A Dependency Parser for Swedish

Jonas Pålsson
D98,
Lund Institute of Technology, Sweden
d98jpa@student.lth.se

Marcus Stamborg
D03,
Lund Institute of Technology, Sweden
d03mst@student.lth.se

Abstract

This paper describes an implementation of a dependency parser for swedish using Joakim Nivre's algorithm and which features to include in a feature set for a machine learning algorithm to get the best possible accuracy for the parser.

1 Introduction

This is our final project in the course Language Processing and Computational Linguistics[1] and describes the work done to create a dependency parser for swedish using the Java programming language. The goal is to implement a working parser but also to find a good feature set for the machine learning algorithm which is used.

Dependency Parsing is the act of parsing a sentence in order to find the relations between the words. Different algorithms exists for dependency parsing and we have chosen to implement Joakim Nivre's algorithm since it has reported the best results for the swedish language. Instead of using predefined rules we use the statistical classifier Weka[2] to decide parse actions. In order to find a good feature set for the machine learning algorithm we ran simulations using several different combinations of features which are all described in the paper.

As test and train data we are using "Talbanken05" which is a swedish corpus manually tagged with part of speech, a Golden Standard corpus. This specific version of "Talbanken05" is the same used in CoNNL-X Shared Task: Multi-lingual Dependency parsing (CoNLL-X, 2006). The train set contains approximately 11500 sentences and the test set contains 300 sentences. In order to use Nivre's algorithm the corpus must be projective so before the corpus is used it is projectivized using nonproj2proj(nonProj2Proj, 2005). Note that no support for projectivization or de-projectivization is built into the program and this has to be done manually if the corpus is non-projective.

[1] http://www.cs.lth.se/eda171

[2] Weka is a collection of machine learning algorithms used for machine learning. http://www.cs.waikato.ac.nz/ml/weka

2 Nivre's Algorithm

Nivre's algorithm extends the basic shift-reduce algorithm by adding two parsing actions, "Left Arc" and "Right Arc". The algorithm uses an input queue initialized with the current sentence and a stack for temporary storing words to be processed. The following actions are used by the parser.

- Shift - shifts the first token on the input to the top of the stack. Only performed if there are more tokens on input and no other action is currently possible.
- Reduce - removes the top token of the stack. Only performed if the token has a head.
- Left Arc - creates an arc from the first token on input to the first token on the stack and performs a reduce. Only performed if the token on top of the stack does not yet have a head.
- Right Arc - creates an arc from the first token on the stack to the first token on input and performs a shift. Only performed if the first token on input does not yet have a head.

The dependency graph produced by the algorithm is both projective and acyclic. For more information about the algorithm see (Nivre, 2003).

3 The Implementation

The main idea behind the implementation is to train a Weka classifier using a Gold Standard corpus. All machine learning algorithms need a feature set which is the facts the classifier needs to make a decision. A feature can be for example the part of speech or lexical value of tokens on the input queue or stack.

The trained classifier is then used to decide which action to use in Nivre's algorithm. Our implementation is divided into three steps.

3.1 Collect Data

Since we have a corpus which is hand annotated (Gold Standard) it is possible to determine the action sequence used to produce this corpus. This technique is called Gold Standard Parsing (Nugues, 2007) and the rules used are the following.

- If the first token on input has the token on top of the stack as head, do Right Arc.
- else if the token on top of the stack has the first token on input as head, do Left Arc.
- else if there exists an arc between the first token on input and any token in stack, do Reduce.
- else, do Shift.

The feature set for each step of the Gold Standard Parsing is put in an arff-file which is a special format that Weka can read. Entries in this file describes the current state of the algorithm and which action to choose in that state.

3.2 Train Classifier

Weka has a lot of different machine learning algorithms and we chose to use the well known C4.5. In weka this algorithm is called J48 and it was chosen since it has a good time/performance ratio, ie execution time is low enough for continuous testing. A better algorithm could be used but the longer execution time made that impossible.

The result of this step is a model (trained classifier) that can be used in the main program.

3.3 Parse

In this step the previously trained classifier is used to determine the head for each word in a sentence. This is done with Nivre's algorithm and the classifier decides which action to use in each step. To be able to compare different feature sets, and to be able to measure the results, a "correctness"-score is calculated by dividing the number of words that were assigned a correct head with the total number of words.

4 Feature Sets

In order to get a good result from the parser there are two main components that can be changed, namely the classifying algorithm and the feature set. Since it is already decided that the C4.5 algorithm is to be used the only way we can affect the result is by choosing different feature sets. Without a solid background in computational linguistict we found it difficult to determine which features to use. Therefore we chose to implement a program that tested several combinations of the features we chose to use. The feature values used are Part Of Speech (POS) and the lexical value of a word (LEX). These values can be chosen from words on the input queue and the stack. The notation 1.input refers to the first token on the input and 1.stack to the top token of the stack. The special case 0.input refers to the word immediately before 1.input in the original sentence. As a last measure to try to increase performance the POS of the leftmost and rightmost children (LMC and RMC) of 1.stack and the POS of the leftmost child of 1.input are also included. The children are derived from the dependency graph.

The feature set included in the simulations are combinations of:

- 1 to 6 words from input, POS
- 1 to 6 words from stack, POS
- 1.input LEX and 1.stack LEX
- 1.input LMC and 1.stack LMC and RMC
- 0.input POS

Every feature set also contains constraints to model valid actions in the current state of the parser.

The results of the tests will be discussed in chapter 5, Results and Discussion.

5 Results and Discussion

We have chosen not to include results where lexical values were used since this always resulted in a lower score. This is the case both when using the lexical value of all used words in input and stack

and when only using the lexical value of 1.input and 1.stack. We belive this might be a limitation in the classification algorithm used, C4.5, since using lexical values possibly generates too many states for each feature. Better results might be achieved using a different algorithm such as State Vector Machines. The complete results for the following tests can be found in Appendix A.

5.1 Input and Stack

In this test every combination where the window size of stack and input varies between 1 and 6 was evaluated. The best result was obtained when using 6 tokens from the stack and 4 tokens from the input which resulted in a ”correctness”-score of 0.7986. This is a very basic feature set that obviously can be improved.

5.2 Input, Stack and Children

When using children we obtained the best results using 2 tokens from the stack and 5 from the input which resulted in a ”correctness”-score of 0.8136. Looking at the results one can see that overall results are improved by approximately 1-2% and that a large input window now is much more important than a large stack window.

5.3 Input, Stack and Previous Input

First of all, this is the single best additional feature we have added to the basic feature set. The score increased to 0.8186 and overall almost all combinations of window sizes gave a good result when using this feature. The importance of stack and input is about the same in this case and the score table is very balanced with few deviations.

5.4 Input, Stack, Children and Previous Input

Last we tried all of the features together resulting in a ”correctness”-score of 0.8142. Once again a large input and small stack is the best combination when using children. Using children by itself improved the score a bit, using previous input by itself increased the score significantly but using these two features together didn't improve the score even more, instead the score landed in between the two previous ones.

6 Future Improvements

There is a lot that can be improved given enough time to implement the changes and most of all time to do more simulations and classifying.

First of all a better machine learning algorithm can be used, such as SVM since it has proven to be significantly more effective than C4.5 but with the drawback that it takes a very long time to execute.

Another area of improvement is to try other features in the feature sets. For example one could use siblings in the graph, try to increase the window size and to try more combinations from the original sentence. All of this will require a considerable amount of computing power so access to a scientific computer grid network would be a big advantage.

References

CoNLL-X Shared Task: Multi-lingual Dependency Parsing. 2006. http://nextens.uvt.nl/˜conll

nonProj2Proj. 2005. http://w3.msi.vxu.se/˜nivre/research/proj/0.2/doc/Proj.html

Joakim Nivre. 2003. *An efficient algorithm for projective dependency parsing*. Växjö University, Växjö, Sweden.

Pierre Nugues. 2007. *Language Processing and Computational Linguistics - Lecture 11 - Parsing Techniques*. Lund Institute of Technology, Lund, Sweden.

A Results

Scores using features:
Stack_n_POS, Input_n_POS

	Input 1	2	3	4	5	6
Stack 1	0.6936	0.7765	0.7804	0.7801	0.7779	0.7806
2	0.7297	0.7937	0.7970	0.7961	0.7958	0.7946
3	0.7300	0.7933	0.7963	0.7958	0.7940	0.7944
4	0.7309	0.7940	0.7967	0.7972	0.7960	0.7953
5	0.7327	0.7944	0.7974	0.7984	0.7969	0.7960
6	0.7313	0.7940	0.7972	**0.7986**	0.7965	0.7960

Scores using features:
Stack_n_POS, Input_n_POS, Children

	Input 1	2	3	4	5	6
Stack 1	0.7161	0.8007	0.7972	0.7967	0.8036	0.8064
2	0.7268	0.8078	0.8055	0.8094	**0.8136**	0.8129
3	0.7275	0.8066	0.8076	0.8098	0.8129	0.8131
4	0.7300	0.8057	0.8076	0.8094	0.8096	0.8091
5	0.7309	0.8073	0.8071	0.8096	0.8101	0.8097
6	0.7307	0.8064	0.8071	0.8089	0.8092	0.8094

Scores using features:
Stack_n_POS, Input_n_POS, Previous_Input_POS

	Input 1	2	3	4	5	6
Stack 1	0.7242	0.8022	0.8055	0.8052	0.8046	0.8050
2	0.7558	0.8156	0.8168	0.8179	0.8174	0.8182
3	0.7580	0.8152	**0.8186**	0.8184	0.8174	0.8184
4	0.7581	0.8158	0.8177	0.8184	0.8172	0.8175
5	0.7594	0.8167	0.8182	**0.8186**	0.8174	0.8177
6	0.7574	0.8161	0.8181	0.8177	0.8165	0.8172

Scores using features:
Stack_n_POS, Input_n_POS, Previous_Input_POS, Children

	Input 1	2	3	4	5	6
Stack 1	0.7210	0.7999	0.8004	0.8002	0.8062	0.8076
2	0.7279	0.8064	0.8068	0.8108	0.8110	**0.8142**
3	0.7283	0.8068	0.8068	0.8101	0.8136	0.8138
4	0.7307	0.8068	0.8089	0.8106	0.8108	0.8105
5	0.7316	0.8068	0.8075	0.8103	0.8114	0.8114
6	0.7344	0.8064	0..8076	0.8101	0.8106	0.8108