

This is my list of features I added to my model: S_0wp, B_0wp, S_0d, S_0dp, B_0d, B_0dp, S_0dB_0d, S_0dS_1d, $S_1dS_0dB_0d$, S_0wB_0w, S_0wS_1w, $S_1wS_0wB_0w$ (S_0 means top of stack, S_1 means the second from top of stack, B_0 means top of buffer, w means word, p means POS tag, d means dependency label).

Table 2: Performance of the greedy model. 'Sentence' means the number of training sentences the model was trained on. All configurations were run on 5 epochs. For SGD, the learning rate is 1/epoch number

Sentence	Configuration	UAS
1000	SGD, no features	73.30
	ADA, no features	71.95
All	SGD, no features	79.39
	ADA, no features	79.01
1000	SGD, with features	87.78
	ADA, with features	88.14
All	SGD, with features	92.01
	ADA, with features	92.27

It increased my greedy model performance with 1000 sentences (with SGD, 5 epochs) from 73.30 to 87.78 UAS. With all training sentences (SGD, 5 epochs), the UAS increased from 79.39 to 92.01. Using Adagrad increased the UAS a little to 92.27.

In contrast, the additional features did not do well with the beam model. Training on 1000 sentences, beam size 5, and 10 epochs with SGD, it gave me 52.15 UAS. Meanwhile, beam size 10 and 5 epochs (also with SGD) gave me 54.69. Without features, I got scores close to 70 UAS. Thus, I did not proceed with the whole training set because it would take too much time. I believe that the beam model performed worse with additional features because there are too many features too learn. Probably, with larger beam size and more epochs, the beam model with additional features might perform as well as the basic beam model.

Table 2 shows the summary of my experiment: both greedy and beam model, SGD and Adagrad, with or without additional features. In conclusion, the best model is greedy with Adagrad and additional features (92.27 UAS).

References

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the*

Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics, Berlin, Germany, pages 2442–2452. http://www.aclweb.org/anthology/P16-1231.

Yoav Goldberg. 2015. A primer on neural network models for natural language processing. *CoRR* abs/1510.00726. http://arxiv.org/abs/1510.00726.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA - Short Papers*. pages 188–193. http://www.aclweb.org/anthology/P11-2033.