Dependency Parsing

Axel Antonsson
Department of Computer Science
Lund University, Sweden
pi05aa8@student.lth.se

Jimmy Pettersson
Department of Computer Science
Lund University, Sweden
pi05jp6@student.lth.se

Abstract

The goal of this project was to write an efficient dependency parser for Swedish. Dependency parsing is a way of structuring a sentence by assigning every word a head and thus linking the words in a tree of dependency relations. This has been done by extending a pre-written implementation of Nivres algorithm. The focus has been to increase the score for unlabeled parsing and the main work has therefor been placed on finding an optimal feature set.

1 Introduction

This report is the final project in the course Language Processing and Computational Linguistics, EDA171 (Nugues, 2009a). The goal was to chose any field in either language processing or computational linguistics and implement a program of said topic. The chosen topic was to extend an implementation of Nivre's dependency parser algorithm. Dependency parsing is briefly put a technique of finding links between words in a sentence an thus finding the overall structure of every sentence in a text. In the CoNLL-X (Tenth Conference on Natural Language Learning) Joakim Nivres algorithm was the most successful at labeled dependency parsing (CoNLL-X, 2006) and since one of the programing assignments of the course were to improve an implementation of this algorithm the choice to use the implementation as a foundation for this project was natural (Nugues, 2008a; Nugues, 2008c).

The corpus used in this project is the hand-annotated Swedish corpus from Talbanken (Nivre, 2005). This was the same corpus that was used in the CoNLL-X conference. This implementation uses a data mining tool called Weka which features many different statistical classifiers. The classifier used in the project was the J48 decision tree. These tools was together with an implementation of Joakim Nivre's algorithm used throughout this project.

The main goal of this project was to increase the score for unlabeled dependency parsing and the main work was spent on the features since this was the simplest way to increase the overall score. Many different features was implemented and most of them were useful in the original feature set but many of them were discarded since too many features decreases the score. Finally a so called optimal feature set was found although more work could be spent on finding an even better one.

2 Corpora

A corpus is a collection of data which is correctly hand-annotated with respect to the gold standard. It is crucial when training a parser that you have access to a to good corpus. What makes a good corpus is not only the size but the diversity of the corpus to make it as representative of the language as possible. The corpora were originally hand-annotated but today are now mainly derived by an automated process, which later is corrected by hand.

CoNLL-X defined a format of the corpora used in the competition which contains ten columns for the features and is encoded in UTF-8. The set of ten features in the corpora represents a token and is expressed by a word or a punctuation sign. The

sentences are numbered from one and up and each sentence is ended by a blank line.
The layout of a single token in the corpora is demonstrated below.

ID	The token counter starting at one for each sentence.
Form	The lexical form of the word/punctuation mark is shown here.
Lemma	This is the basic form of a word, i.e the words eat, eats, eating etc. has the lemma eat. How this word is chosen can differ from corpus to corpus and varies with different languages.
CPOS	Coarse Part-Of-Speech. The CPOS tag represents a more coarse part of speech tagging while the POS tag is more precise. The CPOS tag set varies with different languages.
POS	Fine Part-Of-Speech. The POS tag represents classes of words that have common grammatical properties. Also known as word class.
Features	Unordered set of syntactic and/or morphological features. Depending on the language and type of corpus a word class can have a different amount of features.
Head	This is the head of the current token which takes the value of ID or zero if the tokes is the ROOT.
Deprel	Dependency relation. Shows the relation between the head and its dependent.
Phead	Projective Head. Shows the projective head of the current token which takes the values of ID, zero (if ROOT), or underscore if not available. It represents the best tolerable dependent of the real head while keeping the graph projective.
Pdeprel	Projective Dependency Relation. Equivalent to the Deprel but shows the relation between the projective head and its dependent.

3 Shift-Reduce algorithm

The shift-reduce algorithm takes the input file which is built up token by token in the queue. The tokens are then partially processed one by one and put in the stack. Depending on a set of rules different actions will be taken on the processed tokens. The set of rules are as follows:

Shift Push the fist token of the queue onto the stack

Reduce Pop the stack

Right-Arc Add an arc from the token on top of the stack to the preceding token in the queue and pushes that token to the top of the stack.

Left-Arc Add an arc from the preceding token in the queue to the token on top of the stack and then pop the stack.

4 Gold Standard Parsing

Gold standard parsing consists of taking a hand annotated corpus as input and then using the shift-reduce algorithm to determine what sequence of actions what would produce the best result (best being the same result). To determine what action to take the following scheme is used:

- If the word at the top of the stack is the head of the word first in the queue then do a right-arc.
- Else if the first word in the queue is the head of the word in the top of the stack then do a left-arc.
- Else if the there is a head anywhere in the dependency graph to the first word in the queue then do a reduce.
- If none of the above applies then do a shift.

5 Dependency Parsing

Dependency parsing is a technique that utilizes the internal relations of a single sentence to parse it. For all the words in the sentence, the root word excluded, there is a head which in the state of parsing helps to build the graph. The word without a head, the root word, is assigned as a dependant of the auxiliary word ”ROOT”. This is done to make the parser implementation easier and to show where the sentence begins. Figure 1 shows an example sentence. It consists of nine words, punctuation sign and ROOT included. The arc points from every word to its head, the only word left without a head is ”ROOT”.

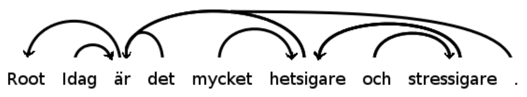

Figure 1: Sentence with unlabeled arcs

6 Training and parsing

6.1 Training

When training the classifier a stand alone software for machine learning called Weka was used (Waikato, 2009). In order for Weka to work with the data a so called arff-file has to be provided to the software. Weka comes with a set of classifier algorithms that are used for training the data. The algorithm J48 was used in this project to train the classifier, it builds a decision tree that Weka exports as a model-file. This model is then used in the parsing of the data to predict what action to take given the features provided.

6.2 J48

The classifier algorithm used by Weka is known as J48. J48 is an extension of an earlier algorithm developed by J. Ross Quinlan (2009), the C4.5 algorithm. J48 produces a decision tree and decision trees are a typical way to represent information from a machine learning algorithm, and offer a fast and powerful way to express structures in data. There are however a lot of other different classifiers that can be used in machine learning such as Support Vector Machines, Linear Regression, Neural Networks etc.

6.3 Parsing

When parsing the data the trained model produced by Weka is used to determine what actions to take in the parsing state. The result from the parsing is a dependency graph that can be translated in to a datafile in the CoNLL format. This output file can then be compared to another file in the CoNLL format and provide a score based on the number of deviations from the gold standard.

7 Features

The main focus of this project was the implementation and selection of features. Many different features was implemented and most of them yielded a higher score in some feature set but only a handful was kept in the final and optimal feature set. The initial feature-set were the following: first_postag_stack, first_postag_queue, can_ra, can_la, can_re and action. After this many different new features was implemented. First some of the most obvious such as more POS tags from stack and queue, the POS tag from the Head of the stack and the queue and also some lexical features. Although one could easily find features that improved the score from the original feature set it soon became obvious that finding features that worked well together was a bit more complex. The features that ended up in the so called optimal feature set were:

Top_pos_stack The POS tag of the first word in the stack

Second_pos_stack The POS tag of the second word in the stack

First_pos_queue The POS tag of the first word in the queue

Second_pos_queue The POS tag of the second word in the queue

Third_pos_queue The POS tag of the third word in the queue

Fourth_pos_queue The POS tag of the forth word in the queue

FirstHeadPostag The POS tag of the head to the first word in the queue

SecondHeadPostag The POS tag of the head to the second word in the queue

Random1 See section 7.1

Random2 See section 7.1

DeprelHead The dependency relation from the first word in the queue to its head

PredPostagStack The POS tag of the word immediately before the word in the stack in the original sentence

FirstWordStack The lexical value of the first word in the stack. See section 7.2

Can_do_leftarc Whether or not it is possible to do a left arc in the current state

Can_do_rightarc Whether or not it is possible to do a right arc in the current state

Can_reduce Whether or not it is possible to reduce in the current state

Action The action preformed.

7.1 Random features

Perhaps the most interesting part of this project are the two random features. These were found during a debugging and were never intended to be features at all. Nevertheless they were among the features that increased the score the most so they were eventually included in the final feature set. If these features have any real meaning or could be explained from a theoretical point of view is left unanswered in this project.

To understand the random features one must first understand how the dependency graph is represented. In the implementation of Nivres algorithm used in this project the graph is represented by an arraylist. Every time a right or a left arc is created the element is placed last in the arraylist. The dependency relations are represented by a integer value in each element called Head. This value simply tells you which word is the head of any given word in the graph. The feature Random1 checks if it is at the given state possible to do a right arc, and if so it extracts the last element in the graph. The feature Random2 on the other hand checks if its possible to do a left arc and if it is it extracts the first element in the graph.

7.2 Lexical features

The lexical features were the hardest to implement because the number of words included would have to be carefully fitted. The parser is given a list of words and then checks any incoming word if a match is made in the list, if so the parser returns the word itself otherwise it returns the string ”Nothing”. The words to include in this list turned out to be of great significance. Many different sets of words were tested and a list of just 17 words were found to be the best fit of the many different possibilities tried. The words included were the following: ”och”, ”att”, ”i”, ”det”, ”som”, ”en”, ”av”, ”man”, ”den”, ”de”, ”inte”, ”har”, ”med”, ”till”, ”ett”, ”om” and ”kan”. More work could be done in the selection of the words to include/exclude.

Many features were discarded but some of the features that at the time they were implemented yielded a higher score may still be worth mentioning. These features included:

X Postag Stack/Queue Much experimentation were made with how many POS tags from the stack and queue would give the highest score.

Leftmost/Rightmost Children from Stack/Queue Others have used this features in their final feature-set, but in the one used in this project they reduced the score.

Succ. Postag Stack and Pred. Postag Queue These features are more or less the same as predPostagStack but would not fit in the final feature-set

First Word Queue and Second Word Queue In the final feature set only one lexical feature were included

Stack/Queue ID The position of a word in the original sentence. On the time it was implemented it increased the score but ware later removed

# in graph Number of elements in the graph at a given state.

Stack/Queue in graph The number of times the word in the queue/stack occurred in the graph.

Much work could still be done in the feature selection but perhaps a more systematical method preferably scripted would be preferred to the Edisonian approach tried in this project.

8 Results

The final results for our unlabeled dependency parsing was 88.95%. This value represents the number of words assigned a correct head divided by the total number of words. This is an increase of 16.69 percentage points from the original implementation. We wont go over the progression of the score chronologically as the work progressed because of the sheer number of different feature sets tested. Instead we will include only a handful of the feature sets to give the reader a feel of the importance of different features. One must keep in mind the difference between the improvement a feature yields when added to the original feature set and the very same feature added to a set of already "high scoring" features. As previously mentioned many of the features implemented yielded a higher score at the time of implementation but were later discarded when they were found out to lower the score in a different set of features. With that said we did not achieve our highest score by adding new features in the later stages of the project but rather removing features to get a higher score.

The first improvement made was to alter the number of POS tags included from the stack and the queue. We ended up taking two POS tags from the stack and four POS tags from the queue. These extra feature alone increased the score to 80.08% which is almost 50% of the total increase in percentage points. For the next improvement we looked at the POS tags for the head of both the first and the second word. Some experimenting was done here with the number of POS tags to include and taking only the first and the second proved to result in the highest score. With these POS tags the score increased to 84.86%. Next step was to look at the dependency relation between words in the queue and their respective heads. Here it was made clear that if more than one word from the queue was included it would result in a decrease in score but when including only the first word in the queue the score increased, and together with the other features already added the score was increased to 87.45%. Next we looked at the POS tags of the predecessor and successor words in the stack. What this means is that you take the first word in the stack and look at the POS tag of the word that is immediately before/after this word in the original sentence. We ended up only having the POS tag of the predecessor since the POS tag of the successor resulted in a lower score. With this feature added the score rose to 87.91%. In the later stages of the project we started working with lexical features, which we knew decision trees such as J48 did not handle very well so many of the lexical features implemented yielded lower scores time and time again. However the feature "first word in stack" which is explained above did after some trial and error work put the score up to 88.47%. Finally the two random features were added on top of all these features mentioned above and the final score 88.95% was reached.

9 Improvements and discussion

Much could be done to improve the results in this project. The most obvious and important change would probably be the change of classifier. Either Linear Regression or Support Vector Machines would probably result in a score above the 90% mark although this is hard to confirm before the implementation is done. Implementing these would result in a need to re-pick the feature-set seeing the feature-set is optimized for the J48 classifier. This re-pick of features could then be done either manually as it was done in the project or it could be done in a more systematic way. Even if the J48 classifier was kept the results could probably benefit from this approach to the selection of features. If one were to script an optimization of the features we propose it would be done in the following way.

First a small part of the training-set would have to be broken off from the rest of the set and be used to train the classifier since training it on the actual training-set would result in an over-fit to this set. Then a script would have to be written to automatically choose different sets of features to try out. One could probably come up with hundreds of features or variations of features that could be tested but testing these manually would not be feasible seeing the total number of combinations would be enormous. The script could then start from a small set of features and every time make the greedy choice, that is to pick the feature from the list that gives the highest increase in the result. It could also be done the other way around that is to start the script with all the features and remove the greedy choice. This approach would probably get near the optimal subset of the features implemented but it can not guarantee that an optimal solution would be chosen. Another way to find this optimal subset would be a genetic algorithm. Letting a solution evolve in the genetic algorithm would probably be very time consuming but could find solutions that the greedy algorithm could not. Even a combination of the two would be possible, letting the genetic algorithm run and choose a different solutions and then see if any greedy choices were possible from these.

References

Pierre Nugues. 2009a. Eda171 course web. http://cs.lth.se/eda171/

Pierre Nugues 2009b. Assignment 5: Dependency parsing using machine learning techniques. http://http://cs.lth.se/english/course/eda171/coursework/assignment_5_java/

Pierre Nugues. 2009c. Nivre parser implemented in java. http://fileadmin.cs.lth.se/cs/Education/EDA171/Programs/parsing/Nivre.zip

WEKA. 2009. http://www.cs.waikato.ac.nz/ml/weka/

Joakim Nivre. 2005. Talbanken 05. http://w3.msi.vxu.se/ nivre/research/Talbanken05.html

CoNLL-X. 2006. Conll-x 2006 shared task: Multilingual dependency parsing. http://nextens.uvt.nl/ conll/

Ross Quinlan. Ross quinlans personal homepage. http://www.rulequest.com/Personal/