Discriminative Reranker for Parsing with Optimal Features Algorithms for NLP 11-711: Assignment 3

Kinjal Jain[1]

***Abstract*— This paper addresses the implementation for reranking the parse trees generated from an existing Base Parser, using two classification algorithms namely Perceptron and PrimalSVM, and the various strategies and experiments conducted for feature extraction to train aforementioned algorithms on. It is shown that using certain span based and word based features, the accuracy with which the best parse tree is predicted substantially exceeds the accuracy of the Base Parser. Additionally, this Discriminative Reranker implementation is constrained by memory usage of 6GB and thus aims to use only the best features after performing multiple ablation studies.**

I. INTRODUCTION

Generative Parsers like CKY based parsers predict only one parse tree, and do not allow any further optimization in terms of getting the gold parse tree. However, they can be extended to provide top K parses which can then be consumed by Discriminative Reranking models such as those trained using Perceptron, SVM, MaxEnt algorithms to predict the best parse tree (effectively gold or closest to gold parse tree). These algorithms allow us to accommodate numerous syntactic features which might not be possible with just generative parsers. Identifying the syntactic structure of a sentence is very important in order to be able to represent it in terms of POS tags. The syntactic features help us in identifying the underlying semantics by attaching more context to the POS symbols. The features used for reranking the parse trees are presented in the Featurs section later. This paper specifically discussed two algorithms namely Perceptron and PrimalSVM.

Perceptron: It is an algorithm for learning a binary classifier (threshold function) that maps its input x(a real-valued vector) to an output value f(x)(a single binary value). Weights w are learned by the algorithm during training for each component in x.

$$\text{f(x)=}\begin{cases} 1 & \text{if } \mathbf{w}\cdot\mathbf{x}+b>0, \\ 0 & \text{otherwise} \end{cases}$$

For the purpose of reranking, Perceptron weights are used to weight each feature importance for predicting the best parse tree. The algorithm learns a linear separator, and so if the data is not linearly separable, the Perceptron will never converge. Therefore, it becomes important to use the validation data set to tune the number of epochs (iterations) that the algorithm should train for in order to achieve a minimum loss value. Previous studies highlight variants of the standard Perceptron algorithm such as Average Perceptron and Average Voted Perceptron both of which are were ecperimented with as part of the implementation. As thought of intuitively, the Average Perceptron Algorithm tends to converge faster than the standard Perceptron algorithm.

PrimalSVM: Similar to Perceptron, Support Vector Machines aim to learn the input weights such that the output value correctly indicates the class of the input, but it also extends the idea of linear separability to maximum margin separability. So, it is thought to perform better than Perceptron on unseen data. To account for the misclassified data points, it adds some penalty to the actual function. The Primal optimization objective of the SVM is:

$$\begin{aligned} &\underset{w,\xi_i}{\text{minimize}}\ \tfrac{1}{2}||w||_2^2 + C\textstyle\sum_{i=1}^{n}\xi_i \\ &\text{s.t.}\quad \text{y}_i(w^T x_i) \geq 1-\xi_i \forall i = 1,\dots,n \end{aligned}$$

The PrimalSVM implemented in this paper uses Adagrad which uses Step size, Batch size and Regularization constant C as hyperparameters. Different values for these were experimented and the results are presented in Experiments & Analysis Section. In the next section, implementations of these two algorithms are explained in detail for the objective of reranking parse trees.

II. IMPLEMENTATION DETAILS

Two Reranking Factory classes are implemented, one for each of the above algorithms.

BasicParsingFactory: This factory is used for invoking $PerceptronReranker$ and $AveragePerceptronReranker$ classes. Under $PerceptronReranker$ class, the best parse tree is predicted by maximising the following objective function:

$$\text{y'} = arg \max_{y\in Y(X)} (w^T f(y))$$

where y' is the predicted best parse tree. If y' is not correct, the weight vectors are updated by the rule:

$$\text{w} \leftarrow w + (f(y*) - f(y'))$$

where y* is gold, and y' is wrongly predicted best parse tree. Under $AveragePerceptronReranker$, updated weights vector are averaged with the previous weights vector to generate the new weights vector after every n mistakes. n is treated as a hyperparameter. The Average Perceptron Algorithm helps by not giving too much weight to only recently mistakes.

[1] Email:kinjalj@andrew.cmu.edu

AwesomeParsingFactory: This factory is used for invoking $PrimalSVMRerankerBinaryLoss$ and $PrimalSVMRerankerAbsoluteLoss$ classes. Under $PrimalSVMRerankerBinaryLoss$ class, standard PrimalSVM is implemented where loss for each tree in the ranked list is calculated to be either 0 or 1 based on whether it is the best parse or not. In $PrimalSVMRerankerAbsoluteLoss$ class, loss is calculated as *1 - F1 score*. The experiments show that the latter performs slightly better than the former.

MyFeatureExtractor: This class is used to extract all the features used by the above two algorithms. The features are implemented under the $extractFeatures()$ function and are described in detail in the next section.

III. Features

Following is an exhaustive list of features that were experimented with in order to get the best accuracy metrics:
1. Position: This feature specifies the position of the parse tree in the KBestList reported by the Base parser. It is added in the form- $\{Posn = index\}$ where $index$ is rank of that particular tree. This feature is good because it adds a relative ordering of trees to the model and essentially makes use of ranking given by Base parser.

2. POS Grammar Rules: This feature specifies the grammar rules used in the parse tree. It takes the form- $\{Rule = rule\}$ As part of experimentation, it was concatenated with other span based rules which are discussed later.

3. Span Length: This feature captures the lengths of all subtrees present in a parse tree. Since the feature is very sparse, binning is performed and binned value is added in the rule. It takes the form- $\{Rule = rule\_SpanLength = binnedSpanLength\}$. The bins are of values [1,2,3,4,5, $\leq$10, $\leq$20, >20].

4. Span Words: This feature is further divided into two features- FirstWord and LastWord which correspond to $\{Rule = rule\_FirstWord = firstWordInSpan\}$ and $\{Rule = rule\_LastWord = lastWordInSpan\}$.

5. Span Contexts: This feature is further split into two features- LeftContext and RightContext which correspond to $\{Rule = rule\_LeftContext = lastWordBeforeSpan\}$ and $\{Rule = rule\_RightContext = firstWordAfterSpan\}$.

6. Span Shape: This feature captures the shape of span in terms of punctuation, capitalization, difference between letters and digits of words in the sentence. This helps in capturing the overall structure of the sentence and identifying proper nouns from all phrases. It is specified in the form- $\{Rule = rule\_SpanShape = XxN!\}$ where captilized words are marked by X, other words by x, digits by N and other symbols and punctuation are marked by those symbols and punctuation themselves.

7. Heavyness: This feature is added based on the paper- $LessGrammar, MoreFeatures$. It captures the span length and distance of subtree from the end of the sentence. It is added in the form- $\{Heavyness = subtreeParentLabel\_spanShape\_distanceFromEnd\}$.

8. Split Rule: This feature is added for the spans which have exactly two children. It is added in the form-$\{splitRule = parentLabel(leftChildLabel...splitWord)rightChildLabel)\}$. This feature is especially helpful for disambiguating prepositional phrases which are usually followed by only a set of specific words.

9. Word Ancestors: This feature is added for the entire tree capturing n last Non terminal symbols as ancestors which have that particular word as child. It is specified in the form-$\{Ancestor = n = word\_parent\_...\}$. n=4 was identified to work the best.

10. Trigrams: This feature adds all trigrams present in all the subtrees of the parse tree. It is specified in the form-$\{Trigram = trigramLabels\_...\}$

11. Right Branching: This feature adds a lot of goodness by specifying the total number of Non Terminals present on the rightmost branch of the tree. It is specified in the form- $\{Rule = rule\_RightBranchingNT = numberOfNTOnRightMostChild\}$. Another variant of the same was also tried which specifies the number of Non Terminals not present on the rightmost branch of the tree.

12. Semantic/Syntactic Heads: This feature captures the relation of the POS tag of a subtree's parent label to its head in the subtree span. It is specified in the form- $\{Rule = rule\_Head = head\}$. They seem to aid in issues with prepositional phrase attachment.

13. Neighbours: This feature combines the information provided by span length, parent of the subtree defining the span and the POS tags of the words present in the span. It is added in the form- $\{Rule = rule\_parentLabel\_spanLength\_POSTags\}$.

14. Log Probability Score (binned): This feature helps in actually differentiating between closely scored and closely ranked parses better than just position of the tree in the list. Different bins for the score were tried and ultimately 5 groups were used.

Multiple experiments were run to identify the best set of features and the results of these ablation studies are discussed in Experiments & Analysis section.

IV. Collection Statistics

Following are the generic statistics for test and validation data sets:

Data Type	Number of Sentences	Total Number of KbestList And Gold Tree Pairs	Number of Gold Trees Present in KBestList
Valid	1578	36765	21254
Test	2245	36765	21254

These indicate that for about one-third of the data points, the gold tree is not present in the KBestList provided, and so, experimentation is done by taking treating one of the trees in the list as gold tree (using F1 evaluation for similarity) and also by using the absent gold tree itself. Both the algorithms perform better using the tree present in the KBestList which is closest to the given gold tree.

V. Experiments & Analysis

On Perceptron: Using the vanilla $PerceptronReranker$ algorithm with all the above features, the best F1 was just 85.8 on validation dataset and 85.66 on test dataset. With $SimpleFeatureExtractor$, F1 was observed to be 83.54 on validation dataset. However, with $AveragePerceptronReranker$ algorithm, a much better performance was achieved. It was also noticed that the perceptron algorithm converges (near convergence in this case) much more quickly ie. in lesser epochs when using averaging. The following table gives a trend of results observed on validation and test set for different values of $numEpochs$ and hyperparameter $mistakesCounter$. It was observed that numEpochs = 10, and $mistakesCounter$=2000 performed best by giving an F1 score of 86.4 on test data. It should be noted that only a subset of features were ultimately used to give the best F1 score, the information about which is presented in the upcoming section.

Data Type	Number Of Epochs	Number Of Mistakes To Average on	F1 Score
Valid	10	1000	86.34
Valid	15	1000	86.44
Valid	10	2000	86.61
Valid	15	2000	86.34
Test	10	1000	86.11
Test	15	1000	86.26
Test	10	2000	86.4
Test	15	2000	86.32
Test	10	3000	85.97
Test	15	2000	86.21

The Voted Average Perceptron Algorithm was also tried out, but it took a lot of time to train and didn't converge even in 60 epochs, and the updates were slow as well. Further, results were worse than Vanilla Perceptron. It could be argued that it demanded some feature pruning but due to time constraints that wasn't tried.

Feature pruning was also tried for features that don't appear on more than 5 trees but it didn't result in any performance gain and so it isn't performed in the final submission.

For weight vector initialization, two approaches were tried zero initialization, and uniform distribution based on the total size of feature vector. In the best case scenario, the results from both were same and so it can be noted that with sufficient number of epochs, the initialization value doesn't make a difference.

On PrimalSVM: The following different values of hyperparameters were tried for the two implementations of $PrimalSVMRerankerBinaryLoss$ (0/1 loss) and $PrimalSVMRerankerAbsoluteLoss$ (absolute loss):

- StepSize: [0.01, 0.001, 0.0001]
- regConstant: [0.1, 0.01, 0.001, 0.0001]
- batchSize: [128, 256, 512]
- epochs: [10, 20, 30, 50]

The best combination was identified to be StepSize: 0.01, regConstant:0.0001, batchSize:512 and epochs:30 which gives F1 of 87.23 on validation data and 86.89 on test data.

The $PrimalSVMRerankerAbsoluteLoss$ performs slightly better than $PrimalSVMRerankerBinaryLoss$ although the latter takes substantially lesser time to train.

Feature pruning was tried for features that don't appear on more than 5 trees but like in Perceptron, it didn't result in any performance gain and so it isn't performed in the final submission.

The following table marks the difference in F1 scores obtained on validation dataset by performing ablation study on the above mentioned features using $PrimalSVMRerankerAbsoluteLoss$ class. It was used to combine some rules into one rule and remove some rules altogether to achieve the best F1 score.

Features Used	Number of Features	F1 Score
Position + Rule (Baseline)	49783	84.12
+ Span Length (binned)	101233	84.79
+ Span Words	577753	85.33
+ Span Contexts	938126	85.42
+ Span Shape	1467337	85.51
+ Heavyness	1590607	86.38
+ Split Rule	1680102	86.44
+ Word Ancestors	2134110	86.66
+ Trigram (Siblings) (decreases F1)	2218903	86.59
+ Right Branching	2429177	86.89
+ Semantic Heads (decreases F1)	2915686	86.58
+ Neighbours	3004333	87.12

VI. Performance

The following are the best performance metrics reported for both the algorithms on validation and test data sets executed on MacBook Pro 1.4GHz Intel Core i5 Processor with JDK 1.8, with features resulting in decrease in F1 score removed:

Model on Validation Data	F1 Score	Decoding Time (in Millis)	Number of Features Used
Baseline	84.12	78	-
Basic	86.88	7568	1857313
Awesome	87.23	3598	1857313

Model on Test Data	F1 Score	Decoding Time (in Millis)	Number of Features Used
Baseline	83.66	57	-
Basic	86.4	8134	1857313
Awesome	86.89	5366	1857313

The results indicate that Primal SVM is able to learn better from the same set of features than the Average Perceptron algorithm. However, it is good to note that these results are empirical and based on certain heuristics with respect to feature selection.

VII. Conclusions

As part of the assignment, 2 different discriminative reranking algorithms were implemented in the form of Perceptron and PrimalSVM. Various combinations of features were tried and some of them like $Heavyness$ and $RightBranching$ seemed to increase the accuracy by a large amount, while some like Head based features didn't improve the accuracy at all. Some correlated features were also observed like $LogProbabilityScore$ and $Position$ features, where removing the latter and keeping the former increased the F1 score. Overall, PrimalSVM based model outperformed Perceptron, but with different features one cannot not guarantee that it will always be the case.

References

[1] Eugene Charniak and Mark Johnson. 2005. Coarse-tofine n-best parsing and MaxEnt discriminative reranking. In Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics, pages 173–180, Ann Arbor, Michigan, June. Association for Computational Linguistics.

[2] Michael Collins and Terry Koo. 2005. Discriminative reranking for natural language parsing. Computational Linguistics, 31(1):25–70.

[3] Mark Johnson and Ahmet Engin Ural. 2010. Reranking the Berkeley and Brown Parsers. The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics.

[4] Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In ACL, pages 423–430.

[5] David Hall, Greg Durrett, Dan Klein. 2014. Less Grammar, More Features. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)

[6] Jurafsky, Dan. "Speech language processing.“ Pearson Education India, 2000.