CS 224n Programming Assignment 3: Maximum Entropy Markov Models & Treebank Parsing

Todd Sullivan
todd.sullivan@cs.stanford.edu

Pavani Vantimitta
pavani@stanford.edu

1 Introduction

For PA3 we implemented a Maximum Entropy Classifier, which we used for a Maximum Entropy Markov Model. We developed many features for the model, of which sequence features such as previous word and next word has the most positive effects. Our best feature set achieves an accuracy score of 90.57 and overall FB1 of 67.38 on the genia data set. Our parser achieves its best F score of 84.28 on the genia data set by using third order vertical markovization. Due to our optimizations described in Section 3.1, our parser is able to parse 20-word sentences in 125 milliseconds and 69-word sentences in 4.8 seconds on average when running on a single core of an Intel Pentium Core 2 Duo T7500 @ 2.2 GHz.

2 Maximum Entropy Markov Models

In this section we discuss the implementation details related to fast training and selecting the optimal feature set from the features that we created. We end the section with a performance analysis of the positive and negative aspects of our best feature set.

2.1 Training Implementation

The training routine for Maximum Entropy models is fairly straightforward and easy to implement. With that said, there are several key design decisions that can have a large impact on performance. In this section we briefly discuss the key modifications to the base algorithm that gave us significant reductions in computation time.

The first task in implementing the Maximum Entropy Classifier was to implement the getLogProbabilities function. This function is used by many parts of the classifier and is important to implement efficiently. Thankfully, there was only one obvious way of implementing the function, which we will not bother to describe. We tried using the provided SloppyMath class to perform calculations quickly, but the class' use actually resulted in incorrect output.

The primary function where optimization can make a difference is the calculate function, which takes a weight vector as input and computes the objective function and its derivatives. The most naïve way to implement this function is to compute the objective and derivatives in two separate loops, applying the regularization at the end. This is the most straightforward way to implement the method because it is exactly how the process is described in mathematical terms.

Assuming one implements the function in this way optimally, one would include the objective and derivative penalties in a single for loop and assign variables as far outside of for loops as possible. For example, the objective penalty is $\sum_i \frac{\lambda_i^2}{2\sigma^2}$, so one would perform the computation $2\sigma^2$ once within the function and then use the value as needed. The first obvious optimization to notice here is that the value that we divide by in the object and derivative penalties, which are $2\sigma^2$ and σ^2 respectively, do not change between calls to the function. Since the calculate function is called many times, computing these values in the class' constructor results in the training being 1.08 times faster.

The second important optimization is to simply compute the objective and derivatives in one for loop. This halves the number of calls to getLogProbability, making the function 1.23 times faster than the base, naïve implementation.

The third important optimization is to notice that for each item in the training set, only one value from getLogProbabilities is used in its log form – the one that is included in the objective calculation. Yet all of the values returned by getLogProbabilities are needed in their straight probability form for the derivative calculations. Thus for our final optimization we created a copy of the getLogProbabilities function called getProbabilities, that did not compute the log of each probability before returning the array. We then take the log of the single value that is included in the objective calculation and do not have to use the Math.exp function to convert all of the log probabilities for use in the derivative calculations. Including this final optimization resulted in an implementation that was 1.75 times faster than the base, naïve implementation.

2.2 Features

In this section we describe the features that we developed, our hill-climbing method for finding the best feature set, and the importance of each feature in our final feature set.

2.2.1 Feature Descriptions

We developed many features for the task of named entity recognition of DNA, RNA, cell line, cell type, and protein names. Our features can be characterized into several groups: base features, number-related, alphabetical features, non-alphanumeric-related, Greek characters, sequence features, positional features, and other features. We created each feature by examining the words in the test set that are of each entity category and finding similarities between the words. The following tables describe each feature.

Base Features
WORD The current word.
PREV_LABEL The label assigned to the previous word in the sentence.

Number-Related Features
HAS_NUMBER The current word contains at least one number.
NUM_ON_END The end of the word contains a number.
CHAR_NUMBER_COUNT The amount of characters that are numbers.
CONTINUOUS_NUMBER_COUNT The amount of numbers in the word where contiguous spans of characters that are numbers are counted as a single number.
ALL_NUMS All characters are numbers.

Alphabetical Features
LOWERCASE All alphabetical characters are lowercase.
UPPERCASE All alphabetical characters are uppercase.
FIRST_CHAR_CAP The first character is an uppercase letter.
HAS_CAP The word contains an uppercase letter.
CHAR_CAP_COUNT The amount of uppercase letters.
CONTINUOUS_CAP_COUNT The amount of uppercase letters in the word where contiguous spans of uppercase letters are counted as a single uppercase letter.
A_COUNT The frequency of the vowel 'A'.
E_COUNT The frequency of the vowel 'E'.
I_COUNT The frequency of the vowel 'I'.
O_COUNT The frequency of the vowel 'O'.
U_COUNT The frequency of the vowel 'U'.
Y_COUNT The frequency of the vowel 'Y'.
VOWEL_COUNT The amount of vowels in the word.
CONSONANT_COUNT The amount of consonants in the word.
CONSONANT_VOWEL_RATIO The ratio of consonants to vowels, truncated to one decimal place. The ratio is set to -1 * consonant count when the vowel count is zero.

MORE_VOWELS_THAN_CONS The vowel count is greater than the consonant count.
ALL_ALPHANUMERIC All characters are alphanumeric.

Non-Alphanumeric-Related Features
DASH The word contains a dash ('-').
DASH_COUNT The amount of dashes ('-') the word contains.
COMMA The word contains a comma.
HAS_NONALPHANUMERIC The word contains at least one non-alphanumeric character.
ONE_NONALPHANUMERIC The word contains only one non-alphanumeric character.

Greek Character Features
GREEK_CHAR_CONTAINS The word contains at least one spelled-out Greek character.
GREEK_CHAR_INDIV_CONTAINS One feature for each Greek character. The amount of occurrences of that Greek character in the word.
GREEK_CHAR_COUNT The amount of Greek characters in the word.

Sequence Features
PREV_WORD The previous word in the sentence (if it exists).
PREV_PREV_WORD The word at (position-2) in the sentence (if it exists).
PREV_PREV_PREV_WORD The word at (position-3) in the sentence (if it exists).
PREV_PREV_PREV_PREV_WORD The word at (position-4) in the sentence (if it exists).
NEXT_WORD The next word in the sentence (if it exists).
NEXT_NEXT_WORD The word at (position+2) in the sentence (if it exists).
NEXT_NEXT_NEXT_WORD The word at (position+3) in the sentence (if it exists).
NEXT_NEXT_NEXT_NEXT_WORD The word at (position+4) in the sentence (if it exists).
PREV_WORD_SHORT The previous word is less than four characters long.
PREV_WORD_LONG The previous word is greater than ten characters long.
FIRST_WORD The word is the first word in the sentence.
WORD_POSITION The raw position of the word in the sentence.
WORD_POSITION_NORM The position of the word normalized by the length of the sentence, truncated to one decimal place.

Other Features
LENGTH The amount of characters in the word.
SHORT_LENGTH The word is a length smaller than four.

2.2.2 Feature Selection

We began our feature development process by creating features one at a time and executing the program with various feature combinations to see if each new feature improved performance. Aside from the fact that the possible number of combinations quickly spirals out of control, we ran into other problems. While testing with our first five or six features, we found that many features performed horribly alone or in certain combinations with other features. The amount of combinations that resulted in dismal results would have quickly led us to believe that many of our features were worthless, but we found that for most features a few special combinations with other features existed that actually improved performance over the baseline, which was using only the WORD and PREV_LABEL features.

We soon realized that manually picking feature sets and evaluating each one in turn would not work. To solve this problem, we developed a feature selector that attempts to find the optimal feature set through hill climbing with random walks. This hill climbing strategy is primarily controlled by the setFeatures function within the Maximum Entropy Classifier.

2.2.2.1 Hill Climbing with Random Walks

The basic premise of our feature selector is that we start with our current best known feature set, continually make changes to the best feature set creating a new feature set, apply the new feature set to the data generating a score, and replace the best feature set with the new feature set if the score increases. In the beginning, our best known feature set is the

base features WORD and PREV_LABEL. These features are permanent features that we include in all potential feature sets.

All other features are considered "possible features." During each iteration, with a 15% chance we try a completely random feature set where each possible feature has a 50% chance of being included in the set. The other 85% of the time we create a new feature set based on the current best feature set such that each possible feature is flipped on or off with a probability of 0.5 / (number of possible features). Features are flipped on with the given probability if they are not included in the current best feature set, and features are flipped off with the given probability if they are members of the current best feature set.

We keep track of all of the feature sets that our feature selector has attempted so that we do not waste time attempting the same feature set twice. For each feature set attempted, our feature selector writes the output results to a file, executes the nerEval script for computing per entity F scores given the previously written file, and records the scores for the feature set in a CSV file.

2.2.2.2 Feature Selector Objective Function

We tried optimizing several different scores while hill climbing. Our first objective function that we tried to maximize was the overall F score. While this certainly worked in selecting features that improved performance overall, it had disappointing side effects for some of the entity categories.

The amount of protein entities in the corpus vastly outweighs the amount of other entities, such as RNA entities. Specifically, the hand-labeled test set contains only 313 RNA entities while it contains 7,598 protein entities. With our initial objective function (overall F score), our feature selector was able to maximize the objective by choosing feature sets that had a large positive impact on the protein category but had a large negative impact on the RNA category. When not including the sequence features, which we created after our initial hill climbing run, optimizing the overall F score resulted in our "best feature sets" not labeling any words as RNA entities.

While optimizing the overall F score led to improvements across the frequently occurring entity categories, it resulted in poorer performance across the less frequent entity categories such as RNA entities. To fix this problem we changed our objective function to be the sum of each entity category's F score. This change to the objective function was effective and caused our feature selector to select feature sets that generally improved performance in all categories. While our new objective function does not necessarily result in an optimized overall F score, we feel that it is more important to do reasonably well on all categories rather than have high performance on a few categories and low performance on others.

2.2.2.3 Best Feature Set

Over the course of one night our feature selector attempted 1,421 different feature sets. Table 2.2.2.3 details our highest scoring feature set given the sum of category F scores objective function.

Our most important features (aside from the base features) were the PREV_WORD and NEXT_WORD features. While creating features, we quickly found the previous and next words to be valuable and we continually manually added additional features such as the "previous previous word" and "next next word" features to the possible feature set until we saw no increase in the objective function. As shown in the table, our feature selector found the four words after the current word to be useful while only the three words before the current word were useful.

Out of our five number features, the feature selector only found NUM_ON_END to be useful. Of the non-alphanumeric features, only DASH and COMMA were important. The only alphabetical feature that was helpful

was the distinction between capitalized and uncapitalized words. While the features for containing spelled-out Greek characters and how many of each Greek character were useful, the overall count of how many Greek characters were in the word did not help. None of the length features or word position features were helpful.

Our feature selector also found two other feature sets that had identical scores to our best feature set in all categories. One of these additional feature sets contained all of the features from our best feature set, but also contained the feature ONE_NONALPHANUMERIC that had no positive or negative effect on the scores. The second additional feature set swapped the FIRST_WORD feature for the WORD_POSITION_NORM feature. Amongst these three feature sets that tied for the top score, we chose the one that resulted in the fewest features. While the second additional feature set swapped one feature for another, it actually results in more features because the FIRST_WORD feature is a single feature while the WORD_POSITION_NORM feature actually contains 11 features – one for each decimal value from 0.0 to 1.0 in 0.1 increments.

Table 2.2.2.3: Best Feature Set
WORD
PREV_LABEL
NUM_ON_END
DASH
FIRST_CHAR_CAP
COMMA
GREEK_CHAR_CONTAINS
GREEK_CHAR_INDIV_CONTAINS
FIRST_WORD
PREV_WORD
PREV_PREV_WORD
PREV_PREV_PREV_WORD
NEXT_WORD
NEXT_NEXT_WORD
NEXT_NEXT_NEXT_WORD
NEXT_NEXT_NEXT_NEXT_WORD

2.3 Performance Analysis

This section details the performance of our Maximum Entropy Classifier using our best feature set that is described in the previous section. We first discuss the precision, recall, and FB1 scores on the test set. We then provide an analysis of the successes and failures of our parser. We end with a discussion of possible improvements that could fix our parser's popular errors.

2.3.1 Precision, Recall, and FB1

Table Set 2.3.1a shows the scores produced by the provided nerEval script using our best feature set. Table Set 2.3.1b shows the baseline scores using only PREV_LABEL and WORD as features. As seen in the tables, we were able to make significant improvement in the overall accuracy from 83.72% to 90.57%, and in the overall FB1 from 40.40 to 67.38. While we were able to improve all entity category FB1 scores, the DNA and RNA entity categories saw the largest improvements with 17.56 to 59.60 and 6.12 to 59.92 respectively. These categories benefitted greatly from the "previous word" and "next word" groups of features.

Phrases Exist	7,119
Phrases Found	6,591
Phrases Correct	4,619
Overall Accuracy	**90.57**

	Precision	Recall	FB1	Phrase Count
Overall	70.08	64.88	**67.38**	6,591
DNA	66.76	53.82	59.60	1,128
RNA	66.98	54.20	59.92	106
Cell Line	63.78	53.03	57.91	439
Cell Type	72.22	67.18	69.60	853
Protein	71.32	69.96	70.63	4.065

Table Set 2.3.1a: Performance scores produced by the nerEval script on the standard test set with our best feature set (detailed in Table 2.2.2.3).

Phrases Exist	7,119
Phrases Found	4,173
Phrases Correct	2,281
Overall Accuracy	**83.72**

	Precision	Recall	FB1	Phrase Count
Overall	54.66	32.04	**40.40**	4,173
DNA	35.59	11.65	17.56	458
RNA	9.23	4.58	6.12	65
Cell Line	75.00	16.48	27.02	116
Cell Type	45.15	47.76	46.42	970
Protein	61.90	38.30	47.32	2,564

Table Set 2.3.1b: Performance scores produced by the nerEval script on the standard test set with the baseline feature set (PREV_LABEL and WORD only).

To gauge the relative importance of each feature in our best feature set we gathered score data for each feature set that could be generated by removing one feature from the best feature set. Table 2.3.1 shows the change in scores caused by removing one feature from the best feature set. As highlighted in the table, the most important of our extra features was the NEXT_WORD feature. Removing this feature caused significant drops in FB1 scores, with the largest changes of -16 in DNA FB1 and -44.70 RNA FB1!

The important of the additional "next word" features decreases as we move farther away from the current word. A similar pattern is seen in the "previous word" features. PREV_WORD is not nearly as important as NEXT_WORD because removing PREV_WORD only causes a maximum drop in an FB1 score of 7.43, but the other "previous word" features seem to have slightly less than equal importance to their respective "next word" feature (i.e. removing PREV_PREV_WORD causes slightly smaller drops in comparison to removing NEXT_NEXT_WORD).

Another interesting result of this analysis is that, as highlighted in the table, on three occasions removing a feature actually increases the score. Removing FIRST_WORD or PREV_PREV_WORD improves the cell line FB1 score while removing NUM_ON_END improves RNA FB1. The improvements are rather small and are outweighed by the up to 5.02 FB1 decreases that occur in other categories, so the overall net performance change is still negative.

Table 2.3.1: Change in Scores by Removing Features

Removed Feature	Change in… Overall Accuracy	Overall FB1	DNA FB1	RNA FB1	Cell Line FB1	Cell Type FB1	Protein FB1
COMMA	-0.40	-1.40	-2.17	-0.26	-1.67	-1.39	-1.10
DASH	-0.76	-2.82	-5.30	-4.26	-5.10	-3.40	-1.95
FIRST_CHAR_CAP	-0.44	-2.45	-1.88	-0.36	-1.80	-0.99	-2.98
FIRST_WORD	-0.25	-1.01	-2.27	-1.24	0.23	-1.68	-0.58
GREEK_CHAR_CONTAINS	-0.49	-1.67	-2.34	-1.30	-2.86	-1.87	-1.33
GREEK_CHAR_INDIV_CONTAINS	-0.68	-2.42	-5.27	-1.59	-3.34	-1.33	-2.08
NEXT_NEXT_NEXT_NEXT_WORD	-0.21	-0.86	-1.85	-1.00	-0.74	-1.31	-0.50
NEXT_NEXT_NEXT_WORD	-0.57	-1.82	-3.85	-3.14	-0.16	-1.48	-1.45
NEXT_NEXT_WORD	-1.17	-3.72	-5.88	-4.02	-6.63	-3.09	-2.85
NEXT_WORD	-2.51	-9.21	-16.00	-44.70	-7.74	-14.32	-5.94
NUM_ON_END	-0.62	-2.21	-2.14	0.69	-0.29	-3.38	-2.35
PREV_PREV_PREV_WORD	-0.68	-2.24	-5.27	-0.19	-2.22	-2.58	-1.47
PREV_PREV_WORD	-0.56	-2.52	-5.02	-4.17	0.11	-1.60	-2.41
PREV_WORD	-0.61	-3.96	-7.43	-7.11	-2.73	-4.36	-2.91

2.3.2 Successes and Failures

In this section we explore the successes and failures of our Maximum Entropy Classifier. We present six examples that detail the ranges of good and bad decisions that we found in our classifier's output. As all of these examples show, our previous and next word sets of features are the strongest explanations for most of our correct and incorrect classifications.

2.3.2.1 Perfect, Perfect, Perfect

In several instances, our classifier was able to correctly label all words in the sentence. Example 2.3.2.1 shows such a sentence. The sentence contains both words of the DNA and protein categories. Our classifier successfully labels proteins that span single words, such as "AML-1B" and "OSF2." Our system is also able to correctly classify sequences of words in both the DNA and protein categories, such as "osteoblast-specific cis-acting element" and "PEBP2 alpha/AML-1 family." It is also able to sequences as proteins correctly where a filler word exists inbetween the sequence and the classifier correctly classifies the filler word as in the O category. This is seen in the correct classification of the word "of" near the end of the sentence.

Example 1		
Word	Gold Label	Guessed Label
Thus	O	O
this	O	O
study	O	O
demonstrates	O	O
that	O	O
AML-1B	protein	protein
can	O	O
increase	O	O
gene	O	O
expression	O	O
of	O	O
an	O	O
osteoblast-specific	DNA	DNA
gene	DNA	DNA
through	O	O
its	O	O
binding	O	O
to	O	O
an	O	O
osteoblast-specific	DNA	DNA
cis-acting	DNA	DNA
element	DNA	DNA
and	O	O
presents	O	O
evidence	O	O
that	O	O
OSF2	protein	protein
is	O	O
a	O	O
member	O	O
of	O	O
the	O	O
PEBP2	protein	protein
alpha/AML-1	protein	protein
family	protein	protein
of	O	O
transcription	protein	protein
factors	protein	protein
.	O	O

2.3.2.2 Errors on Long Sequences

Example 2.3.2.2 is another example in which we identify most labels except for a few. The first error is that the word "erythropoietin" is labeled as other instead of as a protein. After examining all of the other occurances of "erythropoietin" in our corpus, we found that "erythropoietin' is labeled as protein only when it is followed either by "(EPO)" or "induces." In some sentences, it was labeled as a cell type, but only when proceeded or followed by different words. Our most important features aside from the base features are the "previous word" and "next word" sets of features. We presume that since "erythropoietin" is not preceded or followed by any of these words to give it weight as a protein or cell type, we miss the target here.

The case for "highly" and "purified" are similar in that we believe our classifier missed the labels because of the "previous word" and "next word" sets of features. Our best feature set contains the previous three words as features as well as the next four words. The main identifying word that must be known in this case is the word "cells." The fact that the se-

quence of words "highly purified human colony forming unit-erythroid" is immediately followed by the word "cells" is what leads a human reader to determine that many of the words before "cells" are part of a sequence that define a cell type. Since our best feature set only includes the next four words, the labeling decision for "highly" and "purified" are not privy to the knowledge of the word "cells" that is nearby. Our classifier is able to correctly classify the words in the sequence as soon as the word "cells" becomes one of the "next word" features.

Like the majority of the long cell type sequence, the last two labeled words in the sentence, "CD34(+)" and "cells" are again labeled correctly because of the "CD34(+)" labeling decision's knowledge of the future word "cells" and because of the "cell" labeling decision's knowledge of the previous label being a cell type and the previous word being "CD34(+)." We experimented with removing all of the previous and next word features and saw that as a result that many of the sequences were not successfully labeled.

Example 2.3.2.2		
Word	Gold Label	Guessed Label
We	O	O
examined	O	O
signaling	O	O
by	O	O
erythropoietin	protein	O
in	O	O
highly	cell_type	O
purified	cell_type	O
human	cell_type	cell_type
colony	cell_type	cell_type
forming	cell_type	cell_type
unit-erythroid	cell_type	cell_type
cells	cell_type	cell_type
generated	O	O
in	O	O
vitro	O	O
from	O	O
CD34(+)	cell_type	cell_type
cells	cell_type	cell_type
.	O	O

2.3.2.3 Same Word Different Labels

In these section we examine how the same word can have different labels depending on its context. Example 2.3.2.3a is a sentence where the word "sickle" appears as both a cell type and as an "other." Similar to our explanation in the previous section, the correct labeling of "sickle" as a cell type in the first occurrence is primarily due to its knowledge of the word "cells" in the future. In the second occurrence, the correct label of other is given even though the word "cell" appears as the next word. We believe that the correct labeling in this case is because of its knowledge of the word "disease" immediately after "cell", which is a strong indicator that both the word "sickle" and "cell" are modifiers specifying a particular disease.

Example 2.3.2.3a		
Word	Gold Label	Guessed Label
The	O	O
abnormal	O	O
adherence	O	O
of	O	O
sickle	cell_type	cell_type
red	cell_type	cell_type
blood	cell_type	cell_type
cells	cell_type	cell_type
(	O	O
SS	cell_type	cell_type
RBC	cell_type	cell_type
)	O	O
to	O	O
endothelial	cell_type	cell_type
cells	cell_type	cell_type
has	O	O
been	O	O
thought	O	O
to	O	O
contribute	O	O
to	O	O
vascular	O	O
occlusion	O	O
,	O	O
a	O	O
major	O	O
cause	O	O
of	O	O
morbidity	O	O

in	O	O
sickle	O	O
cell	O	O
disease	O	O
(	O	O
SCD	O	O
).	O	O

Example 2.3.2.3b shows another example where "sickle" is correctly identified as being in the other category. Again, like the case with the word "disease", the labeler's knowledge of "anemia" and "patients" after the word "cell" are probably important indicators that "sickle" and "cell" in this case should both be in the other category.

Example 2.3.2.3b		
Word	Gold Label	Guessed Label
In	O	O
addition		O
HU	O	protein
stimulates	O	O
the	O	O
synthesis	O	O
of	O	O
fetal	protein	O
hemoglobin	protein	O
in	O	O
sickle	O	O
cell	O	O
anemia	O	O
patients	O	O
.	O	O

The following set of examples show the word "endothelial" labeled correctly as both a cell type and a cell line depending on the context. In Example 2.3.2.3c "endothelial" is correctly labeled due to the presence of "cells" immediately following the word. In this example we believe "sickle erythrocytes" is not labeled as a cell type because the word "with" is generally a separator between entities and ends up putting a lot of weight on the other category, overruling the information that "erythrocytes" has about the word "cells" as a future word.

Example 2.3.2.3c		
Word	Gold Label	Guessed Label
sickle	cell_type	O
erythrocytes	cell_type	O
with	O	O
endothelial	cell_type	cell_type
cells	cell_type	cell_type
in	O	O

Example 2.3.2.3d presents another example where "endothelial" is correctly labeled as a cell line. In this case, we believe "endothelial" is labeled as a cell "something" because of the word "cells" immediately following it, and that particular "something" is a cell line because of the PREV_LABEL feature. This example is also a great example where our classifier is able to correctly label a sequence of words that is more than five words long. In this case, "cultured" is such a strong indicator of a cell line that it does not need to know about all of the words in the sequence in order to make the proper decision. In fact, the decision that "cultured," which was propagated along to the other decisions by the PREV_LABEL feature, is probably the sole reason that the rest of the sequence is labeled a cell line instead of a cell type.

Example 2.3.2.3d		
Word	Gold Label	Guessed Label
cultured	cell_line	cell_line
human	cell_line	cell_line
umbilical	cell_line	cell_line
vein	cell_line	cell_line
endothelial	cell_line	cell_line
cells	cell_line	cell_line
(	O	O
HUVEC	cell_line	cell_line
)		O
resulted	O	O

Example 2.3.2.3e is another example where our classifier correctly classifies a sequence longer than five. In this case, the decision was easy because the words "cell line" are in the middle of the phrase and visible to all of the words in the sequence. Unfortu-

nately, our classifier is a little overzealous and continues the labeling escapade by labeling "with" and "mutated" with cell line.

Example 2.3.2.3e		
Word	Gold Label	Guessed Label
transfectants	O	O
of	O	O
the	O	O
porcine	cell_line	cell_line
vascular	cell_line	cell_line
endothelial	cell_line	cell_line
cell	cell_line	cell_line
line	cell_line	cell_line
PIEC	cell_line	cell_line
with	O	cell_line
mutated	O	cell_line

Example 2.3.2.3f shows an interesting example where we label "endothelial" incorrectly. We have our reservations in this case in regards to the accuracy of the hand labeled "truth" answers. We feel that the phrase "endothelial and RAW264.7 cells" is actually distributing both "endothelial" and "RAW264.7" to "cells", and thus making the meaning more explicit we read the phrase as "endothelial cells and RAW264.7 cells." Due to our interpretation of the phrase's meaning, we believe that endothelial should not be an "other" in the hand labeled set and should be a cell type due to similarity with Example 2.3.2.3c.

Example 2.3.2.3f		
Word	Gold Label	Guessed Label
expression	O	O
in	O	O
endothelial	O	cell_type
and	O	O
RAW264.7	cell_line	cell_line
cells	cell_line	cell_line

2.3.2.4 Previous Label Issues

Example 2.3.2.4 shows an instance where the first word in a conjunction is correctly labeled as a protein but the second word is labeled as an other. We believe that this is mostly because we only include knowledge of the previous label, and thus the labeler does not know about TCRzeta's label when making its decision on "p56(lck)." We posit that with "previous previous label" knowledge our classifier would be able to correctly classify "p56(lck)" in this situation because the fact that "p56(lck)" is in a conjunction with another word that is a protein should increase the probability that "p56(lck)" is also a protein. We did not include more than one previous label as a feature because the provided infrastructure prohibited it.

Example 2.3.2.4		
Word	Gold Label	Guessed Label
Diminished	O	O
expression	O	O
of	O	O
TCRzeta	protein	protein
and	O	O
p56(lck)	protein	O
that	O	O
are	O	O

2.3.3 Future Improvements

As indicated by Example 2.3.2.4, we believe the most important improvement to our parser will come from adding additional label history, such as the "previous previous label" and "previous previous previous label." We did not find an easy way to include these features within the provided infrastructure, but we believe the addition of these features will increase performance when labeling items in conjunctions or lists.

Another possible improvement would be to incorporate a second pass through the sentence. Example 2.3.2.2 included a long sequence of words that should all be labeled as a cell type. Our classifier missed the first two words "highly" and "purified." Perhaps if we took a second pass through the sentence and allowed "highly" and "purified" to have knowledge of the labels in front of them, then we would be able to correctly label the entire sequence.

3 Treebank Parsing

We implemented a CKY parser for the treebank parsing task. In this section we describe our parser's implementation, provide an analysis of our parser's performance, and discuss possible improvements to the parser that would correct frequently occurring errors.

3.1 Implementation

The CKY parsing algorithm is straightforward and well known. In this section we will only describe the key optimizations used. We will not describe the standard implementation details such as how to follow the back points to generate the final parse tree.

After generating the lexicon and grammar in the training phase, we create several data structures that we using in the CKY algorithm to manage nonterminals and grammar rules. First, we create an array ntToWord that contains all of the nonterminals that are preterminal. Next, we create an array that contains all nonterminals, with the nonterminals from ntToWord being at the same index as in ntToWord. Additionally, we create two flat arrays binaryRules and unaryRules, containing the binary rules and unary rules from the grammar. Each rule in the two arrays contains as the parent/children by their index into the array of nonterminals as well as the rule's score. We also create a temporary hash map within the training method that maps the string version of each nonterminal to its index in the array. We need this hash map because the provided grammar class does not reference nonterminals by an integer ID and instead directly uses the string representations. We use standard three-dimensional arrays for the score and back pointer data structures.

All optimizations revolve around minimizing the number of nonterminals that we loop over as well as minimizing the amount of processing time required to iterate over a set of values. The algorithm has three basic sections. The first section involves looping over the nonterminals and placing a probability in a cell of the score array if the nonterminal is a preterminal. Naturally, instead of looping over all nonterminals here we only loop over the nonterminals in ntToWord.

The second section involves considering unary rules. In the standard pseudocode, one iterates over all pair combinations of nonterminals indicated by the placeholders A and B. This can be unwieldy because our corpus contains 3,919 nonterminals. In two layers of for loops this results in considering 15.4 million different nonterminal pairs. To sidestep this issue, it is important to notice that we only perform meaningful work if the rule A→B exists in the grammar. Thus to efficiently implement this section we loop over the unaryRules array, skipping the inner computations if the rule's score is zero or if the score of B in the current cell is zero.

The third section involves three loops over the length of the sentence. Unfortunately, these three loops cannot be reduced. Inside the innermost loop are two sections. The second section is managing unary rules, which is optimized in the same way as described in the previous paragraph. The first section inside these loops involves iterating over all triplet combinations of the nonterminals indicated by the placeholders A, B, and C. If we were to naïvely implement this section as three for loops then we would be iterating over 60.2 billion triplets with our current dataset.

To efficiently iterate over these triplets, we iterate over the binaryRules array. If B's score in the cell of the score array that will be used in the impending probability calculation is zero, or C's score in its respective cell is zero, or the rule's score is zero, then we skip immediately to the next rule. Otherwise, we perform the calculations.

A final, smaller optimization that we implemented was from noticing that within many of the for loops the first two dimension indices used to index into the score and back pointer arrays are fixed. To reduce the amount of memory lookups, at the beginning of such sec-

tions we index two dimensions into the arrays and keep pointers to the resulting one-dimensional array. We use these pointers within the inner loops instead of re-indexing into the original arrays. This optimization was far less important than the others, but it reduced the average computation time per sentence by 400 milliseconds in the case of 69-word sentences.

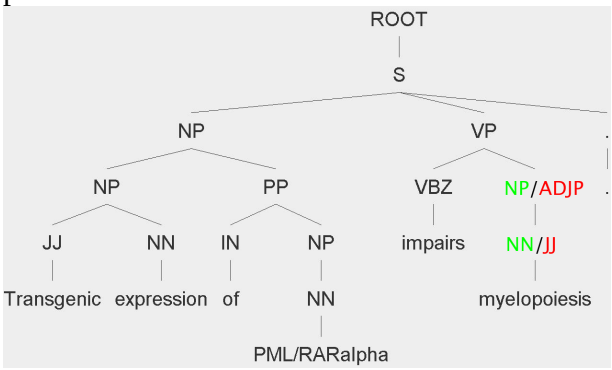
These simple but important optimizations enable our implementation to parse 20-word sentences in 125 milliseconds and 69-word sentences in 4.8 seconds. Table 3.1 shows the average processing time of our parser by sentence length. The only lengths that are shown are lengths that exist in the provided genia test set. All computations were performed using one core of an Intel Core 2 Duo Mobile Processor T7500 running at 2.2 GHz.

Table 3.1: Parsing Speed

Length	Average Time (milliseconds)	Length	Average Time (milliseconds)
6	6	30	432
7	6	31	374
8	7	32	436
9	13	33	634
10	14	34	811
11	19	35	635
12	35	36	860
13	31	37	676
14	41	38	1,160
15	48	39	1,087
16	119	40	1,319
17	72	41	1,165
18	117	42	1,393
19	165	43	1,007
20	125	45	1,615
21	154	46	1,433
22	164	47	1,283
23	190	49	1,455
24	226	51	2,034
25	212	54	2,206
26	252	58	3,171
27	288	59	3,154
28	386	69	4,849
29	347		

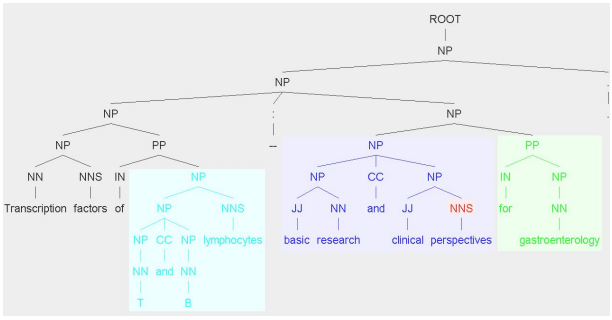
3.2 Performance Analysis

In this section we state and show some examples of the parser without the chunking and the er-rors it leads to. In Example 3.2a, the only error made by the parser is to label NP as ADJP and NN as JJ. That is, parsing a Noun Phrase as an Adjective Phrase. This in turn leads to parsing a Noun as an Adjective. This is a common error that the non-chunking parser makes.

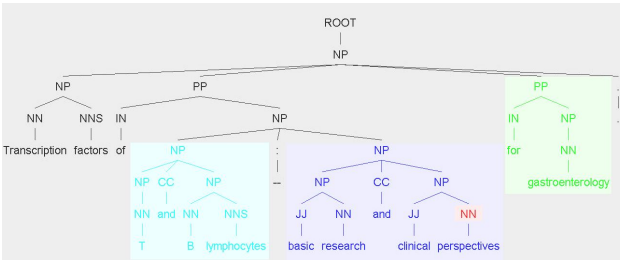


Example 3.2a: Green indicates "Gold" tag while red indications our parser's tag.

In Example 3.2b, the non-chunked version positions four branches of sub-trees in a different location and there is one labeling error of an NNS tag as an NN tag. A noun in plural form is parsed as a noun in singular form, which we consider a minor error. The wrong positioning of the subtrees is a more serious error. The green-colored subtree in the images is supposed to be positioned as a Prepositional Phrase (PP) under an inner Noun Phrase (NP) subtree whereas our parser positions it as a PP under an outer level NP subtree which leads to the outer Noun phrase now having two consecutive PP's. One possible way to ensure that this doesn't happen is to penalize the score when it tries to place such prepositional phrases one after the other. We are proposing this solution as we think that such consecutive prepositional phrases are not common in English. In the second case of "T and B lymphocytes" (the teal-colored subtree) the change in positioning can be attributed to the use of the conjunction "and." It could be that instead of interpreting the statement as "T lymphocytes and B lymphocytes" it is interpreted as "T AND B lymphocytes" which leads to the separation of "B lymphocytes" as a separate Noun Phrase.



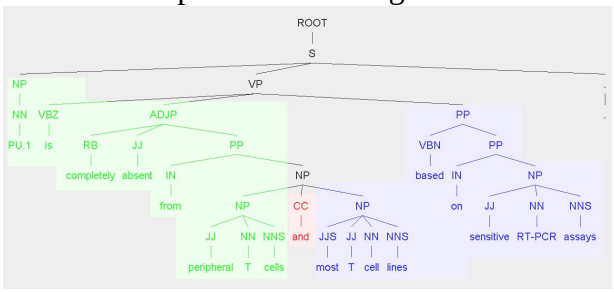
The "Gold" tree.



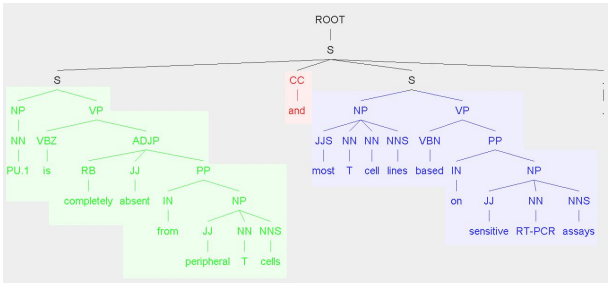
Our parser's tree.

Example 3.2b: Similar colored regions in each image are identical subtrees. The red NNS or NN tag shows the incorrect tag that exists in the blue-purple subtree.

In Example 3.2c, the sentence is "PU.1 is completely absent from peripheral T cells and most T cell lines based on sensitive RT-PCR assays." The gold parsing breaks the sentence into a Noun phrase and a Verb phrase. But our non-chunked parser is breaking this into two sentences connected by an "and." This can again be explained by following the argument for the previous example. This is again a wrong interpretation of what the "and" in the middle of the sentence is connected.



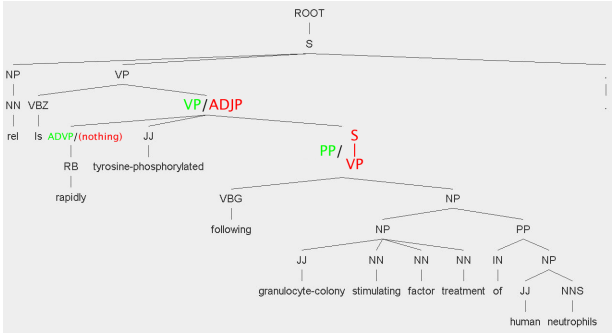
The "Gold" tree.



Our parser's tree.

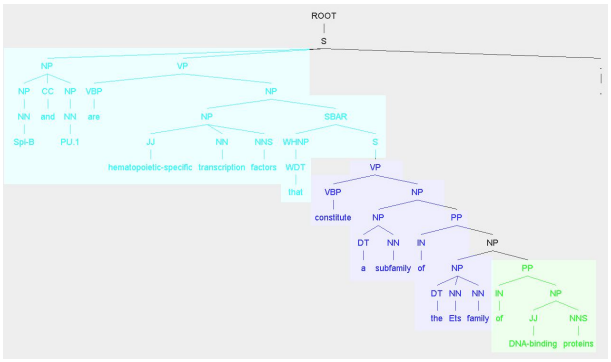
Example 3.2c: Similar colored regions in each image denote similar subtrees.

Going on to Example 3.2d, here the errors are due to wrong tagging. The incorrect back propagation of the right sub tree as a verb phrase and then as a sentence instead of as a Preposi-tion phrase leads to the wrong interpretation of a verb phrase with "rapidly" as an adverb as an adjective phrase with "tyrosine-phosphorylated" as an adjective phrase.

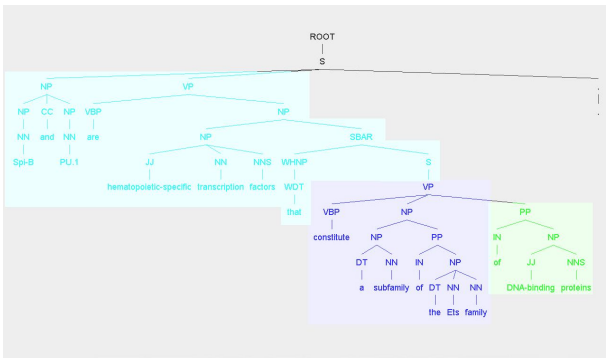


Example 3.2d: Green = "Gold" tag, red = parser's tag.

In Example 3.2e, at the start of the bottom-up parsing, the part of the sentence "a subfamily of the Ets Family of DNA-binding proteins" is where the wrong positioning of sub trees hap-pens. The rest of the sentence is parsed correctly. So, instead of tagging the sentence as "a sub-family <u>of</u> the Ets family of DNA-binding proteins" our parser is tagging it as "a subfamily of the Ets family <u>of</u> DNA-binding proteins." Thus the wrong positioning of the trees due to con-fusion about which preposition combines which part of the sentence.



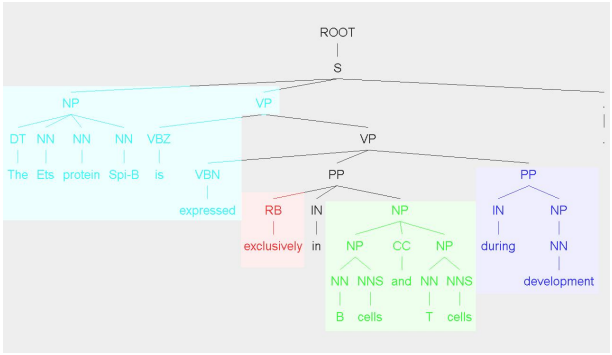
The "Gold" tree.



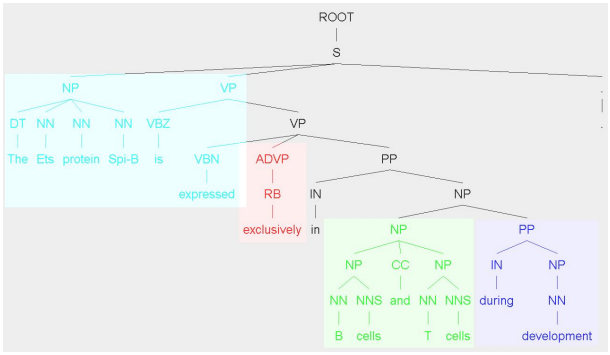
Our parser's tree.

Example 3.2e: Similar colored regions in each image denote similar subtrees.

Example 3.2f is again of wrong positioning of a sub tree. The verb "expressed" in the sentence should apply to the prepositional phrase "during development" whereas our parser com-bines it with the noun phrase "B cells and T cells." This leads to the assumption that the adverb "exclusively" applies to the prepositional phrase "in B cells and T cells during development." There are many examples in which our parser does create the entire structure correctly but we are not presenting that here. These exact matches, as described in Table 3.3a, are up to 33% of the test sentences.



The "Gold" tree.



Our parser's tree.

Example 3.2f: Similar colored regions in each image denote identical subtrees.

3.3 Improvements

We implemented second order vertical markovization to improve our parser's performance. Our implementation of vertical markovization allows a number to be passed by command line to the program that specifies the order of vertical markovization to use. We experimented with different orders of vertical markovization to see the effect on the F1 score for the test set. Table 3.3a shows the average score for several orders of vertical markovization for all sentences in the test set with no more than twenty words in the sentence while Table 3.3b shows the average score using the entire test set, which includes sentences up to length 69. We also include the total processing time for the respective test set.

As seen in Table 3.3a, the sweet spot given our current training set is a third order vertical markovization, which provides a slight improvement over second order. High orders decrease performance in comparison with the third order because of the increasing sparseness due to a fixed training set size. If we were to use additional data, third order improvement over second order would probably be larger and we might even find additional improvement by moving to fourth order. One

important note is that while sixth order performs worse than third order, it still has higher scores in comparison to first order.

Another interesting feature of this data is that third order vertical markovization requires three times the processing time of order one, while order four requires five times that of order one and order five requires almost eight times that of order one. This increase in processing time is mostly due to increasing the amount of nonterminals and thus increasing the amount distinct rules that exist in the grammar. The only slight increase in processing time from order five to order six is presumably because our training set contains few sentences with a tree depth larger than five.

Table 3.3a: Average Scores on All Sentences with Twenty Words or Less					
Order	Precision	Recall	F1	Exact Match (%)	Test Set Processing Time (seconds)
1	76.84	73.94	75.36	9.26	4
2	84.44	83.52	83.98	33.33	6
3	**84.41**	**84.14**	**84.28**	**33.33**	**12**
4	81.72	82.10	81.91	29.63	20
5	80.90	81.79	81.34	25.93	31
6	79.47	79.59	79.53	22.22	33

Table 3.3b: Average Scores on All Sentences from the Test Set					
Order	Precision	Recall	F1	Exact Match (%)	Test Set Processing Time (seconds)
1	71.35	66.63	68.91	3.90	66
2	77.34	76.59	76.96	**16.88**	123
3	**77.38**	**77.08**	**77.23**	16.23	211
4	75.37	75.62	75.49	14.29	325

Table 3.3b shows similar trends when testing on the entire test set, which contains sentences of up to length 69. We only include up to order four in the table because the higher orders required more memory than the 2.7 GB that we were allocating to the process. Again, third order vertical markovization scores the best (again only slightly better than order two), but the cost of almost twice the computation time over second order is magnified by the larger test set. In terms of computation time, it is clear that one gets the most performance for the dollar with order two vertical markovization.

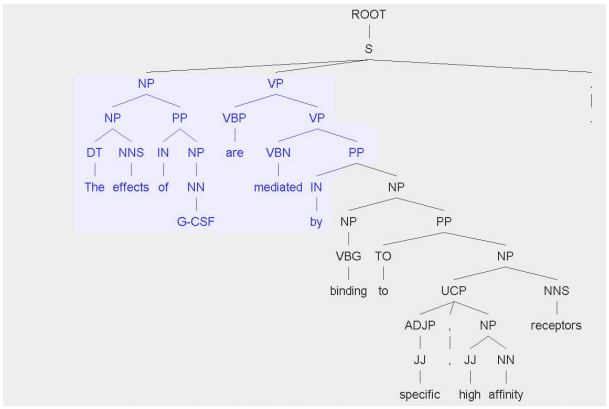
In Section 3.2 we saw that labeling a noun as an adjective, and singular/plural noun mixups were common. We believe that additional training data would significantly help in these cases because the errors are probably because the words were never seen or infrequently seen in the training set.

Another problem we saw as incorrect positioning of subtrees due to misinterpreting the phrases and their positions in the sentence. These types of errors fall into two categories: grammar rule violations and sentence interpretation dependence. The grammar rule violation errors can be avoided by using additional training data that would result in strong probabilities that give "penalties" to grammar violations. The sentence interpretation issues are more difficult to solve, but could definitely benefit from some type of shallow semantics processing.

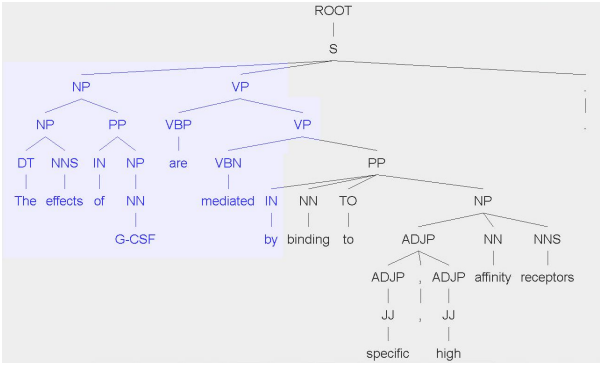
4 NER/Parser Combination

This section deals with the performance results after the NER and the parser are combined. We compare the results we obtain with the non-chunked versions to see how much of an improvement we receive from chunking. In Example 4a, the non-chunked version has the error of positioning stuff at the wrong place. It combines parts of three phrases to form one phrase, "by", "binding", "to" and putting them as part of a prepositional phrase and breaking up a noun phrase "high affinity receptors" by forming a "Unlike Coordinated Phrase" with an adjective. As we see in the chunked version, one of the two errors is corrected due to the chunking part. The "high affinity receptors" gets put back into one phrase and the UCP is no longer created. But the other error with relation to "by binding to" is still made. This can be considered as an example to show that chunking does make things a little better.

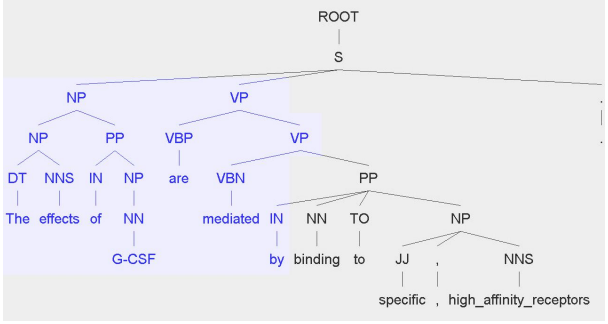
In Example 4b, the non-chunked version again exhibits both errors, wrong labeling of tags and wrong positioning of a few sub trees. The non-chunked version wrongly labels "resting" as an adjective, which leads to the combining of "not in memory" as an adverb phrase and further on combined with "resting" to become an adjective phrase. This wrong labeling leads to most of the wrong combination. When you look at the chunked version, even though one of the errors of labeling "resting" as an adjective is rectified, it still does not get the sub trees right. A major part of the right tree is combined with the left subtree leaving an even worse version of the non-chunked tree.



The "Gold" tree.

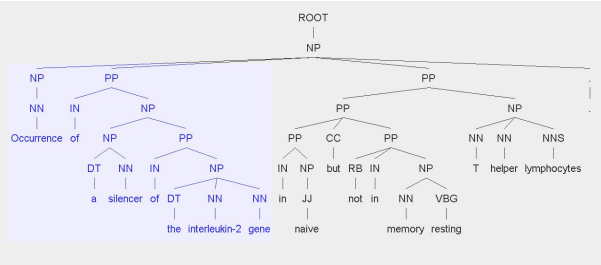


Our parser's tree without chunking.

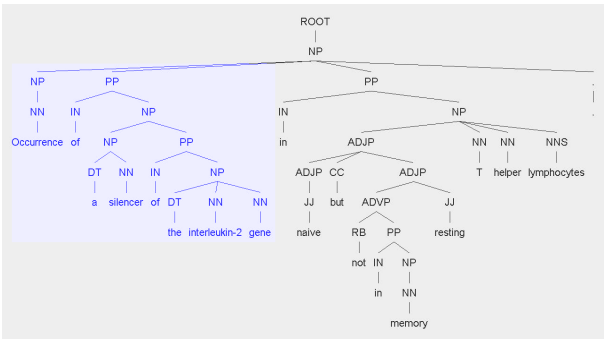


Our parser's tree with chunking.

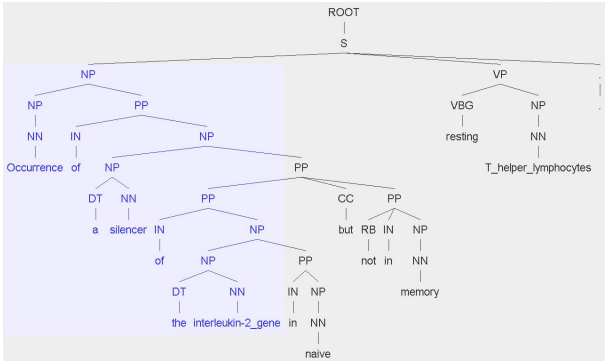
Example 4a: Similar colored regions in each image denote identical subtrees.



The "Gold" tree.



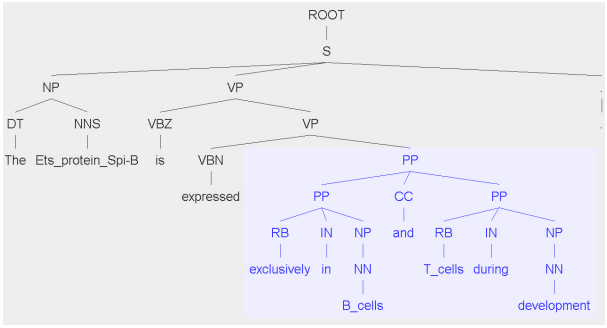
Our parser's tree without chunking.



Our parser's tree with chunking.

Example 4b: The colored region is the region that remains fairly constant.

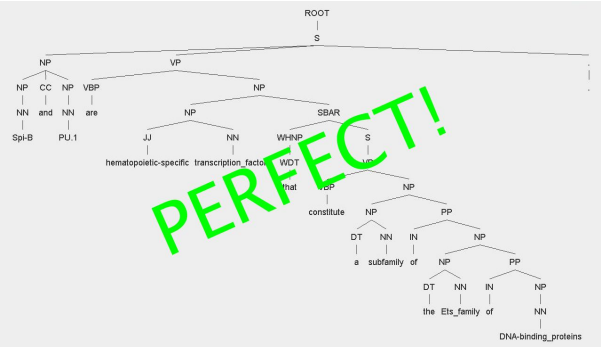
We had dealt with the non-chunked and gold versions of Example 4c as Example 3.2f. Now, if we compare the non-chunked and chunked version, we can see that it still interprets it in a wrong way. The verb "expressed" is applied to a prepositional phrase; it is applied to more than just the phrase it is supposed to apply to. The prepositional phrase "exclusively in B cells and T cells" should be interpreted as the "AND" connector between B cells and T cells, but the chunked parser interprets it as an "AND" connector between "exclusively in B_cells and T_cells during development". Thus though it avoids one error, it brings in a different kind of error. This example stands for cases where the chunked parser slightly worsens the result because of chunking.



Example 4c: Our parser's tree with chunking. The colored region is the region that changes in relation to the unchunked version from Example 3.2f.

Example 4d continues from Example 3.2e, the non-chunked version made the error of misinterpreting the purpose of the "of" preposition and thus instead of tagging the sentence as "a subfamily of the Ets family of DNA-binding proteins" our parser is tagging it as "a subfamily of the Ets family of DNA-binding proteins." In the chunking version, it does not do this and instead perfectly follows the correct structure and also identifies the correct labels. This is an example to show that adding

the chunking part actually helps avoid the errors in the non-chunking parser.



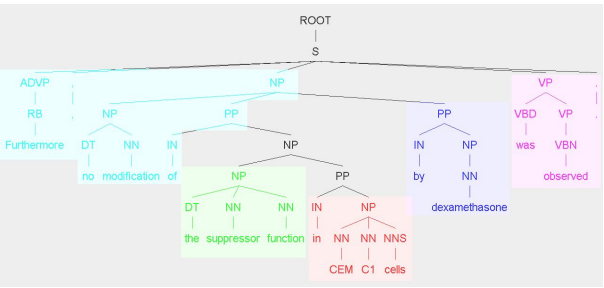
Example 4d: Our parser's tree with chunking. The parse is a perfect match to the Gold version from Example 3.2e.

In Example 4e, we show a case where the non-chunking parser does not make any mistakes but the chunked parser makes an error in interpreting the preposition usage. Notice that the gold and non-chunked versions are similar (including the labels). In the chunked version, the sentence is interpreted as "the suppressor function in CEM_C1_cells by dexamethasone" instead of "the suppressor function in CEM C1 cells by dexamethasone."

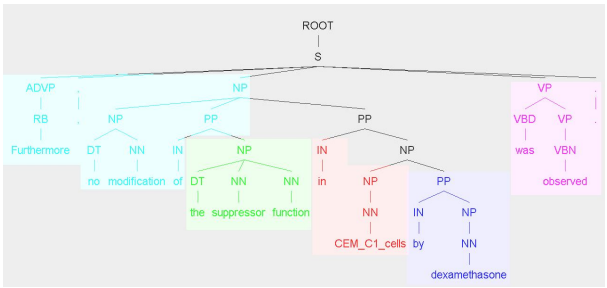
Thus it can be seen that we cannot really say whether chunking helps or not even when we do an in-depth analysis by comparison. In some cases it seems to help, in others it seems to worsen, and in some it seems neutral.

5 Member Contributions

Todd and Pavani pair programmed the majority of the assignment. Todd individually created the hill climbing feature selection for the Maximum Entropy Classifier, conceived and implemented all optimizations, and developed most maximum entropy classifier features. Pavani individually selected the performance analysis examples. Todd and Pavani collectively wrote the report, discussed the examples/performance analysis, and created the tree visualizations. Todd wrote all sections of the report except the sections with examples.



The "Gold" tree and our parser's tree without chunking.



Our parser's tree with chunking.

Example 4e: Similar colored regions in each image denote similar subtrees.