Dependency Parsing

Johan Aulin
D03
Department of Computer Science
Lund University, Sweden
d03jau@student.lth.se

Carl-Ola Boketoft
ID03
Department of Computing Science
Umeå University, Sweden
calle_boketoft@hotmail.com

Abstract

The aim of this project was to extend an implementation of Joakim Nivre's dependency parser for the Swedish language. Dependency parsing consists of finding links between words in a setence. This is done through building a structure of dependency relations among the words by assigning them with values of heads and dependents. In order to increase the accuracy of the parser, a number of features were added to the algorithm and the use of a different classifier was explored. There was an evident increase in accuracy for the parsing algorithm after the new features were implemented. The use of another classifier was not successful and was not further investigated.

1 Introduction

This is the final project for the course in Language Processing and Computational Linguistics, EDA171 (Nugues, 2008b). The objective for the project was to define a study topic and an application in either the field of language processing or computational linguistics.

The topic chosen for this project was inspired by the CoNLL-X (Tenth Conference on Natural Language Learning) Shared Task: Multi-lingual Dependency parsing (CoNLL-X, 2006), 2006, where the task was to implement an algorithm for a dependency parser. Dependency parsing can shortly be described as a technique of finding relations between words in a sentence. Many different algorithms were developed for the conference and Joakim Nivre's dependency parser for the Swedish language was the algorithm with the highest accuracy rate for labeled parsing (CoNLL-X, 2006).

It was decided to work with an implementation of Nivre's algorithm for this project due to its reported success in the conference. Since one of the programming assignments for the course EDA171 was to improve a Java implementation of Nivre's parser (Nugues, 2008a; Nugues, 2008c), it was convenient to use this as a foundation for the implementation of the project. The implemetation was using a statistical classifier in the data mining tool Weka to decide parse actions (WEKA, 2009).

The aim of the project was to improve the available implementation of Nivre's dependency parsing algorithm by adding new features and using different classifiers.

The test and training sets used as gold standard in the project were the manually annotated Swedish corpus *Talbanken05* (Nivre, 2005). Talbanken05 was also used in the CoNLL-X conference and it contains all the tags that were used in the features for the parsing algorithm for the Swedish language. The training set consists of 6316 sentences of informative prose.

The first step of the algorithm is a so called gold standard parsing, which is explained in further detail later in the report, that uses Nivre's algorithm to parse the hand annotated corpus. The result from the procedure is stored in an output file with a special format to be used with Weka. The next step is to import the output file in Weka to create a decision tree classifier. The decision tree classifier is then used by Nivre's algorithm in the process of parsing a test file with a number of sentences. The parsing results in a file where the words in the sentences have been assigned tags that identify them as heads and dependents, according to the method of dependency parsing. In order to evaluate the results, a perl script was used to calculate a ratio of accuracy for the implemented parsing algorithm.

The two aspects that were investigated in order to improve the accuracy of the algorithm were the implementation of new features for the parser and the use of different classifiers in Weka.

The most successful results were acquired by implementing new features. The features that were implemented included the relation between words with different grammatical values, dependency relations between words and the frequency of words with different lexical value. Throughout the project, a classifier called J48 was used when producing the decision trees in Weka. Another algorithm called Naïve Bayes was used to produce another decision tree model but the results from using the model were much lower than the results acquired when using the J48 model, so no more experiments with that classifier were conducted.

2 Corpora

When training a parser, some sort of correct data must be used. Such a collection of data is called a corpus, and it contains *gold standard* annotated sentences. This means that someone (or indeed a large group of people) has manually annotated a text.

A corpus has to be large and diverse, in order to be representative of a language. This means that corpora normally contain tens of thousands of sentences. Parsing them manually is obviously a huge piece of work, and therefore many corpora are reused for a long time, perhaps with additions but essentially the same.

CoNLL-X defined a corpus format to use in the shared task. A CoNLL-X corpus is a text file encoded in UTF-8, with ten columns representing the different features of tokens. A token can be either a word or a punctuation sign. Each sentence ends with a blank line, and each word within a sentence is numbered, starting from one.

The following list describes all the different elements that are contained in CoNLL-X formatted corpora.

ID This is merely a token counter, starting at one for each new sentence.

Form If the token is a punctuation mark, the mark itself is displayed, otherwise the form of the token is shown. For a word, this means the actual word as it is spelled out.

Lemma The lemma of a word is a basic form of a word. How to select this basic form varies between languages. It can also vary from one corpus to another, depending on the purpose, structure and what is most convenient for that particular corpus.

One *lexeme* contains many forms, and one of these is chosen as the lemma. For example, the lexeme containing *run*, *runs*, *ran* and *running* has the lemma *run*. This enables, among other things, the counting of different forms with the same lemma as the same word, when doing frequency analysis.

Coarse part-of-speech (CPOS) The coarse part of speech is used for languages that have hierarchical grammars. The CPOS tag then represents the main category, while the POS tag represents the precise part of speech.

Fine part-of-speech (POS) The lexicon can be divided into parts of speech (POS), that are classes in which the words share similar grammatical properties (Nugues, 2006). A part of speech can also be called a *word class*.

Features (feats) A word class can have an arbitrary amount of features, depending on language. In English, the word class *pronoun* can be either first, second or third person, and it can be singular or plural. The word "we" would then be a pronoun with the features "person: 1" and "number: singular". Some languages don't recognize grammatical features at all.

Head (head) This determines which other token in the sentence is the head of the current token.

This may for instance be an adjective describing a noun, which would then have the noun as its head.

Dependency relation (deprel) This is the type of relation a dependent has to its head. For the relation between an auxiliary verb and its main verb, this could for example be "AUX".

Projective head (phead) If a projective graph is required, the projective head can be used for those arcs that are non-projective. This represents the closest permissible ancestor of the real head, such that the graph remains projective (Nivre and Kübler, 2006).

Projective dependency relation (pdeprel) This is the same as dependency relation, but covers the relation between a dependent and its projective head.

3 Dependency Parser

A *dependency parser* is a parser that uses the internal dependencies of a sentence to parse it. Every word in a sentence, except the root word, has a *governor*, also called *head*. Words that have heads are called *dependents*.

The figure below shows an example sentence, that consists of six words, one punctuation sign and one meta word called *ROOT*. The arrows on the arcs point from the dependent to the head. The structure of arcs shows that for example "dagens" is a dependent of "samhälle". The only word without a head, is the root word, "Är". Though in the figure, "Är" is marked as a dependent of ROOT. This ROOT is an easy way to show where the sentence starts, and to make a parser easier to implement.

A labeled dependency parser, would have labels on every arc, to further describe the dependency between dependent and head. The parser implemented in this project is unlabeled, and so does not have any labeled arcs.

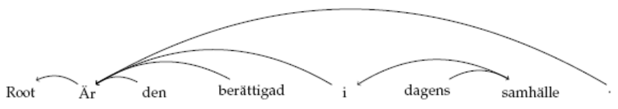

Figure 1: Example sentence with unlabeled dependency arcs

4 Nivre's Algorithm

Nivre uses a dependency parsing algorithm that starts with *gold standard parsing*. After this, a classifier should be trained so that the algorithm has a model to use for its final step, the parsing. Each of these steps will be explained below. First though, the shift-reduce algorithm extended with left-arc and right-arc will be presented. This algorithm is used throughout the whole process.

4.1 Shift/reduce/left-arc/right-arc algorithm

The input file is stored token for token in a queue. A stack is used to store partially processed tokens. Each token is then processed and an action is taken based on certain facts. The different rules and their meaning is described below:

Shift Pushes the first token of the queue onto the top of the stack.

Reduce Pops the stack.

Right-arc Adds an arc from the token on top of the stack to the next token in the queue, and pushes the latter onto the top of the stack.

Left-arc Adds an arc from the next token in the queue to the token on top of the stack, and pops the stack.

4.2 Gold Standard Parsing

Gold standard parsing means reading a hand annotated corpus, and determining what series of actions in the set {shift, reduce, left-arc, right-arc} would obtain the same result. The process for selecting an action can be described in pseudo code as follows:

- If the first word in the queue has the top word on the stack as its head, then do right-arc.
- Else, if the top word of the stack has the first word of the queue as its head, then do left-arc.
- Else, if the first word in the queue has a head anywhere in the dependency graph, then do reduce.
- Else, do shift.

Then the feature set for each word is saved in an output file that can be read by the classifier software used. In this project, a so called *arff* file is created. Entries in this file describe the current state of the algorithm and which action to choose in that state. The arff file also contains a header, describing all the different features, and each of their possible values.

4.3 Training the Classifier

In order to train a classifier, the code can be built into the parser software, or it can be done using a standalone application. This project uses a standalone application for machine learning, called *Weka*. Weka reads a so called arff file, with a formatting specific for Weka, and extracts data from it. When this is done, a classifier algorithm must be selected. For example, in this project, both the algorithm *J48* and *Naïve Bayes* were used.

Weka then creates a model, which contains the classifier that will be used when parsing. For J48, this will be a decision tree. For other algorithms, this could be something completely different, such as a rules engine, a function or a neural net. The model is then used in the parsing step to predict the next parsing action from the current features.

4.4 Parsing

In the parsing step, the parser is finally put to use. The input data for this step should be different from the training data. It is possible to overfit a parser to a certain vocabulary, if the corpus lacks diversity. Parsing a completely different input file, and evaluating the result can give a hint on whether or not this is the case.

In order to parse a file, Nivre's algorithm uses the trained model to make decisions about which action to take in each parser state. In using different actions as dictated by the model, all the words in the input queue receive a head. The result is a dependency graph that can be output to a file, for example in CoNLL format.

The advantage of using the CoNLL format, is that the output can then be compared to a manually annotated corpus. The CoNLL website(CoNLL-X, 2006) contains an evaluation script that gives a score based on the gold standard file being correct. Every deviation from the gold standard file in the output file will deduct from the score.

4.5 Evaluation

Run an evaluation program with the CoNLL test file and the result file from the parse. The rate of accuracy of the parser is then calculated by dividing the number of words with a correct assigned head with the total number of words in the test file.

4.6 Classifier

The implementation of Nivre's algorithm that was provided as a basis for the project was designed to be used with an algorithm called J48. This J48 is an implementation of the decision tree learner C4.5 by Ross Quinlan. (Quinlan,) A decision tree learner analyzes a source statistically, and builds a tree that can later be used to quickly make decisions based on those statistics.

A decision tree learner is only one type of classifier. There are other types, and often a vast amount of different classifiers belonging to the same type. One other type, which was also tried in this project is *Naïve Bayes*. What is different about this classifier, is that looks at all features independently, and adds their results up afterwards.

Other examples of classifiers are *Support Vector Machines*, *Neural Networks* and *Linear classifiers*.

4.7 Implemented Features

The improvement of the algorithm was mainly done by adding features to the initial version of the parser from the lab. A number of features were already in use in the base implementation so there was a relatively high accuracy rate from the parse using only the already available features, as seen in the result section. Comparing those results with state of the art results from the CoNLL-X conference, it was evident that there was much room for improvement to reach higher accuracy scores. A number of features were added to the implementation and assesed to see how the success rate of the algorithm was affected. The selection for these was simply based on common sense and guessing what would be most successful in improving the parsing result. A list of the features that were used in the implementation of the algorithm is shown below.

Features numbered 1 through 7 are the initial features that were available in the implementation for the lab. Features 8 through 11 were the ones added during the process in order to raise the score.

1. **Can do left arc** Whether or not the algorithm is able to perform a left arc in the current state.

2. **Can do right arc** Whether or not the algorithm is able to perform a right arc in the current state.

3. **Can reduce** Whether or not the algorithm is able to perform a reduction in the current state.

4. **First POS from stack** The part of speech from the token on top of the stack.

5. **First POS from queue** The part of speech from the first token in the queue.

6. **Second POS from stack** The part of speech from the second token in the stack.

7. **Second POS from queue** The part of speech from the second token in the queue.

8. **Third POS from stack** The part of speech from the third token in the stack.

9. **Third POS from the queue** The part of speech from the third token in the queue.

10. **First deprel from stack** The dependency relation from the token on top of the stack.

11. **First form from stack** The form, or lexicality, of the token on top of the stack. This feature is different in the sense that it looks for the most common words in the corpus (which were determined beforehand) and ignores tokens with other form.

5 Results and Discussion

After gold standard parsing and the training of a classifier, the parser was set to parse the test file from CoNLL-X, disregarding the columns that later were going to be produced by the parser. The resulting output was stored in a file of the same format as the test file.

In order to evaluate the result, a CoNLL evaluation script was used. This script assumes that the test file is correct gold standard and deducts points for deviations occuring in the output file. More precisely, the score is

$$score = 100 \cdot \frac{tokens_{correctly\_parsed}}{tokens_{gold\_standard}}$$

The results in section **??** represent the full output of the evaluation script. The project has implemented an unlabeled parser; however the result still shows a labeled attachment score and a label accuracy score, none of which is zero.

The reason why these scores are not zero, is that the parser reuses the input file. Since both the test- and training input files contain both labels and dependencies beforehand, and because the parser changes only the dependencies, the labels remain intact. This results in the label accuracy getting a score of 100% and the labeled attachment score to be the same as the unlabeled attachment score.

The reader is advised to simply ignore the results of labeled scores.

The script is run as below:

```
perl eval.pl -q
-g swedish_talbanken05_test.conll
-s resultWeka.conll
```

5.1 Project results

When running the script on the initial seven features, the results were taken as the baseline. This baseline scored as follows:

```
Labeled   attachment score:
    4054 / 5021 * 100 = 80.74 %
Unlabeled attachment score:
    4054 / 5021 * 100 = 80.74 %
Label accuracy score:
    5021 / 5021 * 100 = 100.00 %
```

When the part of speech tag for the third word on the stack and the third word from the queue were added to the set of features, the score increased with 0.26 percentage points.

```
Labeled   attachment score:
    4067 / 5021 * 100 = 81.00 %
Unlabeled attachment score:
    4067 / 5021 * 100 = 81.00 %
Label accuracy score:
    5021 / 5021 * 100 = 100.00 %
```

With the addition of an tenth feature, the dependency relation from the top token on the stack, the result managed to climb a bit more; 0.52 percentage points. This is an increase of 0.78 percentage points since the baseline.

```
Labeled   attachment score:
    4093 / 5021 * 100 = 81.52 %
Unlabeled attachment score:
    4093 / 5021 * 100 = 81.52 %
Label accuracy score:
    5021 / 5021 * 100 = 100.00 %
```

It was decided that lexicality should be added as an eleventh feature, by using the *form* of the top word on the stack. The program was modified to look for the 22 most common words (listed in Appendix A) in the training file. This number was arbitrarily selected, by a pseudo random procedure.

The resulting score now rose 1.16 percentage points from the baseline, which seems like a very good result.

```
Labeled   attachment score:
    4112 / 5021 * 100 = 81.90 %
Unlabeled attachment score:
    4112 / 5021 * 100 = 81.90 %
Label accuracy score:
    5021 / 5021 * 100 = 100.00 %
```

In order to see the difference between different classifiers, a quicker classifier, *Naïve Bayes*, was used. The result was remarkably worse, even with the same amount of features.

```
Labeled   attachment score:
    752 / 5021 * 100 = 14.98 %
Unlabeled attachment score:
    752 / 5021 * 100 = 14.98 %
Label accuracy score:
    5021 / 5021 * 100 = 100.00 %
```

It was later realized than due to an error in the counting method, the most common words picked were not in fact not the most common words. After replacing the word list with the correct one and parsing the input using the J48 classifier, the result was actually lower.

This, it was speculated, could be due to the fact that the most common words may be featured in the corpus with more varying other features, such as part of speech. In contrast, *less* common words, may actually be a better measurement, as they should be more likely to have consistent features. The result when using the correct words (see table **??**) was:

```
Labeled   attachment score:
    4104 / 5021 * 100 = 81.74 %
Unlabeled attachment score:
    4104 / 5021 * 100 = 81.74 %
Label accuracy score:
    5021 / 5021 * 100 = 100.00 %
```

This is still 1.0 percentage points above the baseline, and a good result.

An increased number of features might have given a better score. However, there is a very large amount of combinations of words from the queue and the stack and the features that define them. Finding the right combination consists of trial and error, as processing certain features may actually lower the score.

6 Related work

Many different dependency parsers were submitted to the CoNLL-X conference held in June 8-9, 2006 (CoNLL-X, 2006) and were all compared by the rate of their accuracy for different languages. The accuracy was calculated with Dan Bikel's Randomized Parsing Evaluation Comparator (Statistical Significance Tester for evalb Output) (Bikel, 2006), which has also been used to evaluate the scores for the implementation in this project. The best labeled attachment accuracy score for Swedish was 84.58% from the algorithm developed by Joakim Nivre, Johan Hall, Jens Nilsson, Gulsen Eryigit, Svetoslav Marinov from Växjö University, Istanbul Technical University and University of Skövde. The best unlabeled attachment accuracy score for Swedish was 89.54%, by Simon Corston-Oliver and Anthony Aue for Microsoft Research.

7 Conclusions

Comparing the results achieved in the project with the scores shown in the state of the art implementations, it is clear that our implementation needs many improvements to be interesting in such a competition. Still it was very interesting to develop the implementation by adding new features and using the different classifiers.

When implementing the algorithm for other languages than Swedish, there are sometimes other tags availiable in the corpus that can be used to form new features. Some of the tags that were not included in the Talbanken05 corpus, were missing for the simple reason that they do not apply to the structure of the Swedish language.

We were told that the J48 algorithm that was used in the project was not the best algorithm in terms of producing the best classifier, other implementations using LibLinear or SVM (Support Vector Machines) could have resulted in reaching higher scores. The implication with using LibLinear for the project was that it proved to be too tricky to incorporate in the original implementation of the algorithm that was supplied from the lab.

Initial attempts to use LibLinear led to the insight that it took too much time from working towards the main goal of the project, so we went back to using the available implemetation with Weka. An additional from-scratch attempt was made in parallel with the main version, to implement LibLinear support alongside Weka support. This attempt was abandoned as too time consuming.

We learned that SVM could prove to be significantly better than using Weka, except that its computing times were much longer, and therefore not applicable in the given scope of the lab.

The adding of new features led to a constant increase of accuracy scores when evaluated and we believe that it would be interesting to try adding more features to explore how the scores can further be improved.

References

Dan Bikel. 2006. Randomized parsing evaluation comparator. `http://www.cis.upenn.edu/~dbikel/software.html#comparator`, last accessed 2009-01-09.

CoNLL-X. 2006. Conll-x 2006 shared task: Multilingual dependency parsing. `http://nextens.uvt.nl/~conll/`, last accessed 2009-01-08.

Joakim Nivre and Sandra Kübler. 2006. Dependency parsing. Tutorial at COLING-ACL, Sydney 2006. `http://stp.lingfil.uu.se/~nivre/docs/ACLslides.pdf`, last accessed 2009-01-09.

Joakim Nivre. 2005. Talbanken 05. `http://w3.msi.vxu.se/~nivre/research/Talbanken05.html`, last accessed 2009-01-08.

Pierre Nugues. 2006. *An Introduction to Language Processing with Perl and Prolog*. Cognitive Technologies. Springer.

Pierre Nugues. 2008a. Assignment 5: Dependency parsing using machine learning techniques. java version. `http://www.cs.lth.se/EDA171/cw5-java.shtml`, last accessed 2009-01-08.

Pierre Nugues. 2008b. Eda171 course web. `http://www.cs.lth.se/EDA171/`, last accessed 2009-01-08.

Pierre Nugues. 2008c. Nivre parser implemented in java. EDA171 course web. `http://www.cs.lth.se/EDA171/Programs/parsing/Nivre.tar`, last accessed 2009-01-08.

Ross Quinlan. Ross quinlan's personal homepage. `http://www.rulequest.com/Personal/`, last accessed 2009-01-09.

WEKA. 2009. Weka. `http://www.cs.waikato.ac.nz/ml/weka/`, last accessed 2009-01-08.

A Tables and Figures

och	5348
att	5269
i	4420
är	3563
det	3412
som	3395
en	3181
av	2330
på	2303
för	2253
man	2082
den	2076
de	1817
inte	1795
har	1684
med	1667
till	1666
ett	1557
om	1500
kan	1316
sig	1114
så	834

Table 1: The 22 most common words in the training set

B Running the program

The program is implemented in java and it includes a few libraries. Since the project was initially meant to include LibLinear integration, two LibLinear libraries are included. Support for LibLinear was only partly implemented and so this functionality is not available. Although the program will run in arff mode without the liblinear libraries, they were included in the release as well as in the JAR manifest file's classpath, for reference.

The libraries required to run are included in the `lib` folder, and the source code resides in `src`. The *data* folder contains the CoNLL-X files used for input, the evaluation script, the final arff file and model and also the parsed output from Naïve Bayes and J48.

In the root folder of the release is a JAR file containing the packaged code. In order to run the program the following commandline is used for training:

```
java -Xmx1024m -jar ExtendedNivre.jar
-train -(arff|liblinear)
input_file output_file
```

and the one below for parsing:

```
java -Xmx1024m -jar ExtendedNivre.jar
-parse -(arff|liblinear)
arff_file model_file input_file output_file
```

The first parameter decides whether the parser should train a classifier or parse an input file. Since liblinear is not fully implemented, the second parameter should always be `-arff`.

When running in training mode, the input file should be
`swedish_talbanken05_train.conll`
residing in the data folder. The output file will be an arff file, and the user must manually use Weka to process this file to generate a model.

When parsing, the user should supply the arff file and classifier model created in the training phase. The input should be
`swedish_talbanken05_test.conll` and the output will be a parsed corpus in CoNLL-X format. The result of this, can be evaluated by using the evaluation script in the data folder.