

**The University of Texas at Dallas  
CS 6320  
Natural Language Processing  
Fall 2006**

**Class Project  
HMM Clause Detection Systems**

**Stanley S. Jointer II  
Nithya Bondalapati**

#### Abstract

Hidden Markov Models (HMM's) are widely used in many areas of natural language processing, as well as automated speech recognition and human feature recognition to identify places where not all information regarding a system is fully observable. In 2001, a competitive call for papers to detect clauses in writing yielded another potential use for HMM's. This paper will analyze the implementation of HMM's to the task of clause detection as presented by [2]. We explore the proposed solution and examine variations of the methodology that may provide better results in the future.

#### Problem Definition

“...the goal is to identify clauses in text...”[1]. The clause is defined as a sequence of words with a subject and a predicate. It is important to distinguish that, for this task, the software is not only to detect simple sentences, but also complex sentences with dependent, nested clauses. The software is required to detect three aspects with regards to clauses during various tasks: the start of a clause, the end of a clause, and a complete clause, which includes all nested, dependent clauses, as well as the independent, main clauses.

Both training and testing data was extracted from the Penn Treebank that contained portions of the Wall Street Journal (WSJ) corpus. The files were broken into four columns containing the current word, the word's part of speech (POS) “derived by the Brill Tagger”[1], a chunk tag based on a cited paper at [1], and classification of the clause for the current task taken from Treebank. Evaluation software was provided to identify the precision and recall for each implementation of clause detection.

Clause detection is important for several reasons. Among these, detecting clauses allows for better interpretation of the semantic roles for a sentence. Another reason for detecting clauses, especially dependent clauses, is to enhance and extend the semantic meaning embedded in the words used throughout the sentence.

#### Methodology

The main purpose of this paper is to analyze one of the contenders from [1]. Molina and Pla implemented the clause detection tasks using Hidden Markov Models and published their results and methodology in [2]. The purpose for choosing this particular paper over others represented was it had the most consistency in its precision and recall results. We suspected that, if we could detect some locations for improvement with the methodology, that the results would remain balanced, and the paper had the most room for improvements. This is not to say the original idea by Molina and Pla is weak in any manner, but rather to say that, based on their choice, if any improvements could be made, they could be represented across both precision and recall.

[2] uses standard HMM methodology, utilizing an relaxation of Bayes Rule to find the most probable task classification of a particular word. At the outset, [2] states that

$$P(O|I) = \frac{(P(O)P(I|O))}{(P(I))} \quad (1)$$

Since we desire the most likely output, we want to maximize P(O|I) across all possible input and output sequences. The resulting output for a given output sequence  $O = o_1, \dots, o_T$  is:

$$\hat{O} = \underset{o}{\operatorname{argmax}} \frac{(P(O)P(I|O))}{(P(I))} \quad (2)$$

As with all HMM's, the process of finding  $\hat{O}$  is independent of the input and can be reduced to a consideration of only the previous two output sequences. The resulting equation is

$$\underset{o}{\operatorname{argmax}} \left( \prod_{j=o_1}^{o_T} (P(o_j|o_{(j-1)}, o_{(j-2)}) P(i_j|o_j)) \right) \quad (3)$$

The first probabilistic term in (3) represents the probability of the output transition sequence while the second probabilistic term in (3) represents the chance for the input based on the given observation.

[2] then defines the input that is observed as an ordered pair of the given POS tag and chunking tag, while the output is restricted to the type of possible outputs for each task.

### Implementation

Our implementation made one minor change to the system as [2] defined it for their computations. [2] redefines the classifications for the tasks read based on the POS tag. In our system, we made no alterations to the output clauses. The system was developed using the ‘‘C’’ programming language, based on the strengths of team members and the methods used to program popular natural language software used in the field today.

Our implementation creates four linked lists, one for the input vocabulary defined by [2], one to store the two prior classifications observed as one object, a third list was used to store each observation singly, and a final list read and stored the input file. The lists were then used to create a simplistic has table, where each entry in the input list and prior list were considered rows in two different tables, the observation table and the prior table respectively, while the observation list because the columns of the two tables. We developed these simple hashes in order to create the tables we needed once, or maybe with one block reallocation. We will show later the error in this implementation for the tables.

Training was done by reading the input file into the data link directly. In reading these files, we stripped away all the ‘‘blank lines’’ placed into the training and testing files. We will also show later the ill effects of making the assumption that ‘‘blank lines’’ contained no information for this system. Once the file structure was filled, we iterated through the list, storing information regarding the prior two observations, using start symbols at the beginning of the sentences. The input link was created as specified in [2], while the observation for a particular entry was stored in its list. We accumulated the actual numbers in the table of input versus observation and prior two versus current observation. We also stored a running total of all entries in each table.

Testing was accomplished by implementing the Viterbi algorithm on the test file, utilizing the

two tables created in testing. The algorithm was run on the input vocabulary, reading from the observation table, versus the previous two guesses of observations, read from the prior table. The maximum value was then stored with the appropriate guess.

Our program considered utilizing Good-Turing smoothing on the data set, with a threshold of five, as recommended. We found, however, that none of the training data entries contained values lower than the threshold, and smoothing was abandoned in an attempt to save time. The choice to not utilize smoothing also differs from [2] where, for the full clause detection task, "...the smoothed model guarantees complete coverage of the language..."[2].

### Results

All training, testing, and evaluation files were downloaded from [1]. First, the baseline script was run against the training and testing files for the task type (start, end, or full clause detection), then our software was run on the same two file.

Table 1 shows the baseline results for the competition and the results for our implementation of [2] for the start of clause identification task. Under each file, the first number in the source program's row is the precision of that program, while the second number is the recall for that program.

Source Program	testa1.txt		testb1.txt	
Baseline	96.32%	38.08%	98.44%	36.58%
JB	93.31%	42.48%	56.22%	42.23%

Table 1: Results for Start Clause detection

### Discussion

For the start task, we noticed that our program suffered when trying to identify the starts of certain embedded clauses. One of the identifiable troubles our program had was discovering a sentence that started with an interjection (q.v. under, in, around, etc.). Almost all such sentences discovered in casual examination of the test files were incorrectly classified. Our program also had trouble discovering certain types of dependent clauses. We found many of these types of sentences in the "testb1.txt" file provided by [1]. It is also possible that our failure to implement smoothing algorithms played a role in the second test set's precision, in that the possibility exists that the program encountered information for the first time that it had never seen before, and that could have been smoothed after the training had completed. What we found encouraging about Table 1 came from the recall rates being higher than the baseline rates. The website published only the results for the full clause detection task, so we couldn't directly compare our results to that of the original paper, as our program was incapable of performing the third task, which we are about to discuss.

Our results for end clause detection task all turned out to be zero. Casual inspection of our results found that none of the end clause classifications were found. We then inspected the values in our links and tables, and found that the prior table was dominated by the entries for (X | X,X) to such a degree that class X would always be predicted by the Viterbi Algorithm, as opposed to class E, the desired prediction for the end of a clause. It was at this point that we noticed an inherent error with our data file reading functionality. Originally, we ignored the sentence separators as irrelevant and took our end of sentence marker to be the POS tag ".". We now feel that, had we allowed the blank lines to pass through our data read, we could have used these to identify the end of a sentence, allowed the blank lines to propagate into our tables, and ultimately been able to identify at least the end clauses for the sentences themselves, even if we missed some of the embedded, dependent clauses.

Our attempts at the third task revealed an even deeper rooted issue with our implementation. As stated previously, we attempted to allocate memory for the Viterbi tables in advance, after reading in the data files and creating a dynamically linked list for our input vocabulary, output classifiers, and trigram lists. Inevitably, we weren't able to create a large enough block initially, and attempted to rectify this issue by using function calls to reallocate the tables. In the end, this method led to memory boundary errors. What should have been done, given the method we used to load and store the data from the files, was to implement the tables as a linked list of linked lists, in order to easily allocate memory for tables entries we hadn't seen. Even if this implementation error hadn't occurred, we suspect that our implementation may not have fared well, due to the fact that smoothing had not been implemented.

Our original plan for selecting [2] was to find a series of improvements that would increase the system's reliability in one or more of the tasks. In the original implementation of [2], the authors created a new list of output symbols based on the POS tag and the classification. We felt this redefinition was both unnecessary and not a true representation of the classification task. We also noticed that, for the full clause detection, the authors decided to add a depth level to their newly created output classifications.

[2] further states that, when encountering more end tokens than start tokens, they would simply add start tokens at the beginning of the sentence. In situations where more start tokens than end tokens were encountered, [2] likewise add simple end tokens to the end of the sentence. Our intent to improve the system, since we maintained the same classification tokens as the original training and testing files, was to also implement a simple stack; a depth counter to determine the number of embedded, dependent clauses had been discovered. Instead of adding a start token to the beginning of the sentence when we encountered too many end tokens, our intent was to add an extra start token to the most recently discovered start token. We feel this would have made the largest improvement to the system. It should be noted that, in such an implementation, the Viterbi Algorithm's values and selections are suddenly in error and a recalculation of the probabilities would need to be made with the new information. The original implementation would require a restart from the beginning of the sentence, while our implementation would have improved based on needing to only return back to the most recent clause start.

### Conclusion

While HMM's are widely used in many natural language processing tasks, the results of [2]'s performance in [1], as well as our own implementation of [2] seem to imply that HMM's are not well suited to the task of clause identification. Given the vast amount of information provided to detect clauses, it would seem that a probabilistic approach to detecting clauses compounds the issue of detecting clauses, given that the taggers used on the sentences of the training and testing files generated these tags using the same or similar techniques. Furthermore, since the classifications were redefined by [2], rather than taking the classifications exactly as they were given, the results found during [1] by [2] also seem to be questionable. If HMM's are to be undertaken to explore this task, the methodology of [2] must be re-examined and implemented in an alternate fashion. We feel our work is an excellent start to re-examining [2]'s methodology and shows promise to improve the results of the task.

## References

- [1] “CoNLL-2001 Clause Identification Shared Task”, <http://www.cnts.ua.ac.be/conll2001/clauses/>, 8 May, 2005
- [2] “Clause Detection Using HMM”, Molina, Antonio and Ferran Pla, Proceedings of CoNLL-2001, Department de Sistemes Informàtic i Computació, Universitat Politècnica de València (Spain), Toulouse, France, 2001