‘These go to eleven’
Investigations in tuning the Stanford Statistical Parser

Ciaran O’Reilly

Introduction

The Stanford Statistical Parser has a long history and has been instrumental in demonstrating what can be achieved in the field of Natural Language Processing (NLP) using various computational techniques [1, 2]. This report details the results of an investigation into tuning the evaluation and runtime performance of this parser. Several aspects were explored, including smoothing rule probabilities/counts, using clusters to subcategorize parts of speech (POS) tags, and possible optimizations to the baseline exhaustive PCFG parser. While no improvements in runtime speeds were made, minor improvements in evaluation performance were obtained.

Experimental Setup

To establish baselines from which to compare any improvements in evaluation performance, six training/test runs were generated based on those detailed in the ACL03 paper [1]. These were chosen based on their performance characteristics and differences in state spaces in order to help validate the generality of any enhancements. These include four from the set of Vertical (v) and Horizontal (h) Markovization runs, which include [v=1, h=∞], [v=1, h=0], [v=2, h=2], and [v=3, h=2]. The final two were established using the final runs described in the ACL03 paper using the development (dev) and final test (test) sets. Training and testing were performed using the same portions of the WSJ Penn Treebank. Sections 2-21 for training, the first 20 files from section 22 for the development test set and section 23 for the final test. All testing was performed against sentences with length <= 40. Table 1 details the F1 scores and number of non-terminals/states generated for each trained grammar with corresponding delta differences indicating where they differ from those reported in the ACL03 paper (see Appendix A for details of the commands used for each run).

BASELINES	F1	#NonTerminals
[v=1h=Inf.]	(-0.26) 72.36	(+3427) 13084
[v=1,h=0]	(+0.34) 71.61	(+422) 1276
[v=2,h=2]	(-0.64) 76.86	(+3107) 15239
[v=3,h=2]	(-0.15) 78.92	(+4357) 27243
[acl03dev]	(-1.12) 85.92	(N/A) 18278
[acl03test]	(-0.77) 85.55	(N/A) 18278

Table 1 Baselines Collected versus ACL03 reported (delta shown)

Excluding the differences in F1 scores, the large variances in the number of non-terminals generated for each run is troublesome. This is likely due to choosing incompatible parameter settings. While a couple

dozen different combinations were tried, there are currently forty five different training options alone to choose from, which emphasizes the parser's current level of sophistication/complexity. In particular the '–compactGrammar = 0' was used in order to be able to work with non-normalized log probabilities when performing smoothing of rule probability experiments at testing time but this alone does not account for the differences. However, for the purposes of this exploratory study the baselines collected are considered sufficient.

Smoothing

The Stanford Parser uses Maximum Likelihood Estimates (MLE) for calculating the probabilities of grammar rules. As noted in [3], there are two main sources of estimation error, bias which for MLE is 0 and sampling error, which due to the sparse data problem caused by the significant number of grammar rules in PCFGs can be very large. Therefore, it is generally accepted that introducing some form of smoothing can help improve the generality of MLE estimates and in this case parsing performance. Due to the large number of possibly unseen rules two separate investigations into smoothing were performed, the first on observed rules only and then on a mechanism for handling unseen rules.

Observed

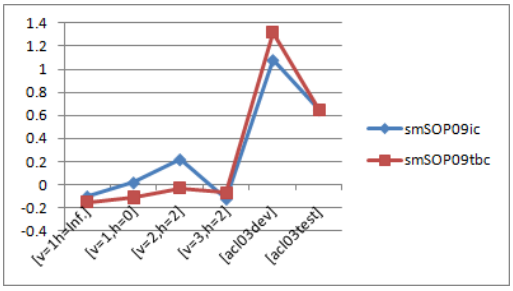

Figure 1 F1 Delta Scores for smoothing

Of relevant interest is the work of S. O. Petrov [4] which improves upon the ACL03 reported scores by approximately 3% in overall performance by constructing a compact latent variable grammar using machine learning techniques. In particular he notes as regards using merging to avoid over fitting that 'smoothing the productions of each subcategory by shrinking them towards their common base category gives us a more reliable estimate' and is what is used here. This is performed in a linear way using the following equation:

$$p_x^{'} = (1-\alpha)p_x + \alpha\bar{p}, \;\; where\, \bar{p} = \frac{1}{n}\sum_x p_x$$

with $p_x = P(A_x \rightarrow B_y C_z)$ and the same value of 0.01 for α used. Only phrasal categories are smoothed in this way as tagging probabilities are already smoothed in order to handle unseen words. For comparison purposes a variant of this smoothing was done within each category (smSOP09ic) as opposed to towards their common base category (smSOP90tbc) in order to test the idea of sharing statistical strength. Figure 1 details the results of applying both of these to the baseline test sets. As can

be seen, smoothing is counterproductive with the simple Markovization baselines but turns beneficial when run with the ACL03 baselines (+1.32% on the dev set and +0.65% on the final test set). This corresponds to the expectation and observations in [4] that smoothing works best when there are large numbers of subcategories, which is when production statistics are most unreliable due to sampling error. In this case it is due to the fact that the ACL03 baselines are using parent annotations on their tags (this is not done for the basic Markovization baselines), which increases the number of subcategories at the tag level significantly (from 45 Penn Treebank POS tags to 559 with parent annotations and other linguistically motivated splits applied during training). In addition, figure 1 also indicates that sharing statistical strength is beneficial at least in the development test set case but no difference was observed in the final test set. Additional analysis is required to concretely identify the benefit of smoothing towards common base categories but it appears effective. One advantage of applying this form of smoothing is that it can be applied at parse/testing time only (Note: a normalization step is required to ensure conditional probabilities are correct after smoothing to base categories is performed) and except for an upfront re-initialization step of rule probabilities, adds no additional performance overhead incurred through the introduction of new rules to handle the unseen case.

Unseen

More generally, smoothing is intended to help handle the unseen case. With parsing, this consists of handling unseen words and unseen grammar rules. Unseen words are already handled by the Stanford Parser. In the case of grammar rules, PCFGs due to there usually large number of non-terminals, will have a huge space of possible rules that could be applicable (i.e. $O(N^3)$ where N is the number of non-terminals) and most will not have been seen in the training data. To get an idea of the number of unseen rule instances to be expected when parsing sentences after training is complete a simple analysis using just the training set was performed. Table 2 shows the number of raw Treebank rule instances not seen when different percentages of the training set data are withheld. The percentages used for the first two correspond respectively to the proportions of development and final test set sentences used for standard evaluation purposes. All non-terminals were seen in each case.

Withheld	#Rule Instances	#Types Unseen	#Instances Unseen	%Unseen Instances
0.977%	8149	97	99	1.214%
5.330%	41849	585	624	1.491%
10.000%	77691	1079	1154	1.485%

Table 2 Approximates of Unseen Raw Treebank Rules

For common evaluation scenarios, this indicates the possibility of approximately 1.4% of the grammar rules comprising valid parse trees not having been seen in training. As these can have cascade affects it makes sense to try and handle this case.

However, considering all possible rules is unrealistic from a space and time complexity perspective. Instead, it makes sense to try and predict a small subset of unseen rules based on what has been observed in training. For this we use the intuition behind Kneser-Ney discounting as described in [5],

where instead we use the counts of child nodes within head nodes as a simple predictor for unseen rules:

$$P_{rulecont\ inuation}\ (A_x \rightarrow B_y\ C_z) = \frac{C(A \rightarrow B_y\ C_z)}{C(A \rightarrow B\ C)}$$

We use this then to generate predicted unseen rules for each non-terminal already containing a unary or binary rule (one predicted for each type that exists) in addition to applying absolute discounting. Parent annotations complicate determining meaningful subsets of the head of a rule and were not addressed as part of this exploratory study. In practice this is overly simplistic as only a very small number of rule types get generated, less than a dozen, as possible predictors for rule types across all non-terminals with observed rules. Likely additional base/sub-categorization information should be used in the prediction. When applied to the [v=1, h=∞], [v=1, h=0] runs, improvements in F1 scores of 0.66% and 0.01% were obtained. However, as the number of rules is increased this also impacts parsing performance, significantly by up to an order of magnitude in the [v=1, h=∞] case due to an almost doubling in the number of rules.

Optimization

The current implementation of the Exhaustive PCFG parser is highly optimized from an execution standpoint, using several techniques including:

- **Efficient Rule Indexing/Comparison:** All rules are indexed using integers and the rules themselves can be compared using these indexes for their heads and children.
- **Log Arithmetic:** All calculations are done using simple addition instead of multiplication, which is approximately 25% faster on standard Java implementations.
- **Unary Rule Closures:** A naive implementation of the generalized probabilistic CKY algorithm requires all the unary rules to be rechecked if one of them is activated, causing a possibly large number of iterations over these rules on each pass through the inner loop. Instead, unary rule closures are calculated up front, and while increasing the number of rules to be checked, it requires they only be checked once each time through the loop.
- **Extracting Best Parse:** Instead of maintaining a set of back pointers in order to obtain the best parse once the CKY algorithm is complete, it is derived from the generated table of scores. This saves space and increases performance significantly as it cuts in half the number of array look up and assignments needed in the inner loop of the algorithm.
- **Skipping Invalid Constituents:** Rules that should not be considered due to generating invalid constituents are skipped efficiently, which decreases significantly the number of index lookups and score calculations/comparisons required.

The majority of the time spent parsing sentences by the Exhaustive PCFG parser is within the inner loops of its probabilistic CKY algorithm. With this in mind, two experimental versions of the parser were developed and tested to see if they could improve parse times. The first looked at the possibility of using an Integer representation of the scores, which currently use floats, to track log probabilities. The

reasoning for this is that integer arithmetic is approximately 30% faster than floating point arithmetic. Integers on their own are insufficient for representing the range of probabilities calculated for reasonably sized sentences but if impactful on performance a related bit based representation could possibly be developed. In practice this only improved performance by approximately 5% and this gain would likely be lost when taking into account a modified data format capable of accurately representing the range of values already supported by the float format. The second approach looked at multithreading the inner loop of the algorithm by segmenting what the threads work on based on the non-terminal state space. For a simple threaded implementation, that creates threads on each loop, on a dual core machine this gives an approximate speed up of 10%. This should improve linearly, to a point, with more available processors and a smarter semaphore based implementation that only requires the threads be initialized once. However, testing showed up an invalid assumption about the state spaces of the individual threads being separate when performing their calculations. The data structures required to support the 'Skipping Invalid Constituents' cause the threads to overlap and corrupt each others' data, resulting in inaccurate calculations. It is believed a significant number of parallel threads would be required to gain the same performance increase obtained from using the 'Skipping Invalid Constituents' approach and therefore until a large number of cores are readily available as standard, a parallelized version of this portion of the algorithm is considered impractical.

Clustering POS Tags

Considering the performance levels and approaches detailed in [4] a possible mechanism for improving the Stanford Parser would be to subcategorize POS tags using a syntactic distributional similarity method [6]. For this the system developed by A. Clark [7, 8] was used. To induce the syntactic clusters the training data sentences were extracted and formatted to the input specifications of the provided program. This consists of capitalizing each word and placing in a file with one word per line and empty lines used to indicate sentence ends. Ten generation runs were executed, which created separate clusters of the order 2, 3, 4, 5, 10, 25, 45, 65, 90, and 120 (see Appendix A for details of options used). These clusters were then used on separate training and test runs to subcategorize the POS tags normally used for a particular run.

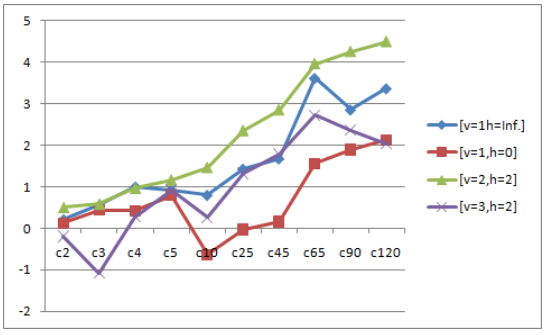

Figure 2 F1 Delta scores for basic Markovization runs using clusters

Figure 2 details the results of using the different sized clusters in conjunction with the four basic Markovization baselines. The [v=2, h=2] run is the best, scoring an improved F1 of 4.49% which is

encouraging and indicates strongly that the underlying clusters do represent a meaningful sub-partitioning of the tag set. Of interest is the performance of [v=3, h=2] which improves till cluster size 65 and then deteriorates in performance. This aligns with the observation made in [1] as regards this run having a large number of states and not tolerating further splitting well.

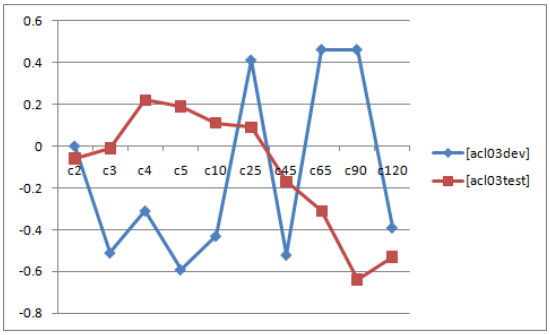

Figure 3 F1 Delta scores for ACL03 dev and test runs using clusters

However, these improvements are not present in the ACL03 dev and test runs that are supplemented with clustering. Figure 3 shows that clustering adds little or negatively depending on the number used and no obvious discernable pattern is present. The reason this is believed to be occurring is due to the number of POS tags used to begin with. Figure 4 details the number of subcategories that get added for each kind of run that are supplemented with clusters. For the four basic Markovization runs they start with the 45 basic Penn Treebank POS tags (no-tagPA). However, the ACL03 dev and test runs start with 559 POS tags (tagPA) due to the use of parent annotations and linguistically motivated splits (e.g. the 'IN' POS tag is broken down into 6 subcategories). Two main factors could be coming into play, firstly a large amount of splitting is occurring at the pre-terminal nodes leading to the sparse data problem. Secondly, the parent annotations may not be aligning well with the clusters causing them to work poorly together. Each of these is explored, initially by classifying the clusters into open and closed sets followed by determining the effects of removing parent annotations when using clusters.

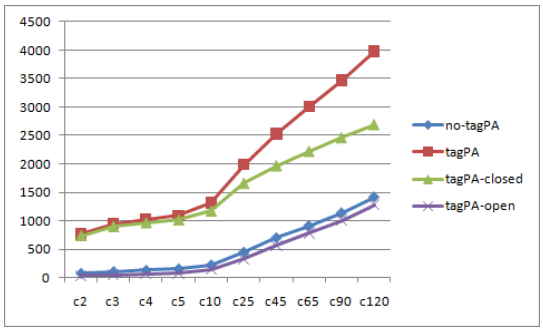

Figure 4 # Subcategories added to POS tags

Segmenting Clusters

For the base ACL03 runs, 17 of its 559 POS tags are considered open. In order to determine whether or not the clustering between open and closed classes were interfering with each other directly, an

additional 4 runs for each cluster were performed with clustering turned off for specific POS tags depending on whether or not they belonged to the closed or open set.

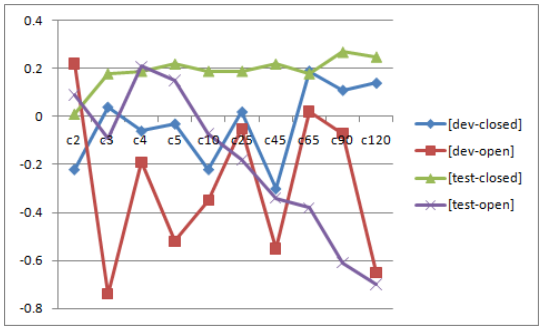

Figure 5 F1 Delta scores based on using clusters on open or closed POS tags

Similar to Figure 3, Figure 5 shows little indication that this is the case. Of interest is the stability of the [test-closed] run which remains fairly consistent across all runs above 2 clusters. Further investigations are required to help identify the reason for this.

Dropping TAG-PA

Figure 6 shows the results of dropping parent annotations in favor of just the clusters themselves. The linguistically motivated tag splits, e.g. the 6 way on 'IN', were left in place and used instead of their cluster counterparts. As can be seen, both sets of runs do not perform as well as their merged parent annotation and cluster counterparts.

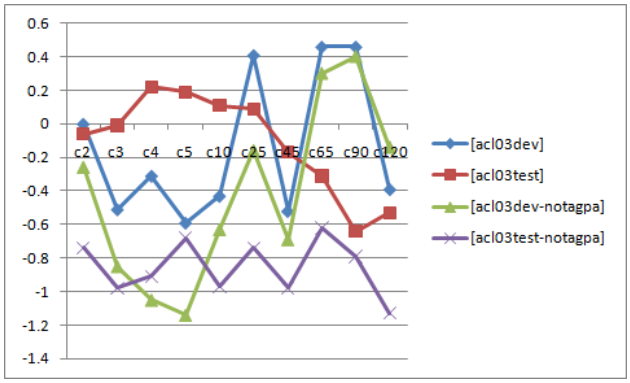

Figure 6 Delta scores for ACL03 dev and test runs using clusters along with no Tag PA

This indicates that it is less of a case of parent annotations and clusters not co-existing well together but likely that there is more distinguishing information available using parent annotations than the clusters used.

Smoothing Clusters

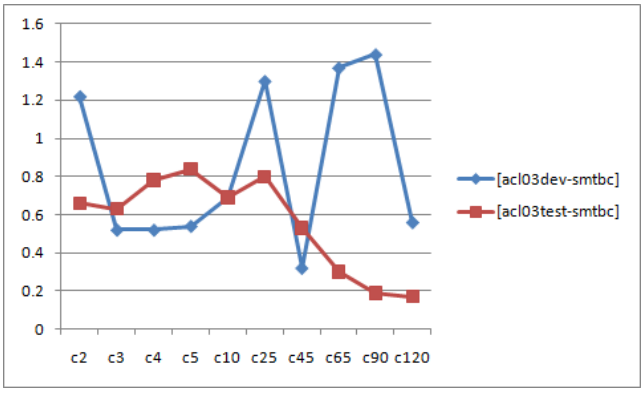

Figure 7 Delta scores for ACL03 dev and test runs using clusters and smoothing

A final run, see Figure 7, of the ACL03 development and tests sets were performed using each of the clusters in conjunction with smoothing to the base category. This gave the best improvements in performance overall, with the development set scoring 1.44% over the baseline and the final test set 0.84%.

Discussion and Future Work

Considering the three main investigations undertaken as part of this exploratory study, there are some areas that appear to be effectively handled already while others require further investigation. In particular, the Stanford Parser is already highly optimized and it is hard to see where additional meaningful improvements in its already good parsing performance could be made. As regards smoothing, more work is required to better handle the unseen case. Even based on the already high performance of Statistical Parsers this still seems like an area where some additional gain could still be made through better prediction of unseen grammar rules. Clustering POS tags looked promising initially but it is unclear if there are good additional gains to be made from their use. One thing to try would be to test each POS cluster independently in order to see if any of them alone add a lot of value. In the same way as linguistically motivated splits demonstrated meaningful improvements this may identify useful automatically identified ones. Though requiring a fair amount of processing time, this would be relatively straight forward to test.

Conclusion

The intent of this project was to dive deep into an interesting piece of technology in order to gain a better understanding of statistical NLP, which has been the case. When a particular level of performance is reached it is very challenging to try and determine where the next set of improvements can be made. Those identified here are only minor and in practice are likely just as easily achieved through using different training configuration settings in the already wide range currently available in the parser. In essence, trying to identify enhancements that co-exist well together is very difficult. On a lighter note,

the author is happy to see that current statistical parsing technology is at the level it is, because if asked to parse a couple thousand sentences he is pretty sure he would not do so well.

References

[1] D. Klein and C. D. Manning. Accurate Unlexicalized Parsing. In Proceedings of the 41st Annual Meeting on Association for Computational Linguistics, pages 423-430. Association for Computational Linguistics, 2003.

[2] D. Klein and C. D. Manning. Fast Exact Inference with a Factored Model for Natural Language Processing. Advances in Neural Information Processing Systems, Vol. 15, pages 3-10. MIT Press, 2003.

[3] M. Collins. Head-Driven Statistical Models for Natural Language Processing. PhD thesis, University of Pennsylvania, 1999.

[4] S. O. Petrov. Coarse-to-Fine Natural Language Processing. PhD thesis, University of California Berkeley, 2009.

[5] D. Jurafsky and J. H. Martin. Speech and Language Processing (2nd ed.). Chapter 4 page-110. Pearson Education Inc. 2009.

[6] C. D. Manning and C. O'Reilly. Personal Communications. May 16th 2010.

[7] A. Clark. Inducing Syntactic Categories by Context Distribution Clustering. Proceedings of CoNLL-2000 and LLL-2000, page 91-94. 2000.

[8] A. Clark. Combining Distributional and Morphological Information for Part of Speech Induction. Proceedings of the 10th EACL, pages 59-66. 2003.

Appendix A – Commands Used

Training and Testing the Stanford Statistical Parser

Version 1.6.2 of the Stanford Statistical Parser was used for each training/test run:

[v=1, h=∞],[v=1, h=0], [v=2, h=2], [v=3, h=2]

LexicalizedParser invoked with arguments: -v -evals factDA,tsv -PCFG -saveToSerializedFile parser_run_output\base_rawtb40.ser.gz -saveToTextFile parser_run_output\base_rawtb40.txt -vMarkov <#v> -hMarkov <#h, used Integer.MAX_VALUE for infinity.> -compactGrammar 0 -maxLength 40 -flexiTag -scTags -printTT 1 -train Z:/ir.stanford.edu/data/linguistic-data/Treebank/3/parsed/mrg/wsj 200-2199 -testTreebank Z:/ir.stanford.edu/data/linguistic-data/Treebank/3/parsed/mrg/wsj 2200-2219

[acl03dev]

LexicalizedParser invoked with arguments: -v -evals factDA,tsv -saveToSerializedFile parser_run_output\base_acl03pcfg_dev40.ser.gz -saveToTextFile parser_run_output\base_acl03pcfg_dev40.txt -compactGrammar 0 -acl03pcfg -printTT 1 -train Z:/ir.stanford.edu/data/linguistic-data/Treebank/3/parsed/mrg/wsj 200-2199 -testTreebank Z:/ir.stanford.edu/data/linguistic-data/Treebank/3/parsed/mrg/wsj 2200-2219

[acl03test]

LexicalizedParser invoked with arguments: -v -evals factDA,tsv -saveToSerializedFile parser_run_output\base_acl03pcfg_test40.ser.gz -saveToTextFile parser_run_output\base_acl03pcfg_test40.txt -compactGrammar 0 -acl03pcfg -printTT 1 -train Z:/ir.stanford.edu/data/linguistic-data/Treebank/3/parsed/mrg/wsj 200-2199 -testTreebank Z:/ir.stanford.edu/data/linguistic-data/Treebank/3/parsed/mrg/wsj 2300-2399

Training POS Clusters

= number of clusters to be generated.

cluster_neyessenmorph -s5 -i10 stco.data stco.data # > stco-s5-i10.nem.#