A part-of-speech tagger for Swedish using the Brill transformation-based learning

François Marier
Lund Institute of Technology
fmarier@uwaterloo.ca

Bengt Sjödin
Lund University
bengtsjodin@hotmail.com

Abstract

This paper describes an implementation of a Brill Part-of-Speech tagger for the CarSim project. It introduces the concepts of part of speech tagging and Brill taggers and presents some results measured with the given implementation. Given the high running time of the Learning algorithm, very few results are available and these limitations, along with some partial solutions, are discussed.

1 Introduction

1.1 Goal

Our task was to implement a Brill part-of-speech (POS) tagger for Swedish using Java as the programming language. The focus was on the learner part of the algorithm as it is the most complex part of such a tagger.

1.2 The CarSim Project

The CarSim project[1] is an attempt to visualize written accident reports. The report is translated into a symbolic template, which is then used to generate a three-dimensional animation. Our tagger is supposed to be used in the translation part of the CarSim.

This is an important part because the texts are to be processed automatically, and as such the tagging of the words is vital for the information to be extracted correctly. Since all language are essentially an encoding of information, the program needs to decipher and reform it in meaningful ways so that it can be further processed into the final state.

For example the program must find the nouns and pronouns in the text and properly realize which of the words refer to separate entities and whether they are static or moving objects. This is done by splitting the text into sentences where the words then are tagged with their part-of-speech. The result is then used both for detecting road objects and clauses, which are used to fill event structures. These two things are what is needed to fill out the template.

1.3 Part-of-Speech Tagging

Part-of-speech (POS) tagging is also called grammatical tagging. It is one of the most common forms of corpus annotation. The labeling of the words in a sentence with their lexical or word classes is called tagging. The POS is divided between the open and the closed classes: The open class words are Nouns, Adjectives, Verbs and Adverbs whereas the closed are Determiners, Pronouns, Prepositions, Conjunctions, Auxiliaries and Modals.

Some morpho-syntactic information can also be provided, marking the word as a proper plural noun or a singular comparative adjective. This captures most of the information that a word contains.

One of the biggest problems with labeling a word correctly is disambiguation, meaning to find the intended form of the word, e.g. the word “can” could be a modal or a noun. An annotated text may be used to improve lexicons and otherwise help with the understanding and learning of languages.

[1] See http://www.lucas.lth.se/lt/carsim.shtml

1.4 Kinds of Taggers

The progress of automated part-of-speech taggers has gone from handwritten rules to Markov probability chains and on to machine-built rules. The early attempts required a large amount of repetitive work. Even the taggers that use Markov chains, though they achieve a high correctness probability, nest their rules inside a large set of such chains making them unreadable by humans.

The main advantages of the Brill tagger, with its machine-built rules, is that its rules can be easily transformed into a readable form and thus increase the human knowledge base. Also, it can be trained on top of a more complex pre-existing tagger to help improve its accuracy.

2 The Brill Method

A POS tagger that uses a transformation-based error-driven learner technique is called a Brill tagger.

Such a tagger must initially be trained to be able to tag a text correctly. A manually annotated corpus is used as a training reference. An initial state annotator first processes the same text without the tags. This annotator can be slightly complex, using statistics derived form the corpus, or very simple, merely tagging everything as nouns.

The machine-annotated text will then be compared with the reference noting the differences as errors. The algorithm will then try to find the most effective tag transformation rule, based on the current errors.

The rules are based on a number of templates that contain the structures and variables. For every error one instance of each type of rule is created. The most efficient rule is the one that corrects the most errors, and that one is found by letting all the various rules be applied to the text, remembering the best one.

The resulting text is, in each step of the learning algorithm, what is used as a base in the next thus always reducing the number of errors until a threshold is reached. The resulting set of rules is ordered by the amount of errors they correct.

Here is some pseudo-code describing the learning algorithm:

```
Annotate every word in the text
 with the most likely tag.

DO
   Create index of errors in
    annotated text.

   FOR every value in error index.
      Create all possible rules.

   Discard duplicate rules.
   Find rule with the largest
    error reduction.
   Store the rule.
   Apply the rule to text for
    the next iteration.

WHILE best rule fixes at least
 (some threshold value) errors.
```

2.1 Error Threshold

The Learner has to consider rules that fix more errors than a certain threshold. We have set this value to be 2 after performing a few tests on a very small training corpus. It appeared as if increasing this number would decrease the accuracy of the tagger since it discarded too many rules. However, bringing this number down to 1 also decreased the accuracy of the tagger.

We believe that this is due to the fact that the Learner would learn too many “bad” rules that might fix 1 error in the training corpus but would introduce many more in the test corpus. After all, a rule that fixes only 1 error overall on a large corpus can hardly be considered as representing a linguistic feature.

3 Implementation Overview

Here is a description of all the classes (see Figure 1) contained in our implementation.

Note that all of these classes are part of the se.lth.cs.BrillTagger Java package.

3.1 Main Programs

- **Tagger**: Class that does the actual tagging given an input file and a list of rules.
- **Learner**: Learns rules from the corpus.
- **FrequencyExtracter**: Extracts the tag frequency of each word from the corpus

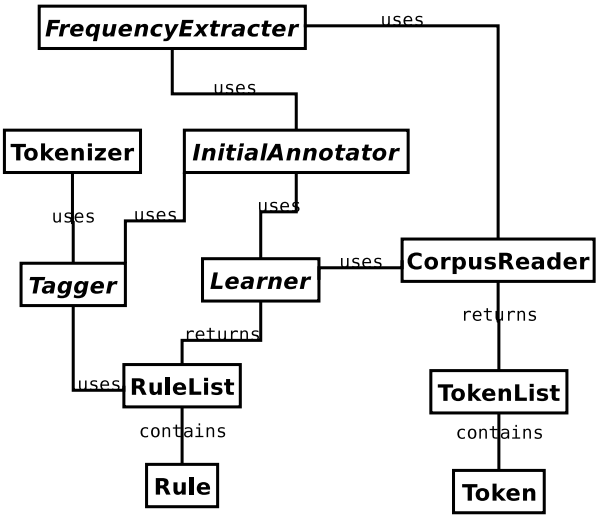

Figure 1: Class Diagram

- **InitialAnnotator**: The initial annotator takes a list of tokens and assign to them the most frequent tag.

3.2 Utility Classes

- **CorpusReader**: This class goes through the corpus, calling the ParseAction each time a word is encountered.
- **ParseAction**: Action to perform each time a new word-tag pair is extracted from the corpus by the CorpusReader. A typical action is to create a Token object containing the word and the tag.
- **RuleList**: Container for all instantiated rules learned by the Learner.
- **Token**: Basic unit of an annotated text.
- **Tokenizer**: Tokenizes a file into its grammatical components.
- **TokenList**: Basic data structure for manipulating an annotated text as a list of tokens (word and tag).

3.3 Rule Templates

- **Rule**: Base class for all the learned non-lexicalized rules.
- **RuleNotInstantiatable**: Exception thrown when there are not enough information to instantiate the rule. For example when trying to instantiate the PrevTagRule on the very first word of the corpus.
- **NextTagAndTwoBeforeRule**: Changes the current tag from source to destination if the next and second previous words are tagged in a certain way.
- **NextTagRule**: Changes the current tag from source to destination if the next word is tagged in a certain way.
- **NextTwoTagsRule**: Changes the current tag from source to destination if the next two words are tagged in a certain way.
- **OneOrTwoAfterRule**: Changes the current tag from source to destination if the next word or the one after that is tagged in a certain way.
- **OneOrTwoBeforeRule**: Changes the current tag from source to destination if the previous word or the one before that is tagged in a certain way.
- **OneOrTwoOrThreeAfterRule**: Changes the current tag from source to destination if the next word or the one after that or two words after is tagged in a certain way.
- **OneOrTwoOrThreeBeforeRule**: Changes the current tag from source to destination if the previous word or the one before that or two words before is tagged in a certain way.
- **PrevTagAndTwoAfterRule**: Changes the current tag from source to destination if the previous and second next words are tagged in a certain way.
- **PrevTagRule**: Changes the current tag from source to destination if the previous word is tagged in a certain way.
- **PrevTwoTagsRule**: Changes the current tag from source to destination if the previous two words are tagged in a certain way.

- **SurroundTagsRule**: Changes the current tag from source to destination if the previous and next words are tagged in a certain way.

- **TwoAfterRule**: Changes the current tag from source to destination if the second next word is tagged in a certain way.

- **TwoBeforeRule**: Changes the current tag from source to destination if the second previous word is tagged in a certain way.

4 User's Manual

4.1 FrequencyExtracter

The initial annotator must be run one time in order to generate the `tagfreq.dat` file that contains a hash of each word encountered in the corpus along with the statistically best tag. It is run in the following way:

```
java se.lth.cs.BrillTagger.
FrequencyExtracter training
```

where "training" is the directory containing the files from the training corpus.

The program will display the number of words added to the hash table.

4.2 Learner

The Learner must also be run once before the Tagger can be applied to a text. It will output a list of rules into the `rules.dat` file. It is run in the following way:

```
java se.lth.cs.BrillTagger.Learner
training
```

where "training" is the directory containing the files from the training corpus.

The program will initially display the corpus size and the baseline on the training corpus. Then it will display the selected rules along with the number of errors that they fix. Finally the program will print out the number of errors remaining as well as the number of rules that were learned and the accuracy on the training corpus.

4.3 Tagger

There are two ways to use the Tagger. One can run the tagger as a stand-alone program by passing the name of the file to tag:

```
java se.lth.cs.BrillTagger.Tagger
test.txt
```

The program will then print each word along with its tag on the same line. There will only be one word (or token) per line.

The alternative is to call the Tagger from within a Java program. The Test.java class, distributed with the implementation, gives an example of such thing.

Basically, one has to use the `Tagger.tag(Reader reader)` method in order to have the Tagger read and tokenize the input text. The result is a TokenList which has a similar interface than ArrayList. Each token can then be extracted and used by a larger program.

5 Results

All of these results were gathered by running the actual Learner discarding rules having fixing less than 2 errors. The baseline on the test corpus was 82.542%.

In the first experiment, the full 13 non-lexicalized templates were used. The results follow:

nb words	nb files	nb rules	training accuracy	tagging accuracy
2336	1	25	96.147	83.032
4849	2	58	96.02	84.07
7136	3	82	96.118	84.212

Another experiment was performed where we only used the NextTag rule template. The results of this second experiment follow:

nb words	nb files	nb rules	training accuracy	tagging accuracy
4849	2	47	94.839	83.954
24345	10	123	95.046	84.544

6 Evaluation

We have implemented the Brill learning and tagging algorithms entirely. The initial annotator we are using is the one that derives the initial tags statistically. We have also implemented all 11 non-lexicalized rule templates mentioned in the original Brill paper, as well as two other ones mentioned in the Roche and Schabes paper ("previous bigram" and "next bigram").

Our implementation does not include the unknown word tagging rules nor the lexicalized rule templates.

6.1 Running Time of the Learner

The learning algorithm takes a very long time to run. Because of the way the algorithm works, there is no easy way out of this excessive run time. Here is a table of the training time for very small corpus sizes.

nb words	nb files	training time
2336	1	6 min
4849	2	55 min
7136	3	157 min

All tests were performed on a Pentium III 650 MHz with 256 MB of RAM.

7 Enhancements

We have considered two enhancements in order to cut down on the learning time. Without having access to a profiler, we decided to attempt to speed up the Learner by attacking one at a time the two suspected bottlenecks. More work on the learner will be necessary to identify other bottlenecks that might explain the performance of the implementation.

7.1 String Comparisons

The first attempt involved speeding up tag comparisons by converting string comparisons to pointer comparisons. We used the "intern()" method of the String class in order to achieve that. However, simple tests revealed that this modification had increased the learning time by approximately 40% on a very small training corpus.

After thinking about this surprising result, we came to the conclusion that string comparisons were probably quite fast in most cases since for the vast majority of tag comparisons, only 1 or 2 characters would have to be compared before the two tags would be deemed different. Hence the overhead involved in creating a hash table of String objects would increase the running time of the algorithm.

This modification is not part of the final implementation.

7.2 Corpus Cloning

In order to apply all rules to the same corpus, we decided to provide the TokenList class with a `clone()` method. That way, we do not have to undo a rule before applying the next one. Unfortunately, this also means that a large data structure is copied quite often.

So we decided to make the Token class immutable. This slows down some aspect of tagging/learning since modifying the tag of a Token now requires the program to create a brand new Token object. However, it also means that the cloning method of TokenList does not have to perform a deep copy anymore. It can just reuse the same Token instances instead of calling their copy constructor.

This modification has reduced the learning time by 20%. It is part of the final implementation.

8 Conclusions

Our initial tests revealed that the algorithm does bring an improvement over the baseline, but its running time is excessively large and it is hard to predict what accuracy we would get if the Learner was allowed to run on the entire corpus.

We were surprised by the slow learning rate of the algorithm since we were expecting a few rules to significantly improve the accuracy of the tagger on the test corpus.

More testing on a larger training corpus needs to happen before the tagger can be used effectively.

We can hope that the preliminary results that we have presented here will scale well with an enlarged corpus and will deliver the kind of accuracy that Brill claimed to get in his paper.

9 Acknowledgements

We are very grateful for Pierre Nugues' insightful comments and for the weekly meetings with him that have kept us on track and allowed us to progress even though we ran into some problems.

References

E. Brill 1995. *Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging*. Association for Computational Linguistics.

Emmanuel Roche and Yves Schabes 1995. *Deterministic Part-of-Speech Tagging with Finite-State Transducers*. Association for Computational Linguistics.