Part Of Speech Tagging Using A Hybrid System

Sean Finney
Swarthmore College
finney@cs.swarthmore.edu

Mark Angelillo
Swarthmore College
mark@cs.swarthmore.edu

Abstract

A procedure is proposed for tagging part of speech using a hybrid system that consists of a statistical based rule finder and a genetic algorithm which decides how to use those rules. This procedure will try to improve upon an already very good method of part of speech tagging.

1 Introduction

The tagging of corpora is an important issue that has been addressed frequently in computational linguistics for different types of tags. Transformation-Based Learning (Brill, 1995) is a successful statistical method of tagging a corpus. It has been useful in part of speech tagging, word sense disambiguation, and parsing among other things. This paper describes a hybrid method for tagging part of speech. The method employs an implementation of Brill's Transformation-Based Learning (TBL) as the statistical part of the system, and a Genetic Algorithm which tries to modify the transformation ruleset in a way that will produce better results.

Part of speech taggers are very useful in modern Natural Language Processing, and have potential applications in machine translation, speech recognition, and information retrieval. They are usually used as a first step on a block of text, producing a tagged corpus that can then be worked with. Of course, the better the tagger is, the more accurate overall results will be. While TBL is a very successful part of speech tagger, it does still create some errors. It might be possible to tag a corpus perfectly, and it might be possible to use TBL, slightly modified, to acheive that goal.

During Transformation-Based Learning, transformation rules are learned which will correct errors in an incorrectly tagged corpus. The incorrectly tagged corpus is compared to the truth in order to learn these rules. Once the rules are learned, they are applied in an order determined by a greedy search. The rules are applied to the corpus to see how many errors they correct. The rule that corrects the most errors is chosen as the next rule in the ruleset. This process is inherently shortsighted.

cond 1	transform x
cond 2	transform y
cond 3	transform x
cond 1	transform z
cond 4	transform w
cond 5	transform x
cond 3	transform z
cond 2	transform x

Figure 1: A Sample Ruleset for TBL

The issue here is the greedy search. As rules are applied, the number of errors corrected in the text is always the largest. However, the transformation rules do tend to create errors, even as they are fixing errors. This process assumes that those errors are unimportant, or that they will be corrected later on in the ruleset. This assumption is made by TBL, and we are hoping to fix this potential problem. With a careful reordering of the rules, it might be possible to create a tagged corpus without any errors.

2 Related Work

Other workers in the field have produced non-statistical systems to solve common NLP problems. There appears

Appeared in: *Proceedings of the Class of 2003 Senior Conference*, pages 23–29
Computer Science Department, Swarthmore College

to be a number of methods that produce varying results. The Net-Tagger system (Schmid, 1994) used a neural network as its backbone, and was able to perform as well as a trigram-based tagger.

A group working on a hybrid neural network tagger for Thai (et al., 2000) was able to get very good results on a relatively small corpus. They explain that while it is easy to find corpora for English, languages like Thai are less likely to have corpora on the order of 1,000,000 words. The hybrid rule-based and neural network system they used was able to reach an accuracy of 95.5(22,311 ambiguous word) corpus. It seems that the strengths of rule-based statistical and linguistic methods work to counteract the inadequacies of a system like a neural network, and vice versa. It seems logical to combine the learning capabilities of an AI system with the context based rule creation capabilities of a statistical system.

3 Genetic Algorithms

The Genetic Algorithm (GA) takes its cue directly from modern genetic theory. This method entails taking a problem and representing it as a set of individual chromosomes in a population. Each chromosome is a string of bits. The GA will go through many generations of chromosomes. One generation consists of a fitness selection to decide which chromosomes can reproduce, reproduction itself, and finally a chance for post reproduction defects. The GA generates successive populations of chromosomes by taking two parent chromosomes selected by a monte carlo approach and applying the basic genetic reproduction function to them. The chromosomes are ranked by a fitness function specific to the problem. The monte carlo approach assures that the two best chromosomes are not always chosen.

The reproduction function is *crossover*, in which two chromosome parents are sliced into two pieces (See Figure 5 in the appendix). The pieces each take one of the pieces from the other parent. The resulting two chromosome children have part of each of the parents[1].

After reproduction, there is a chance that *mutation* will occur, in which some part of the child chromosome is randomly changed (see Figure 6 in the appendix).

When selection, reproduction, and mutation have finished, the generation is completed and the process starts again. The genetic algorithm is a good way to try many probable solutions to a problem. After a few generations of a genetic search, it is likely that the solution generated will already be very good.

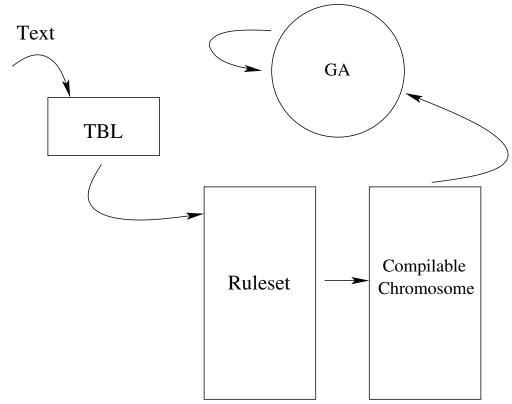

Figure 2: A Sample Flowchart for Our System

4 Representing the Problem

Given that we are trying to produce the optimal set of rules, and the order in which to use them, our chromosomes are best represented as a set of rules in order. The first parent from which all successive generations grow is collected directly from TBL. We let TBL find the rules that it finds useful, trim that set of rules to a satisfactory length, and run that through as the first parent of our GA.

One issue that we ran across was how we would represent each TBL rule as a bitstring. A rule in TBL consists of a series of predicates acting on the words before and after the word in question, and a trigger that changes the tagged part of speech on the word if all of the predicates are met. A predicate can be either a matching of a part of speech, or one of a number of string matching rules. The first of these is a pre or postfix matcher. The second adds a pre or postfix and checks to see if the resulting string is a separate word. The third removes a pre or postfix, and checks if the resulting string is a word. The fourth checks to see if a certain string is contained within the word. The fifth checks if the word in question appears to the left or right of a specified word in a long list of bigrams.

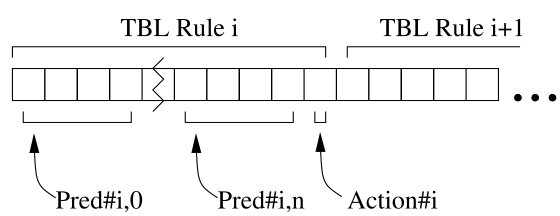

Figure 3: A sample TBL ruleset embbedded in a chromosome

In our actual implementation, every C structure con-

[1]an exception is where the crossover consists of an entire chromosome from one parent and none from the other

Appeared in: *Proceedings of the Class of 2003 Senior Conference*, pages 23–29
Computer Science Department, Swarthmore College

index	lexical meaning	contextual meaning
1	type of predicate	type of predicate
2	extra info for predicate type	whether or not the predicate is a range
3	not used	if a range, the start and stopping bounds
4	index into a table of observed strings/POS's	index into a table of observed strings/POS's

Table 1: Meaning of chromosome values to a predicate

tained member variables that could be easily converted to and from a genetic bitstring. In our population of individual chromosomes, each chromosome represented a lexical and contextual ruleset, such that any individual could be output via a helper function into the files necessary for fnTBL to run.

Each chromosome is first divided in half, with one half representing the lexical portion (prefixes, suffixes, et c.) of the fnTBL tagging rules, and the other representing the contextual (ngram) portion of the rules.

Both of these halves are then divided into several smaller parts of equal length, each part representing a single rule (see Figure 3). If the reader will recall, a rule consists of a series of predicates and a resulting action to execute if the predicates are all matching. The action is an integer which is computed via a modulo operation against the number of possible actions, such that it will always evaluate to the value of an enumerated part of speech.

A predicate is in fact one of two kinds of data structures, depending on whether it is found in the lexical or in the contextual portion of the gene sequence. They are of the same length (16 bytes), but but each 4 byte sequence may have a different meaning (See Table 1).

Our genetic "fitness" heuristic takes a chromosome, and converts it into a structure that conforms to the heirarchy discussed above. this structure is then used to output the necessary data files, which are then used to determine the error rate, which is then returned as the fitness to the genetic algorithm.

5 The Process

Our GA process works in series with TBL. We start with a corpus which gets passed through TBL, but only far enough to output the ruleset to a file. We then run a script that generates a compilable chromosome representation from the TBL ruleset file. That chromosome representation is then passed to a GA program written in C by Lisa Meeden of the Swarthmore College Computer Science Department, and severely hacked by us. In the original implementation, a chromome "bitstring" was an array of integers, where each integer was set to either 1 or 0. The author hopes that the reader can understand how this will become prohibitively large for any kind of complex representation. We therefore modified Meeden's code to have each integer be a random value, and therefore saved the memory resources required by this program by a factor of 32^2.

The GA then loops on itself for as many generations as we deem necessary. The final ruleset chromosome is then converted back into a file in the form that TBL likes, and the results are also written to a file.

The TBL implementation we used is Florian and Ngai's fnTBL(Florian and Ngai, 2001). We worked hard to integrate our scripts and code with their implementation of TBL, which trying to keep their code as intact as possible.

As it is implemented, fnTBL has two parts, the first being a script that trains the ruleset, and the second being the ruleset applicator. It was fortunate for us that fnTBL was broken up in this way, because we were able to take the learned ruleset and run it through our GA, producing another ruleset which could then be given to the fnTBL rule applicator. The results are placed in another file which can then be tested for accuracy.

6 Annealing the Mutation Rate

In order to make sure that we would get enough variation in our populations , we used a bit of the methods of annealing on our mutation rate. For the first few generations of running, the mutation rate is very high so that the children will show a lot of difference from the parents. Over time, we bring this mutation rate down so we converge on a good solution without jumping around as much. We feel that this method further reduces the risk of the solution being the greedy path to the goal.

One potential point of contention arises here. It is impossible for us to guess what the best place to begin our mutation rate is. We also need to pick the decay function, and we cannot choose whether a linear or logarithmic or exponential decay function would be better without trying them all first. We did speculate that the rate should start high and follow an exponential decay function. We want the chaos in the system to start high and come down fast, so the system has a lot of time to work out all of the chaos that was introduced in the first few generations of the GA. We chose to try a linear decay first, with a high starting mutation rate. The idea here is that much chaos over a long period of time will produce a ruleset that is radically different from the parent.

Appeared in: *Proceedings of the Class of 2003 Senior Conference*, pages 23–29
Computer Science Department, Swarthmore College

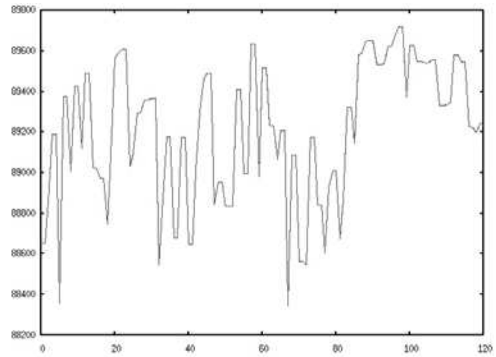

Figure 4: If only we had a month of cpu time...

7 Results and Discussion

After 100 generations without an annealed mutation rate, we had a success rate of 84.45%. When compared to fnTBL's 94.57% success rate, we actually took a loss on the performance of the system. This does make sense, seeing as how the mutation rate was very low, about 0.001. Essentially the only process being run was crossover, and when a set of rules is crossed over, more that likely the better rules at the beginning of the ruleset will be moved to the end. It certainly does not make sense to run the less successful rules first.

We experienced much better results once we annealed the mutation rate and ran the GA for 100 generations. Our success rate on this run was 89.96better results. One problem with this approach was the way the mutation rate was annealed, a strictly linear decay. We hope with an exponential decay rate, our results will be even better.

Another run with an annealed mutation rate used a starting mutation rate of .1 instead of the .8 used in the result above. 100 generations produced a success rate of 89.7even reached the level acheived already by TBL. The inherent nature of the GA approach is to need many generations, and a large population size. Unfortuantely, these runs both took close to 8 hours to complete. We believe that when taken further, and with more time and generations, the hybrid approach could allow for a continued increase across the board in NLP problems. With our minimal settings, the system took 8 hours to run. Increasing the population size and the generations would increase the running time significantly on today's computers. We hope that some day this work can be tested on a parallel or distributed system powerful enough to handle the large memory and processor requirements.

[2]there are 32 bits in an integer, so to represent an integer as a "bitstring" in Meeden's code would require 32 real integers!

8 Future Work

The original goal of the project was to try to improve TBL using artificially intelligent techniques. We feel that we have adhered to that goal. We are of the mind that language and intelligence are very closely related, and that in order for us to create systems that can use language effectively, we will first need to have a better model of intelligence. While the tools we have today in the AI field are not powerful enough to be considered fully intelligent, they can be used to approximate what we might be able to accomplish in years to come.

Work from others such as (Yang, 1999) tries to concentrate on the modelling of development, and argues that a child's developing language pattern cannot be described by any adult's grammar. Yang also states that development is a gradual process, and cannot be described by any abrupt or binary changes. The same sort of developmental thinking is encompassed by the AI approach, with a gradual (albeit hopefully faster) learning process based on human intelligence itself.

As far as our project goes, there are many variables that we did not have the time to explore. Firstly, our fitness function for the GA was simply based upon the percentage correct. Another possible idea would be to have a fitness function which takes the number of errors, either present or created by the ruleset, into account. One way to do this would be to return a fitness of the number of correctly tagged words minus a fraction of the errors. This would discourage the GA from having and causing errors.

There are also variables associated with TBL for which we chose values that made sense to us. The number of rules in a ruleset and the number of predicates in a rule could both be modified by a weight learning program like a neural network which would watch over the entire process. This way, we would have a more focused idea of what the optimal size of these variables would be.

9 Acknowledgements

We would like to acknowledge the efforts of our professor Rich Wicentowski of the Swarthmore College CS Dept. His encouragement and feedback were invaluable to our project.

Thanks also go out to Lisa Meeden of the Swarthmore College CS Dept. for her genetic algorithm code. Our task would have been much more complicated had we needed to impliment a genetic search ourselves.

We would also like to extend our gratitude to Florian and Ngai for their contribution to our project with fnTBL. Without regard to the countless days, nights, and following mornings spent trying to get fnTBL to work for us, the task at hand would have been prohibitively more complicated had we needed to implement TBL ourselves.

Appeared in: *Proceedings of the Class of 2003 Senior Conference*, pages 23–29
Computer Science Department, Swarthmore College

Thanks!

Finally we would like to thank the entire robot lab crew for being friends comrades, and cohorts throughout the entire process. Konane!

References

Eric Brill. 1995. *Transformation-based Error-driven Learning and Natural Language Processing: A Case Study in Part of Speech Tagging*.

Qing Ma et al. 2000. *Hybrid Neuro and Rule-Based Part of Speech Taggers*. Communication Research Laboratory, Iwaoka, Japan.

Radu Florian and Grace Ngai. 2001. *Fast Transofmration-Based Learning Toolkit*. Johns Hopkins University.

Helmut Schmid. 1994. *Part of Speech Tagging with Neural Networks*. Institute for Computational Linguistics, Stuttgart, Germany.

Charles D. Yang. 1999. *A Selectionist Theory of Language Acquisition*. Massachusetts Institute of Technology, Cambridge, MA.

Appeared in: *Proceedings of the Class of 2003 Senior Conference*, pages 23–29
Computer Science Department, Swarthmore College

A Appendix

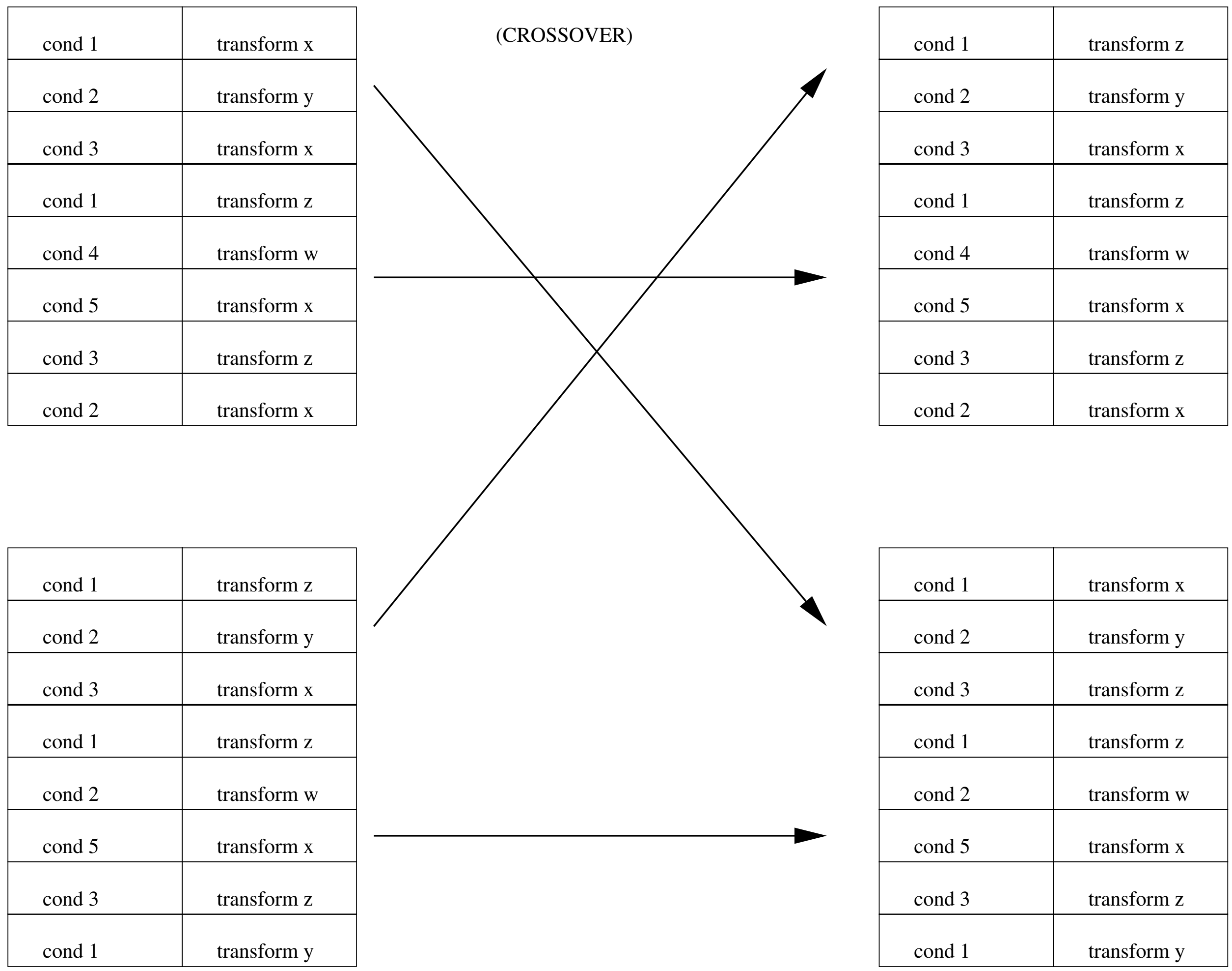

Figure 5: An example of a crossover operation

Appeared in: *Proceedings of the Class of 2003 Senior Conference*, pages 23–29
Computer Science Department, Swarthmore College

cond 1	transform x
cond 2	transform y
cond 3	transform x
cond 1	transform z
cond 4	transform w
cond 5	transform x
cond 3	transform z
cond 2	transform x

(MUTATION) →

cond 1	**transform z**
cond 2	transform y
cond 3	transform x
cond 1	transform z
cond 2	transform w
cond 5	transform x
cond 3	transform z
cond 1	**transform y**

Figure 6: An example of a mutation operation

Appeared in: *Proceedings of the Class of 2003 Senior Conference*, pages 23–29
Computer Science Department, Swarthmore College