Automatic learning of dependency rules from corpora

A project in the course
EDA171/DAT171 Language Processing and Computational Linguistics

Tomas Rutegård e99, Bibi Sandberg e00 and Johan Larsson dat00

Abstract

This paper presents work done in project form in the course Language Processing and Computational Linguistics given at Lund School of Technology during the fall of 2004. The work develops and assesses a new dependency parser for Swedish, based on decision trees learned from corpora. To assess the parser, it is trained and tested on a subset of the MALT corpus and found to perform fairly well considering its stage of development.

1 Introduction

1.1 Project purpose

The given purpose of the project presented in this paper was to construct a software system for the automatic learning of grammar rules used by a certain text parser: the Nivre parser. The Nivre parser is a dependency parser developed by Joakim Nivre [1]. The Nivre parser uses a special kind of D-rules, namely directed D-rules, for parsing.

As it happened, we, the authors of this paper, decided to instead develop our own parser, a parser also using rules derived from corpora, but rules of a different kind, used in a different way. The automatic learning algorithm of our parser is decision tree induction algorithm, simple yet not powerless.

2 Short introductions of theory

2.1 Dependency Grammars

According to the book *An Introduction to Language Processing with Perl and Prolog* [3] dependency grammar is used for describing the structure of a language. It is especially good for describing languages where the word order is flexible. This is the case for Latin and Russian but not for English and Swedish. The rules can be helpful in translations or just for understanding the language.

Every word in a sentence is the dependent of one head with one exception. This exception is the head of the sentence, also called the root, which only have dependants. The head of the sentence is generally the main verb but can also in rare cases be a noun. These dependency rules are marked as arrows from the dependant to the head. The root is marked with an arrow pointed at the top of the screen.

The basic dependency rules are that a dependant links to its noun and a subject noun links to its main verb. Other rules are that determiners and adjectives are dependents to their noun and adverbs to their adjectives. One example is Figure 1.

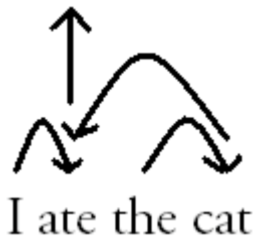


Figure 1. *An example of how dependency grammar can look like*

Here *"ate"* is the main verb and also called the root. The two nouns *"I"* and *"cat"* are the dependants of *"ate"* and the determiner *"the"* is the dependant of its noun *"cat"*.

2.2 Decision tree induction

2.2.1 The decision tree structure

Consider some object or situation to which some set of attributes is related. A decision tree is a structure associating with each possible set of values of the attributes some value, thus constituting a function from the set of possible sets of attribute values to an arbitrary set. If the set of values associated with possible sets of attribute values is discrete, the values associated with possible sets of attribute values are called classifications.

A decision tree is a tree data structure. Each non-leaf node of a decision tree represents a test of one of the attributes of the attributes set and each outbound branch from a non-leaf node represents one of the possible values of the attribute tested in that node. Each leaf node of a decision tree represents a value assigned to some subset of the set of possible sets of attribute values. The value associated by a decision tree with a given set of attribute values is the value represented by the leaf reached when traversing the tree from root to leaf, in each node choosing branch according to the value of the attribute tested in that node.

Decision trees are simple yet somewhat expressive. Any Boolean function can be written in the form of a decision tree, though by necessity some, for example the majority function, are quite large in the form.

2.2.2 The induction algorithm

Let an example of a function be a member of the domain of the function and the associated member of the codomain of the function. A set of sets of attribute values and values associated with these sets of attribute values can be regarded as a set of examples of some function whose domain comprise the attribute values and whose codomain comprise the associated values. The forming of a function consistent with the examples approximating the function exemplified is referred to as inductive inference. The formed function is called a hypothesis.

The algorithm in Figure 1 [2] forms a consistent hypothesis in the form of a decision tree for any set of examples which are examples of a classifying decision tree or some function that can be written in the form of a classifying decision tree. In Figure 2, the goal predicate is an attribute whose value for any set of attribute values is the value associated with that set of attribute values. The algorithm applies Ockham's razor and prefers the simplest hypothesis of a set of hypotheses all consistent with the examples. Forming the smallest consistent hypothesis is an intractable problem, though. The algorithm forms a smallish one. CHOOSE-ATTRIBUTE chooses the attribute which provides the most information, in the mathematical sense.

```
function DECISION-TREE LEARNING(examples, attributes, default) returns a decision tree
        inputs:     examples, set of examples
                    attributes, set of attributes
                    default, default value for the goal predicate
        if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MAJORITY-VALUE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        m ← MAJORITY-VALUE($examples_i$)
          for each value vi of best do
              $examples_i$ ← {elements of examples with best = $v_i$}
              subtree ← DECISION-TREE-LEARNING($examples_i$, attributes – best, m)
              add a branch to tree with label $v_i$ and subtree subtree
          return tree
```

Figure 2. *An algorithm forming a smallish decision tree consistent with a set of examples.*

3 The parser

3.1 *The decision trees generated and the parser algorithm*

This section presents the parsing algorithm we have made. It uses four different decision trees in order to classify the correct part-of-speech tags in a sentence.

We will first describe how the decision trees have been generated.

3.1.1 Is-root

This tree is used to find the most probable root in the sentence. It has learned by looking at the part-of-speech tag and a context of two words on each side. In the example shown in Figure 3, a correct classified sentence is shown. The word ”är” is root in the sentence. So for every word in the sentence, we will add its part-of-speech tag and its context to the database together with a true/false that tells if its the root. Later in the parsing algorithm, the decision tree tries to classify when something is root and not.

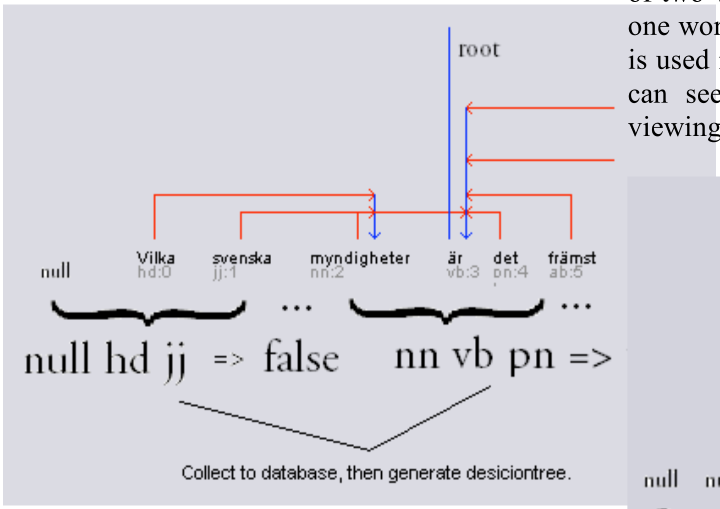


Figure 3. *Is-root*

The tree classifies 80% of all sentences (on an unseen domain) correct. When it fails, there are more than one candidate to be root (the most likely will be chosen).

3.1.2 Find-head-1

This decision tree is trained by taking the part-of-speech (pos) tag and a context of one word on each side, and trains it to find the correct head to the pos-tag. The example in Figure 4 shows how the selection has been made. This is added for every word except root (since it doesn’t have any parent).

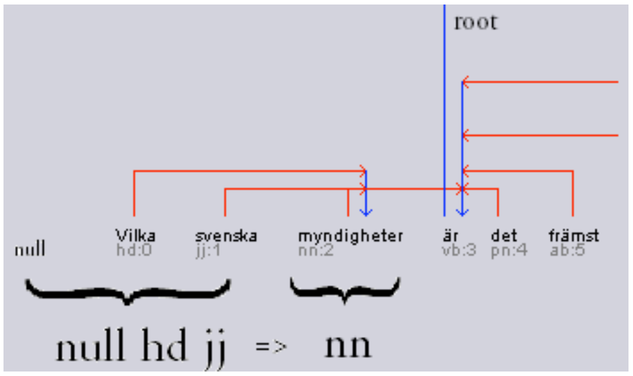


Figure 4. *Find head 1*

3.1.3 Find-head-2

This is an extension of Find-head-1 that is trained by looking at the pos-tag, a context of two words on each side, plus a context of one word on each side of the head. How this is used in practice is explained later, but you can see what the database looks like by viewing the example in Figure 5.

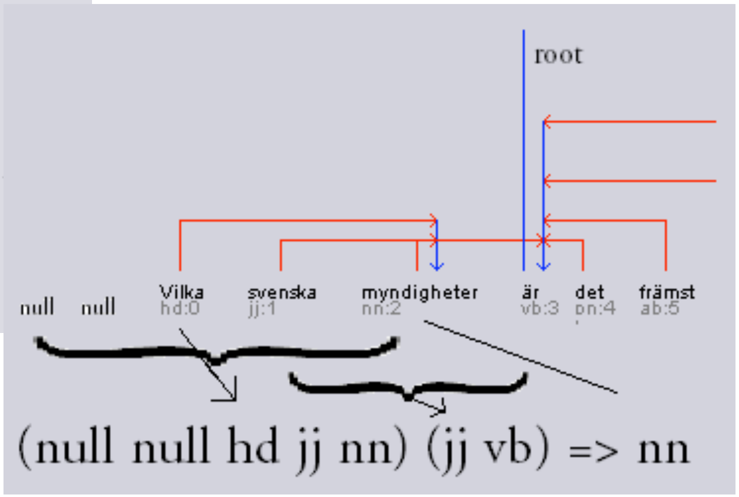


Figure 5. *Find head 2*

3.1.4 Find-Direction

The database that is used to train this tree looks almost the same as the one used to train Find-head-2. In addition it also has the direction of the arrow (right/left) that is used as goal-attribute in the decision tree learning algorithm. This makes it able to classify the most probable direction.

3.2 The parsing algorithm

This algorithm is used to demonstrate how we can use decision trees to classify all words/pos-tags in a sentence. It uses the four decision trees explained above. In practice it works in three different phases. It is at the moment quite simple, and could easily be expanded to use more decision trees and more advance functions. The following explains the three phases of the algorithm.

3.2.1 Phase 1

The first phase uses only the *find-root* and *find-head-1* trees described above. Its main goal is to find the most probable root, and all possible candidates for head for all words. The easiest way to show how this works is by using an example.

Lets say we want to find the head for the Swedish word “inkomsterna” (see Figure 6), this is a noun, and its closest neighbors is a determiner and a verb. We can ask *find-head-1* with this information, and it will answer by providing a list of the most probable neighbors, in this case verb (56.5%) and preposition (43.5%). The next thing we do is to go though all the words looking for prepositions and verbs, we add this to a list of *potential* parents. In this sentence the preposition at position zero, the verb at position three and seven would be added to the list of potential parents for our noun. We do this for all the words in the sentence.

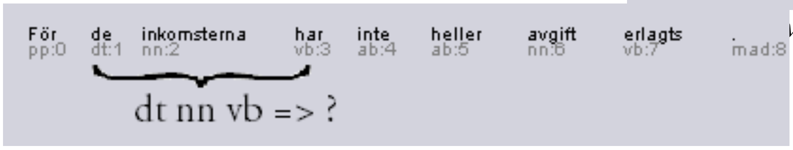


Figure 6.

3.2.2 Phase 2

We now have a list of head-candidates for every word except the most probable root. The next step is to go through all the candidates in the list (and we do this for every word) and ask *find-head-2* and *find-direction* how probable that candidate is. Since *find-head-2* is more accurate, its result gets higher ranked when choosing the candidate. In the example above, the preposition at position zero would get a score of 105, the verb at position three a score of 6.3 and the verb at position seven a score of 1.8. This means we will chose the preposition at position zero as our most probable head.

3.2.3 Phase 3

After phase 2, the algorithm is almost complete, however there may still be easy found errors in the complete graph. For example, we know that two arrows may never cross each other (see Figure 7 “A crossing link”). If they do, at least one of them is pointing wrong. We use a simple method to find and remove all crossing arrows.

We also look for cycles in the graph (see Figure 8 “A cycle”), and use a method to break these. This step is not optimized, but works quite well.

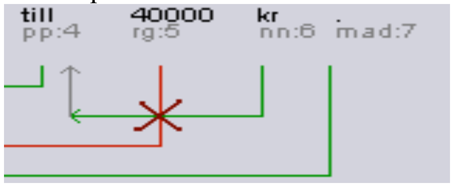
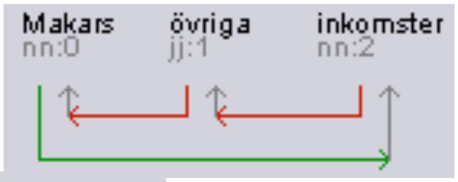


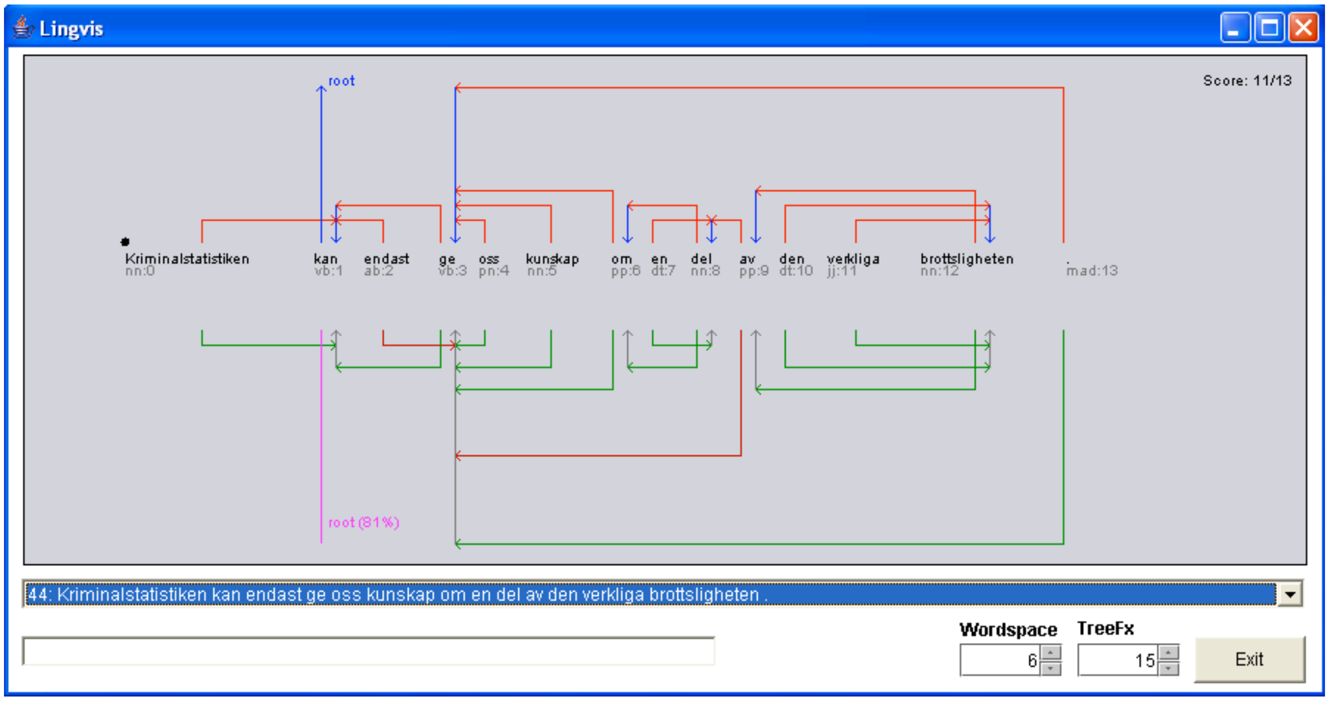
Figure 7. *A crossing link*



ycle

3.2.4 The complete graph

On this page the program and its GUI is explained, and an example of a complete graph is shown in Figure 9.



Score: 11/13

44: Kriminalstatistiken kan endast ge oss kunskap om en del av den verkliga brottsligheten .

Wordspace 6

TreeFx 15

Figure 9. *The complete graph*

3.3 The program and how to use it.

This is what the GUI for our program looks like. In the list a sentence from the loaded XML file can be chosen. When loading a sentence to classify, the correct answer is shown at the top (red and blue arrows), and the result of our algorithm is shown at the bottom (green arrows for correct classifications, and red arrows for errors). So in this sentence, 2 errors are present ("endast" -> "ge", and "av" <- "ge"). There is also another window (not showing here) that is used as log-tool. If we press a word, for example "av" the following info will show in the log-tool.

The first row in Figure 10 shows the chosen form, in this case "av". The next tells us what "av" links to (in this case "ge") and the "x-link: false" says its not crossing any other arrow. The tree shows all children (and their children) and also tells us if there are any cycles present. The next list shows all potential parents for "av". As you can se, the verb at position 3 has gotten the highest score together with the verb at position 1). The noun at position 8 (the correct one) has gotten just slightly less score. The "pr: false" tells us if the arrow goes past the root, in that case, it will get a penalty since its quite unlikely. The last list (rules) shows a more a abstract view of likely parents, telling us its is most likely to link to a verb.

```
Form: av
Links to: ge (x-link: false)
-tree------------------------------------------
<pp:9>
   <nn:12>
      dt:10
      jj:11
-----------------------------------------------
Pot.Parents: pp:9
   vb:3 (106.33559) (pr: false)
   vb:1 (106.33559) (pr: false)
   nn:8 (105.58499) (pr: false)
   nn:12 (61.73145) (pr: false)
   nn:5 (5.5849915) (pr: false)
   nn:0 (0.60047567) (pr: true)
-rules-----------------------------------------
vb (56.89655)
    left (100.0)
nn (41.37931)
    left (99.60318)
    right (0.39682543)
rg (1.7241379)
    right (100.0)
```

Figure 10.

4 Performance analysis and conclusions

4.1 The performance of the program

The program use approximately 800 sentences as a practice set and about 300 sentences as a test set. Best results are given when the program is training and testing on the same set of sentences i.e. the test set is 300 sentences that are selected out of the 800. In 300 sentences there are approximately 4000 arrows (there are approximately 8700 arrows in 800 sentences) and the program gets about 86.7 % of them correct and 73.5 % correct if the test set is new to the program. Sentences where all the arrows are correct is about 37.6 % when the program is training and testing on the same set and 19.3 % else. The training set was later expanded to 5000 sentences and the test set to 1300 sentences. But the result turned out to decrease. When testing on new sentences it gets about 70 % of the arrows correct which is a decrease by 3.5%. This probably has to do with the decision trees that get overfitted. A solution to this problem could be to improve the pruning of the trees.

4.2 Difficulties on the way

We started to use the neighbors of the dependant word to improve the program. Two neighbors to the right and two to the left were showing to be the most efficient so far. After this improvement we also added neighbors to the head word. The statistic improved by roughly 15 %.

To find the correct root in the sentence was harder then first expected. And when the program choused the wrong word as the root it generated more errors to the other arrows in the sentence. Improvements can still be done here.

Another problem that maybe is not that important, but contributes to lower the statistics, is that the arrow from the dot in the sentence more often is wrong than right.

In the beginning the program sometimes made loops which are not acceptable in dependency grammar. To solve the problem temporary an algorithm was made. The algorithm solves the problem with the loops (Figure 2) but not in the most efficient or most correct way. Here big improvements can be done.

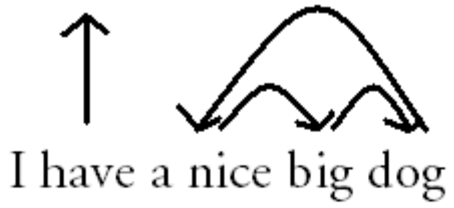


Figure 2. *An example of a loop*

Crossing arrows was also a problem in the beginning but was easily fixed with a small algorithm.

4.3 Future aspects, if more time was given

To find a better way to classify the root is a good start. The root is in most cases the main verb and to locate that easier it maybe would help to add some more attributes to the words. Such as how the verb is intransitive, transitive or ditransitive.

An improvement of the loop algorithm could be done by making the algorithm go trough all possible arrow changes before choosing. At the moment the algorithm picks out the first possible fit.

To examine how much improvement a larger training set will give has not given as good results as hoped for. This is because of the decision trees that get overfitted. A solution to this problem is maybe to get better pruning of the trees.

At the moment only the lexical categories are used as attribute to the words. To improve the program it could also examine how the most common words in a sentence relate to the other words. For example instead of using the attribute determiner to a very common word like *"the"* you could use the fact that the word *"the"* in most cases has the first noun to its right as the head.

To try the program on other languages would be an interesting project and not so hard to accomplish. The program is capable to read xml documents in a certain predefined type and if given, it could train and test on a different xml database in a new language.

5 References

[1] Nivre, J. (2003) An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, Nancy, France, 23-25 April 2003, pp. 149-160.

[2] Russel, S, Norvig, P. (2003) Artificial Intelligence: A Modern Approach. Second edition. Prentice Hall series in artificial intelligence.

[3] Nugues, P. (2004) An Introduction to Language Processing with Perl and Prolog. Lecture notes for the course Language Processing and Computational Linguistics given at Lund School of Technology.