CS395T: Shift-Reduce Parsing Project Report

Zeyuan Hu
Computer Science Department
University of Texas at Austin
Austin, Texas
iamzeyuanhu@utexas.edu

Abstract

In this project, I implement a purely greedy parser using a logistic regression classifier to generate shift-reduce decisions. Then, I implement a global beam-search parser that searches over the same space but perform the inference over the whole sentence. Lastly, I tried to implement the arc eager transition system and I will talk about some questions raided during the arc-eager system implementation.

1 Collaborators

Zhan Shi, Danlu Wang

2 Introduction

Dependency parsing is a task about assigning syntactic annotations for a given sentence. Specifically, the task is to predict the syntactic parents and dependency labels and thus form a dependency tree. For example, in a sentence *education ads:*, the task will predict the parent of word *education* is *ads*. Systems for dependency parsing either operate in a graph-based or trainsition-based paradigm. In this project, we explore the transition-based systems with the greedy and the global beam-search parsers, which make local decisions and process the dependency tree incrementally from left-to-right. There are four components in both systems: input buffer, stack, oracle, and dependency relations. The parser's *state* is captured by a *stack* of partial dependency trees and a *buffer* of words waiting to be processed. During the parsing, we consult *oracle* to get the decisions on how to process the dependency trees and words. Then, we apply decisions and get *dependency relations*. In this project, we construct our greedy parser oracle using a logisitic regression classifier and our global beam-search parser oracle using Structured SVM (SSVM).

3 Implementation Details

3.1 Greedy Parser

For the greedy parser, we first build a `feature_cache` by walking through the gold states of each sentence and then apply all three possible actions to those states. The data structure for the `feature_cache` is a list of `feature_matrix` with dimension 3 by 23, with 3 representing all three possible actions and 23 representing the number of features. Then, for each epoch, I calculate the $P(y|x)$ for all possible y, which, in this case, are all three possible actions. Once we calculate $P(y|x)$, we can get the gradient $f_i(x_j, y_j^*) - \sum_y f_i(x_j, y) P(y|x_j)$ with $f_i(x_j, y_j^*)$ representing the indicator function of a gold feature value. This translates into the program by incrementing the weights associated with the gold features by one. The gradients are stored in a 3 by 23 matrix with each row corresponding to the gradients of the weights associated with each possible action. Once we have the gradients, we can perform the stochastic gradient descent (SGD) update to the weights. After the training phase, we build the oracle by caclulating $\text{argmax}_y P(y|x)$.

3.2 Global Beam-Search Parser

The global beam-search parser has the same code structure as the greedy parser. We first train the model to get the weights and then we construct the parser using those weights. One assumption made by the greedy parser is that the oracle will give the correct operator at each step of the parse, which is unlikely to be true (Jurafsky and H.Martin, 2017). In order to handle this issue, we need to take a look at alternative choices at each step, which

leads to the beam search of all the possible parse sequences. Like the greedy parser, we first implement a `feature_cache` to facilitate our access to the features and help us to initialize our `feature_weights`. In each epoch, for each sentence, we initialize our beam with action "S" because of the specification of arc-standard transition system. Then, for all the following beams, we need to check whether we are already in the final state. If not, we apply all three actions to the states sit in the beam and push the newly-created states onto the current beam. Afterwards, we use the early updating strategy by performing the SSVM update if we find the gold actions already fall out of the beam. If that is not the case, we perform the SSVM update using the features that are cumulative over the whole sentence.

3.3 Arc Eager Transition System

So far, we have assumed that the transitions between states follow the arc standard system. As the extension to the project, I want to explore the alternative transition system: arc eager. Unfortunately, I meet some issues during the implementations. The major one comes from the inconsistent definition of the precondtion for each action under in the system. We can apply `LeftArc` action as long as the top of the stack is not the root node. For `Shift` action, we can apply it as long as there are tokens inside the buffer. There is no restriction for the `RightArc`. However, for `Reduce` operation, there are discrepancies from different literature. Jurafsky (2017) does not explicitly talk about this issue but from Figure 14.10 example, it seems like `Reduce` can only happen when there is no tokens in the buffer and there are elements left on the stack other than root. This conjecture is confirmed by Goldberg and Nivre (2012), which in the paper, they state that "The REDUCE transition pops the stack and is subject to the preconditions that the top token has a head". However, some literature thinks that the precondition should also include: the top of the stack has all its children attached to it (Ballesteros, 2015). During the implementation, I find that there might be some exceptions to the rules. For example, the sentence *Influential members of the House Ways and Means Committee introduced legislation that would restrict how the new savings-and-loan bailout agency can raise capital, creating another potential obstacle to the government's sale of sick thrifts.* has a state: `[-1, 1, 2, 5, 9, 10, 13, 21, 24, 27]` for the stack, `[36]` for the buffer, and `{0: 1, 2: 1, 4: 5, 5: 2, 6: 5, 7: 8, 8: 5, 10: 9, 12: 13, 13: 10, 18: 19, 19: 21, 20: 21, 21: 13, 22: 21, 26: 27, 27: 24, 28: 27, 29: 30, 30: 32, 31: 30, 32: 28, 33: 32, 34: 35, 35: 33}` for the dependency relations. Since there is no parent-child relation between `27` and `36` and we cannot apply `Reduce` as well, the only option left is `Shift`. However, if we check carefully for the elements on the stack, we find that `36`'s parent is `24`, `24`'s parent is `13`, `13`'s parent is `10`, `10`'s parent is `9`, and `9`'s parent is `-1`, which are all on the stack. Since there is no operation allowing us to examine two elements on the stack in the arc eager system, the only choice we left is `Reduce`, which will lead to the failure of parsing. The detailed code implementation and reproduction can be found under the directory `hw2-arc-eager`.

4 Experiments

Figure 1 shows the result of the UAS score for both the greedy parser and the the global beam-search parser on the development set. Table 1 shows the parameters experimentation with different optimizer, learning rate, epoches, sentences and batch size for different parsers. In general, as the number of epoches in the training phase increases, the UAS scores increase. The number of training examples (i.e., sentences) play a major role in the parsing result. The target UAS scores (i.e., 78 for greedy parser and 75 for global beam-search parser) can only be achieved by using the whole training examples (i.e., 39832 sentences).

5 Conclusion and Future Work

In this project, I implement the greedy parser and the global beam-search parser. I show the importance of the training data to the system performance. I also tried to implement arc eager transition system as extension but there are some questions raised during the implementation. In the future, I can finish the implementation of arc eager transition system and perform feature engineering to improve the performance of both parsers.

Table 1: Impact of the parameters on the UAS score

Parser	Optimizer	Learning Rate	Epoches	Sentences	Batch Size	UAS
Greedy	SGD	0.2	10	5000	1	72.27
Greedy	SGD	0.2	20	5000	1	74.67
Greedy	SGD	0.2	30	5000	1	75.81
Greedy	SGD	0.2	40	5000	1	75.74
Greedy	SGD	0.2	30	6000	1	76.35
Greedy	SGD	0.2	30	7000	1	76.43
Greedy	SGD	0.2	30	39832	1	**78.60**
Global	Adagrad(0.0001, 5)	5	10	39832	20	**75.67**

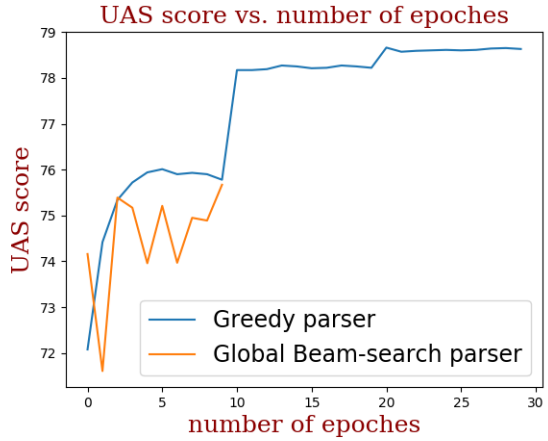

Figure 1: UAS score vs. number of epoches in development set

References

Miguel Ballesteros. 2015. *Transition-based Dependency Parsing*.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *COLING 2012, 24th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, 8-15 December 2012, Mumbai, India*. pages 959–976. http://aclweb.org/anthology/C/C12/C12-1059.pdf.

Dan Jurafsky and James H.Martin. 2017. Speech and language processing. Preprint on webpage at https://web.stanford.edu/~jurafsky/slp3/.