CS224N Final Project Writeup
EMTAIL: Eric-Mike Tree Associations in Language

Eric Karl and Mike Mintz

2007 June 1

1 Learning Entailments

The perfect natural language program would be able to translate natural language sentences into modal or first order propositions, keep a knowledge and belief database, and generate relevant natural language sentences about the state of the world (from the information in its knowledge base). However, we don't have a magical sentence-to-logical-proposition converter, and it seemed like that would take more than 3 weeks to build. The next best thing is a program that can (1) translate sentences into syntactic trees, (2) keep a database of trees that we've read, and (3) generate natural language sentences about the state of the world from the trees we have. The first two steps are straight-forward if you have a parser, but its the third that presents a challenge. Given a bunch of sentence trees, how do we know what unseen trees are true? For example, having a tree for "every man is mortal" and a tree for "Socrates is a man" should allow us to generate (or at least verify) "Socrates is mortal".

We don't intend to bring inference rules from logic to NLP, but rather we intend to find associations that connects a tree to the other trees it can entail. There are two sorts of entailments we hope to find: language-based entailments that are always true given the language, and world-based entailments that are true because of the way the world is. An example of the first is that "*X buys Y from Z*" entails "*Z sells Y to X*" (and the converse), and an example of the second is that "*X killed somebody*" entails "*X is a criminal*". Some entailments are necessary such as the first example, while some are true most of the time, such as one where "*X didn't study for Y*" entails "*X failed Y*", so ideally we would assign probabilities to entailments.

1.1 Applications in Question Answering

Having a system to generate entailed trees given an input tree would greatly aid in the field of question answering. A question answering system has a bunch of sentences it learns from, and a question it is asked. If it has the sentence "*Mike sold his computer to Jane*" in the training set, it would be relatively simple to implement a system that could answer "*What did Mike sell to Jane?*", but having entailments would allow the system to answer "*What did Jane buy from Mike?*". Questions that aren't directly encoded as trees in the training data but entailed by trees in the training data can be answered if the question answering system knows these kinds of associations between trees.

1.2 Applications in Anaphor Resolution

Although anaphor resolution generally aims to find NPs that refer to the same entity, one can use learned entailments to find VPs that refer to the same action. Take the following fragment, "*I sold my computer to Jane. The reason Jane bought my computer is because ...*". Knowing the buy/sell entailment will allow identification of the first VP with "*Jane bought my computer*".

1.3 Applications in Document Summarization

Another application where learned entailments can be useful is in document summarization. To effectively summarize a document, one thing that's guaranteed to help is eliminating redundancy. If one sentence in a document entails another, removing the second sentence will preserve the information contained in the document. Customizing the probability cutoffs of removed entailments can result in different size summaries. Also, using the previous application in anaphor resolution, one could summarize the fragment given as either simply "*The reason Jane bought my computer is because ...*" or "*I sold my computer to Jane. The reason* ***for that*** *is because ...*"

1.4 Related Work

Lin and Patel, in "Discovery of Inference Rules for Question Answering" (2001)[1], had a similar goal in mind. They looked through large amounts of text to find verb phrases that were similar, so that they could find similarity relationships (called inference rules) such as "*X wrote Y* $\approx$ *Y is the author of X*". Using Minipar-generated dependency structures, they created a statistical profile for every verb phrase with two arguments, so that each one had a list of what sorts of arguments it generally takes. By finding verb phrases that had very similar arguments, they generated inference rules. For example "*X wrote Y*" and "*Y is the author of X*" theoretically should both have a lot of names of authors in X and should both have a lot of names of books in Y.

Since there is no gold standard set of entailments (at least not easily creatable) that one can use to test, and they had not implemented a question-answering system, Lin and Patel evaluated their system by manually paraphrasing a set of sentences and seeing how many of these paraphrases their system would generate, as well as seeing what percent of paraphrases the system made were correct (by manual analysis). They used TREC-8 questions so that they could see how well their system would work in a question-answering system.

Most of the paraphrases their system generated were correct, but they only generated a fraction of the ones they expected to get (the manual paraphrases). So the system they implemented can pretty much only help question-answering systems, but it suffers from low-coverage. Furthermore, it can only analyze binary verb phrases, and it can't get one way entailments, such as "*X wrote Y* $\Rightarrow$ *Y knows what X is*" without getting the incorrect entailment in the other direction.

2 Initial Implementation: Surface Structure

To see what kinds of relations we could find, we started looking for patterns on the surface (before parsing into trees). (This gave Mike something to do while Eric tried to integrate Minipar into our project.) We created a regular expression that looked for phrases of the form *X [be] Y and Z*, where [be] is in any conjugation. We also looked for phrases of the form *X [be] Y and [be] Z* and *X [be]*

[1] Available at `http://www.isi.edu/~pantel/Content/publications.htm`

Y and X [be] Z, but very few sentences of this form were found, so we removed them to improve performance.

We then stored two-way entailments between Y and Z. Although this is not logically valid, there is an intuition that when someone says "and", the words often go together. Words that are unlikely to go together never appear in this form, and when they do appear together, it is with a contrasting word like "but" or "yet". This is implemented in the Java class GetSurfaceData.

2.1 Results

Running GetSurfaceData on *Pride and Prejudice* yields 48 pairs of properties (from about 8,000 sentences). Of these, 30 were completely wrong because X and Y were not parallel types (we were intending for them to be adjectives gerunds). We get results like *X was soon joined by Mary and Kitty* and *X was eclipsed by Mr. Bingly and Netherfield*. 5 results were parallel phrases but were not at all related, like *X was affectionate and insincere*. 5 results were related but not perfect entailments, like *X was lively and unreserved*. Finally, 8 results were really good, like *X was esteemed and valued* and *X was really innocent and ignorant*

Our system is able to find similar words, like gone/departed, fallen/dead, defiled/unclean, just/right, and empty/void, but this is nothing that a thesaurus couldn't tell us. Analyzing the text of the bible, we found it gets a ton of numbers with "and"s in them and says that one number entails another. We could write rules for such annoyances, but two major issues always remain if we stay on the surface. One, it's really hard to get phrases with multiple arguments and see how they change, and two, we can't just connect strings of word because they might not be in the same tree (like making "eclipsed by Mr. Bingly" and "Netherfield" separate phrases): we need to make syntactic trees so we can keep constituent parts together. Thus begins our journey below the surface.

3 Our Algorithm: EMTAIL

The EMTAIL algorithm relies on the assumption that sentences (and phrases within sentences) that appear near each other are related, and pairs of nearby phrases that appear together often are likely to be together because there is an entailment. In a nutshell, the algorithm goes through every pair of nearby phrases, sees what they have in common, and records an entailment for the pair if there are any common arguments. If we have a sentence that looks like the antecedent of an entailment we've recorded (with any arguments), we say the consequent is entailed, substituting in the arguments from the sentence.

Although it seems wrong that nearby phrases must be related, the strength of the entailment plays an important role. If two unrelated phrases appear together once, it won't be considered an entailment if there is a lot of other data. However, if one phrase only appears when it's near another phrase, it is likely that they are related, and we can infer that the first phrase entails the second phrase.

In order to train our system, we need large blocks of continuous text, because we want to find sentences that are near each other and the relationships between their trees. As a preprocessing step, we break up the text into sentences, and we generate a dependency tree for each sentence

(using Minipar, which may be relevant to how we deal with trees). There are then three steps to the algorithm:

3.1 Pruning and Simplifying Trees

The first thing we do is ignore sentences with more than 50 dependency nodes, because these sentences are too complex or ill-formed for us to get meaningful data from. We also throw out certain trees that we know are mis-parsed, such as any trees with quotation marks (the parser we use doesn't do these correctly).

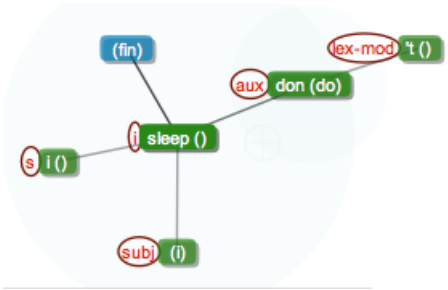
Simplifying the tree is one of the most computationally intensive parts of the algorithm. The idea is to get rid of words that are unnecessary, and bring the important words to the top. Later in the algorithm, we need the root to be empty, the children of the root to be head verbs (or properties) and compose the sentence, and the children of the head verbs to be the important arguments of the verb. Anything below that is unused, so the job of simplification is to make sure the right words are in the right places.

The parser we use already takes care of separating different clauses of the sentence into different trees, so "Mike sleeps and Mike dreams" yields two separate trees, which is perfect for our purposes. There are currently 5 parts to the simplification process, which is applied to each clause independently:

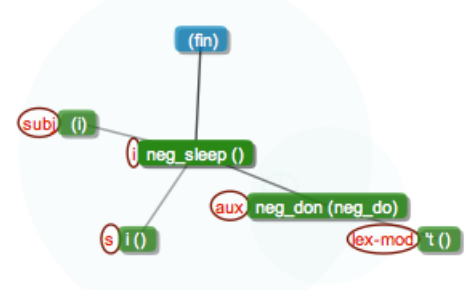
3.1.1 Propagating Negations Up To Verbs

The first stage of simplification does not reduce the number of nodes in the tree, instead it moves information about negation into verb nodes, preparing other nodes for later deletion (for example: "He never runs" will become "He (never) neg_runs" and (never) will subsequently be removed). This process begins by traversing the tree from the top. It then detects if the immediate node is negative, by checking if Minipar has labeled it "neg" or if it matches a number of predefined negative words (never, almost, not, etc...). Once the polarity of the current node has been determined, we check if any of its children are negative (which also repeats the labeling process on them).

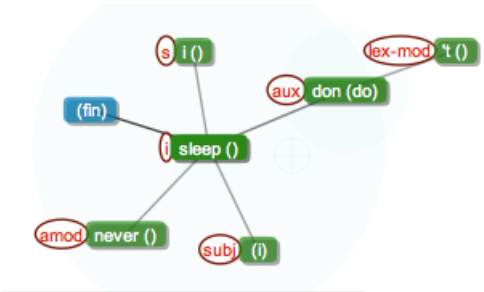
For each child node that is negative, the polarity of the current node is reversed. Therefore, if the current node is negative, and it has a negative child, we have a double negative, so the negations cancel each other out. Also, if two child nodes are negative they will cancel each other out. This allows for fairly robust handling of negations within a single clause. Despite this, there are some problems with our algorithm on very long sentences. In these cases, a negation far down the tree, in an unrelated relative clause, will negate a higher node. To alleviate part of this problem, we decided to discount child nodes that are relative clauses. We also implemented a few special cases, where words that appear negative to our algorithm are actually not (cases like "not only").



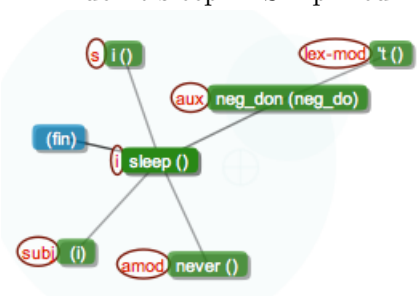
"I don't sleep" - Unsimplified



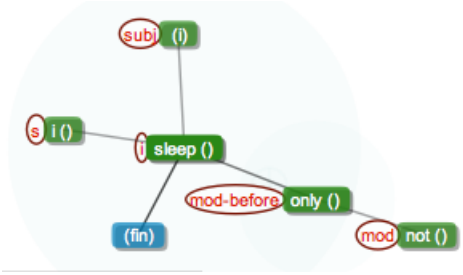
"I don't sleep" - Simplified



"I don't never sleep" - Unsimplified



"I don't never sleep" - Simplified



"I not only sleep"

3.1.2 Collapsing Useless Nodes

Often, there are many nodes that get between the verb and the argument we want, such as helping verbs and words. In the case of "He likes to run," we want to get the "likes" verb with arguments of "he" and "run" (which will make it equivalent to "He likes running" since we use the base word). The tree from Minipar puts an INF node above "run", which we collapse. Also, we collapse the nodes which add no information so should not be arguments, like the auxiliary "do" verb in the case of "I do run", where we want the only argument of "run" to be "I".

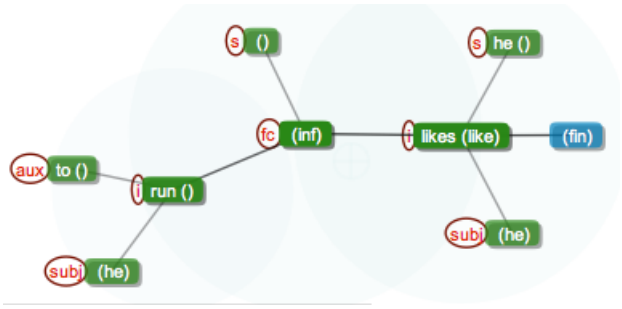


Figure 1: "He likes to run" - Unsimplified

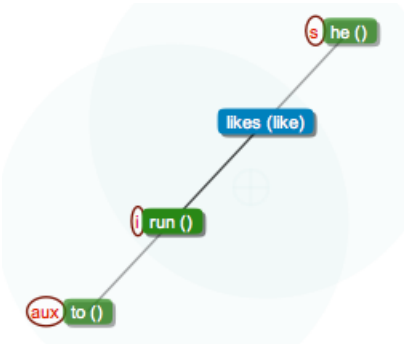


Figure 2: "He likes to run" - Simplified

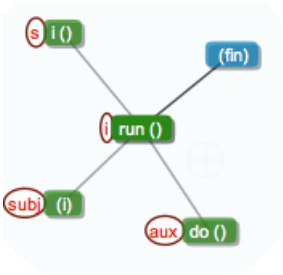


Figure 3: "I do run" - Unsimplified

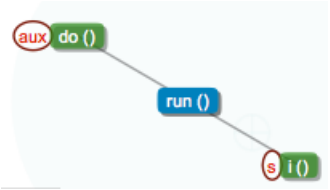


Figure 4: "I do run" - Simplified

3.1.3 Simple Anaphora

We don't have real anaphora resolution, which is unfortunate because it would help us identify many more arguments with one another rather than just relying on the head word, we replace reflexive pronouns with their non-reflexive counterparts so that "I hurt myself" results in "X hurt X" instead of "X hurt Y". We can do this for nominitive and accusative pronouns too, but this hurts us when we have sentences like "he told him..." where the two pronouns are not coindexed. However, the benefits of this outweigh the downsides, so we decided to replace all pronouns.

3.1.4 Fixing Copula-Headed Sentences

We collapse the copula "to be" verbs like other helping verbs, but it has to be dealt with differently. The copula has two children—one subject and one predicate—and if we collapsed it like other helping words, we would end up with its two children as unconnected subtrees of the sentence. Instead, what we do is replace the copula with the predicate and bring up its children. For example, the sentence "I am really good" goes from BE(I)(GOOD(REALLY)) to GOOD(I)(REALLY), which makes it easier to compare copula sentences with different predicates.

Figure 5: "I am really good" - Unsimplified

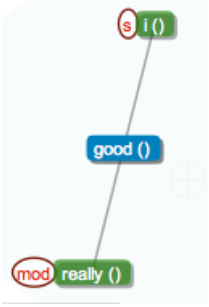


Figure 6: "I am really good" - Simplified

3.1.5 Making Prepositions Relations Instead of Nodes

When a verb phrase has a prepositional phrase, it makes the preposition the child of the verb, and the object of the preposition the child of the preposition. What we want is for the object of the preposition to be the child of the verb, where the particular preposition is specified as part of the relation, so "I read about it" will allow us to identify "it" as the about-argument of "read". Every time we have a preposition, we replace it with its child, and change the relation to the verb from "pred" to "pred_about" (or whatever the particular preposition is).

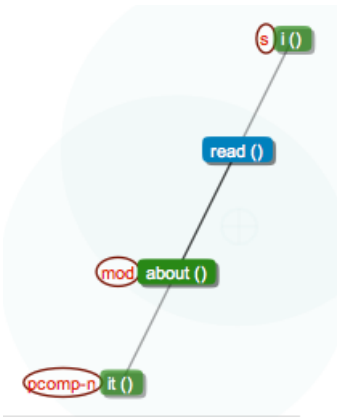


Figure 7: "I read about it" - Unsimplified

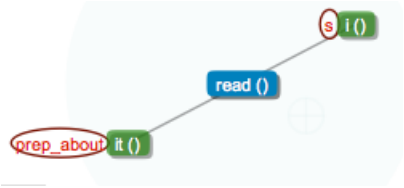


Figure 8: "I read about it" - Simplified

3.2 Finding Entailed Phrases with Common Arguments

At this point, we have a list of sequential sentences, where each sentence has a list of clauses, where each clause has a head (a verb or a predicate of a copula) and a list of arguments and their relations to the head. We identify each head and argument with its base word (so that things like plurals and conjugations are ignored). Then, for every clause in every sentence, we link it up with the other clauses in the sentence, and every clause in the n previous sentences, often stopping before n for paragraphs for document changes (thus giving us links to the nearby sentences).

The linking procedure goes through every possible pair of arguments (one argument from each clause), and if they are equal, adds a link from the relation of the first argument to the relation of the second argument. So if my two clauses are "Jane bought a computer from Mike" and "Mike sold a computer to Jane", there will be three links: subj → pred_to, obj → obj, and pred_from → subj. Sometimes there will be remaining arguments, and we store these as constants along with the head. So if my two clauses are "Jane bought a computer from Mike" and "Mike sold a computer", we'll only have the two links subj → pred_to and obj → obj, and the argument subj=Jane is left over as a constant. Once we've found the common links and the constants left over, we increment the count of this entailment

One alternative method is that for every combination of constants left over, we store an entailment with that combination. By storing all of the constants, we get "Jane bought Y from Z" entailing "Z sold Y" when we really wanted a wildcard on Jane. The problem with this approach is that storing entailments without all of the constants leaves out a lot of information, and in practice we get a lot more incorrect entailments. For example, if we have "I made money" and "I am happy", if we left out constants like "money", we'd get the entailment "X made" to "X is happy".

3.3 Normalizing and Higher Order Entailments

Once we've counted all of the entailments, we remove entailments with one argument that occur infrequently, because with sufficient amounts of data, anything occuring infrequently isn't a general entailment. We then normalize the counts (conditioned on the first word + constants) to take into account the fact the frequently occuring words occur with lots of things, but the probability of entailment is divided among the other words. So if we have "he speaks" occuring with everything,

it only entails each individual thing with a small probability, but the individual things each entail "he speaks" (because everything entails something that's always true).

We also create higher-order entailments, with the intuition that if A entails B and B entails C, then A entails C. For every word w1, for every word w2 such that w1 entails w2, for every word w3 such that w2 entails, we create an entailment from w1 to w3. We only add an entailment from w1 to w3 if all of the links from w1 match up, so for every link from w1 to w2, the relation in w2 must exist in a link from w2 to w3, or else we'll end up with wildcards and get bad entailments like "X made" to "X is happy" (so we'd never make a higher order link out of "X likes money" to "X makes money" to "X is happy" because there aren't as many variable arguments in w1 as there are in w3). We set the count of the higher order entailment at the minimum of the two lower order counts, because if w1 appears with w2 1000 times and w2 appears with w3 1 time, then intuitively w1 could only entail w3 in that one case. An alternative and more correct way to calculate the count is to multiply the normalized probabilities, but since we don't have anywhere near full coverage, the normalized probabilities aren't really probabilities, and are more like scores from 0 to 1, normalized so that frequently occuring phrases get the right score.

We create higher-order entailments in separate data structures so that we can use them to build on each other. Combining entailments from the first-order set with the second-order set yields a third-order set (it doesn't matter which ones comes first, the result is the same).

The reason we implemented higher-order entailments was because we hypothesized that people would restate things in different ways in different places, but in the same contexts. Someone might not ever say "X buys Y from Z" in close proximity to "Z sells Y to X", but it's quite likely that they'll say the former with something that entails another thing that appears near the latter.

3.4 Querying for Entailments

At this point, we have finished training, and we can query for the entailments of an arbitrary sentence. We parse the sentence and run the simplification process on the tree. Taking the head word of each clause, we try every combination of arguments as constant or variable (using the powerset of the argument set), and look look through each set of entailments (lower order and higher order) to see there are any clauses that match up in links. Any clauses that match up get added to the resulting set of entailments, replacing each variable argument with the linked argument.

4 Implementation

4.1 Obtaining Continuous Data

To get continuous data, we started by downloading books from Project Gutenberg to test on. We decided, however, that books tended to have too many quotes and figurative language, and the large corpora of news articles we have access to talked about topics that were too narrow (we'd rather find relationships between verbs that describe people than financial transactions). We decided that blog posts would be ideal, since there is an unlimited supply, and many of them are just describing common events ("Today I bought. . . "). Since Google provides a convenient API to accessing posts and comments from Python, we used its libraries to write a blog crawler. It did not, however, allow us to search for blogs, so we wrote a Google searcher that finds as many unique blogs on BlogSpot

as possible (which is tricky when they only allow access to the first 2,000 search results for every query).

The program we wrote is in bloggerpy/run.py (and all other files in that directory are third-party libraries we used to access the Google API). Running with the option --findnames ⟨n⟩ finds as many as n blogs and records their names in blognames.data. Running with the option --getblogs ⟨n⟩ downloads all of the blogs mentioned in blognames.data and saves every single post and comment to posts⟨n⟩.data and comments⟨n⟩.data respectively, where the blogs are divided up into ten partitions from 0-9 (to allow us to run our script on 10 machines and have 10 segments of blog data). We remove all HTML and blog tags, and replace annoying curly quotes, curly apostrophes, long dashes, and ellipsis characters with their ASCII equivalent. Any blog posts with other non-ASCII characters we deem to be non-English so we throw it out. Running with the option --save ⟨n⟩ takes the data from posts⟨n⟩.data and comments⟨n⟩.data and writes it to posts⟨n⟩.txt and comments⟨n⟩.txt as plain text, where posts are delimited by double newlines and new blogs are delimited by triple newlines.

We ended up downloading 8033 blogs, giving us 2 million sentences (200MB), which equated to about 2GB of trees (explained below).

4.2 Segmenting and Parsing Sentences

To divide blog data into sentences, we used a third-party Perl sentence segmenter [2], found in the sentenceboundary directory. We then fed the sentences into Minipar [3] to generate dependency trees, and used the supplied pdemo program to get the tree in a plain text format we could read in with our programs not written in C.

4.3 Getting Surface Data

Our surface data program is in the java folder and can be run with fp.GetSurfaceData. It reads every .txt file in the folder specified by the first command line argument (a data folder with sentence-segmented files), gets surface structure entailments from each file and prints them as it goes along. When it gets to the end, it creates second-order and third-order entailments, and prints every possible entailment and its conditional-probability-like score.

4.4 Getting Tree Entailments: Initial Version

The initial version of our EMTAIL algorithm was written in Java, and can be run with fp.GetTreeData. It reads every .txt file in the folder specified by the first command line argument (which should be Minipar trees). TreeProximityEvaluator iterates through the trees (using DependencyReader to parse the Minipar output), and adds trees with common arguments to the TreeProximityCounter, which stores the counts of pairs of head words and their TreeLinks (with a CounterMap structure from w1 to w2/links pair). DependencyReader takes care of simplifying the tree before it is returned from the iterator. At the end, various debugging information is outputted depending on what lines we uncomment. This version is not complete, because we rewrote it from scratch in Python:

[2] Obtainable from http://l2r.cs.uiuc.edu/ cogcomp/atool.php?tkey=SS

[3] Obtainable from http://www.cs.ualberta.ca/ lindek/minipar.htm

4.5 Getting Tree Entailments: New Version

Frustrated with the verbosity of Java, we rewrote everything in Python. This allowed us to easily make visualization of trees and didn't seem to cause a large performance hit. Through the process of porting our program to Python, we discovered that using Python sped up our development process, and that by using the Psyco optimizer, we experienced no noticeable performance loss.

4.5.1 draw.py

Using the NodeBox [4] program to display graphics (and the associated graph and boost files in the python directory), we wrote a program using NodeBox to graph the dependency trees so that we could see what we needed to do in our simplification process and how it was working. The draw.py takes an arbitrary sentence, queries the Minipar CGI interface we wrote (we can't install Minipar on our development machines), and displays the dependency structure either before or after the simplification process. Examples of it are shown in our description of simplifying above.

4.5.2 minipar.py

This module provides TreeFileIterator and parse_tree to get hierarchical tree structures (with the Node class) directly from Minipar output files. Trees are returned as one root node (which doesn't exist in the original Minipar tree) which contains all of the head nodes of the clauses in the sentence. The find_negations function propagates negations in a tree and the simplify_tree function does the rest of the simplification process. Running minipar.py by itself goes into an interactive mode where it inputs sentences, uses the CGI Minipar interface to parse it, and displays both the original tree and the simplified tree.

4.5.3 proximitycounter.py

This module provides ProximityCounter, which stores entailments, and provides pruning and normalizing methods. Since we run multi-threaded, we need a way to combine multiple ProximityCounters that have been generated from different files, so the extend method adds everything from the second counter to the first counter. The connect method generates higher order counters from two lower order counters. ProximityCounterSet keeps a list of ProximityCounters and their coefficients and does a linear interpolation of their scores for requested entailments.

4.5.4 trial.py

This is where the core of the algorithm runs, and some of the options are configurable at the top of this file. When running trial.py, run() gets called, which creates a ProximityCounter with the entailments in all of the files in the tree data directory. It then displays the entailments it found. If the get_entailments_from_file line is uncommented, it goes through one of the files and displays all of the entailments it gets from each sentence. If the -i command line argument is given, it runs in interactive mode and allows the user to type in a sentence and displays everything the sentence entails. Most of the work is done in get_proximities_from_file, which uses add_links to find the common arguments between any two clauses.

[4] Obtainable at http://nodebox.net

5 Error Analysis

5.1 Scoring Method

Scoring was a major problem for us, as we could see no easy way to automatically calculate whether an entailment is right or wrong. Therefore, we created a scoring metric called the uScore, so-named because you (or I) manually scores the entailments generated by 50,000 trees (around 100-200 entailments). These entailments are scored on a scale of 0-2, with 0 being completely wrong, 1 being a likely entailment given the current context, and 2 being a correct entailment that applies in general. We then calculate an aggregate uScore by calculating

$$\frac{0.5(\sum score1) + 1.0(\sum score2)}{\sum score0 + \sum score1 + \sum score2}$$

This gives us a metric that rewards the program for generating a higher percentage of correct entailments.

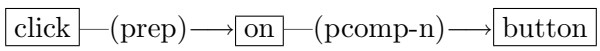
Although we designed the uScore metric, and we used that metric to interpret our final results, we used a slightly simpler metric during the error analysis phase. As the number of entailments either increased or stayed relatively constant, we simply used the percentage of generated entailments that were correct as our scoring method during error analysis, where "correct" corresponds to the sum of score1 and score2.

5.2 Initial Results

After implementing our baseline feature set, we noticed that our performance was fairly bad. In total, we were only able to generate 73 entailments from 50,000 sentences, and of these 73 entailments, only 34.2% were correct.

5.3 Prepositional Phrase Collapsing

After examining the trees, we noticed that sentences such as "click on the button" resulted in the tree:



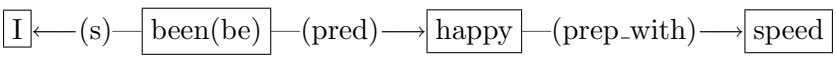
This meant that "button" was effectively thrown out, as it was two links away from the verb. This result was not what we wanted, so we decided to try collapsing prepositional phrase nodes and replacing them with their children. Using this technique, the above sentence would be collapsed to:

In order to preserve the type of preposition being collapsed, the preposition is added to the relation between the verb and the object of the preposition.

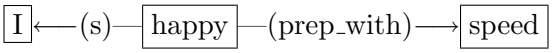
By using this technique, our results improved significantly, going from around 34.2% correct to 39.3% correct. Additionally, we generated 49 more entailments using this technique, which increased our entailments by 40%. Therefore this modification was a definite win for us.

5.4 "be" Verb Collapsing

Another problem we noticed was that sentences such as "I've been really happy with my speed during these rides..." get turned into trees like:



This resulted in "speed" (and many other things) being two links away from the verb, and not able to be used. In order to fix this, we decided to replace "be" verbs with a child node if there was a predicating child. This resulted in the above tree being corrected to:



This allowed us to match more phrases, as the relatively meaningless "be" verbs were replaced with more meaningful verbs. In the above example, the sentence could now be linked to the nearby sentence: "...lets me play with speeds I may not see otherwise...", as a link could now be formed between "speeds" in both sentences.

By using this modification, we saw another good improvement from the previous level of 39% to 43% correct. Additionally, the number of entailments generated increased by 16%.

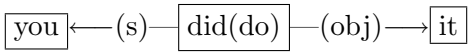
5.5 Handling Negations

At this point, we still were not handling any negations, and a lot of sentences were turning out false because a verb was negated, but we did not include this information in our links between verbs. Therefore, we implemented the negation handling scenario which is described in detail in the previous implementation section.

Unfortunately, fixing negations did not provide the result we were hoping for, lowering our percent correct from 43% to 42%. Normally, we would have removed this change, however in almost every case we looked at, the negations were correct interpretations of the sentences, it just happened that the sentences being dealt with did not entail each other even though they occurred nearby. Therefore, we decided to keep the negations, as they did very little harm, and it seemed likely to us that the harm that was caused was just due to unlucky data as opposed to a problem with our method of handling negations.

5.6 Better Handling of "do"

The next thing we noticed was that the verb "do", which occurs incredibly frequently was generating odd results in a number of places. For example, the sentence "and you did it anyway." was generating the tree:



While this tree is technically correct, the verb "do", when it occurs without any other verb phrases beneath it, is almost without meaning. Therefore we decided to collapse any "do" node encountered. This removed a large number of vacuous phrases which were adding entailments such as:
Y do X $\longrightarrow$ Y pickup X

and

Y do X anyway $\longrightarrow$ Y do X because

Removing these phrases improved our results only slightly, raising them from 42% correct to 42.5%. While this was not a major improvement, we felt our reasoning was sound, and it did not hurt us, so we kept it.

5.7 Quotations

The next thing we noticed was that a large number of entailments seemed to be wrong because Minipar does not seem to handle quotations within a sentence well. Noticing this, we tried not processing any sentence which had a quotation, expecting this to yield an improvement. Sadly, we found that there were so many sentences that had quotations in our data (drawn from blogs), that this restriction eliminated far too many good entailments with the bad ones. Therefore we did not end up keeping this change.

One thought that occurred to us as a possible fix for this problem would be to pre-process our data, turning quotations into un-parsable blocks (turning "go to it" into Q_GO_TO_IT_Q). This would hopefully mean that mini-par would not incorporate the quotations into the dependency trees except as single nouns, and we would therefore easily be able to remove them later (or do other processing on them).

One last thought that occurred to us was that, if we were to chose a data source with fewer quotations, we might be able to get away with, and even benefit from, throwing out all sentences which contain quotations.

5.8 Keeping More Time Related Information

Another thing we noticed was that many sentences were wrong because too much temporal information was thrown out. Therefore, we decided to try keeping certain verb modifiers, if they seemed like they would add important information. The modifiers we chose to keep were "before", "after", "before" and "earlier". We hoped that this would help fix situations such as "I forgive him" and "He hurt me earlier", where X forgives Y does not imply Y hurts X, however it does imply Y hurts X earlier.

In relation to this, it occurred to us that it would be useful to not only store time-related words, but to attempt to determine the tense of the verbs. This could be used to order the direction of entailments. For instance, currently, we assume that any sentence in our data is entailed by the sentences before it. However, this quickly breaks down in cases like: "I am playing with my new videogame" followed by "I bought the videogame yesterday". If we determined the tense of verbs, we could re-order the entailment so that the earlier event implied the later one, regardless of the order they appear in. Unfortunately, while we thought this might help a lot, we did not have time to implement it.

5.9 Different Proximities

Originally, we were only considering sentences as possible candidates for entailment if they were within 2 sentences of each other. This was chosen fairly arbitrarily, so we decided to try raising it, to see if we could generate more results.

We tried using both 3 and 4 sentence proximities, and while 3 helped significantly, four seemed to significantly hurt our results. With three sentences proximities we were able to get up to 48% correct entailments, with 25% more results. This seemed incredibly beneficial, so we kept this change. The variation in results was in fact so great with small changes in proximity that we have included a more detailed analysis of the effects of proximity in our Results section.

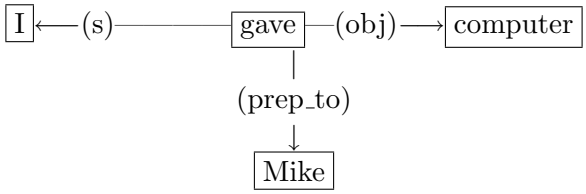
5.10 Fixing Variable Errors

At this point we noticed that we were not allowing the objects of transitive verbs to be linked. This was a major bug, and fixing this resulted in 52% correct entailments with 20% more results.

5.11 More Prepositional Raising

At this point we noticed that Minipar had (what seemed to us) an odd way of parsing many common sentences. For a sentence such as: "I gave a computer to Mike", we would get the following tree:

This seemed counterintuitive, as it does not treat "to Mike" as an argument of the verb, but instead treats it as a modifier of "computer". This prevented us from turning "Mike" into a variable. Therefore, we tried raising every preposition that modified a noun to the same level as the noun, resulting in the structure:



This fixed some problems, but caused many more, as many prepositional phrases should not be raised in this manner (consider "I saw the computer from Japan", in this case, "from Japan" should modify computer and should not be raised). To try to resolve this, we decided to only raise a preposition if it was "to" or "from", which fixes the most common cases while minimizing errors in others.

This ended up giving us a very minor gain, resulting in 0.6%.

5.12 Resolving Problems with Existential It

One final problem we ran into was that the existential "it" was being considered a subject in many cases, and when combined with the fact that we collapse "be" verbs, this was creating incorrect situations. For instance "It is nice to see them" treated "it" as the subject of "nice", which allowed it to combine with other sentences in which "it" is not existential.

In order to prevent this, we initially tried stopping "it" from forming links in all cases, however this lowered our percent correct very slightly. Therefore, we considered the other possibility of just preventing "it" from linking when it is in the subject position (the most likely position for an existential). This ended up working rather well, giving us 53.6% correct entailments.

5.13 Error Conclusions

We ended up finding that a large number of our errors resulted from Minipar parsing sentences in what seemed like unlikely ways. Additionally, many of our errors came from sentences with very complicated and long structures, in which too much information was discarded in the process of simlifying arguments, at which time the processing of these nodes became error-prone. For example, we were presented with the following two sentences in close proximity:

1. and when they couldn 't take that anymore , they voted the white cats out and put the black ones in again

2. they even tried half black cats and half white cats

The entailment we found was that "Y votes out X" entails "Y tries X", where Y was "they" and X was "cat". Clearly this is a problem, because the head "cat" refers to both "white cats" and "black cats", which must be distinguished to relate these sentences. With better rules for collapsing and discarding data, we could probably do much better. Also, with a more consistent source of data, such as an encyclopedia, we would probably also perform better.

We are especially interested in trying different data sources, as when we tried parsing sources written in a consistent manner (we tried Pride and Prejudice and the Bible), we seemed to get better results. The downside to sources like these is that the data retrieved from Pride and Prejudice or the Bible would not be relevant to many NLP tasks, as it is not written in everyday speech.

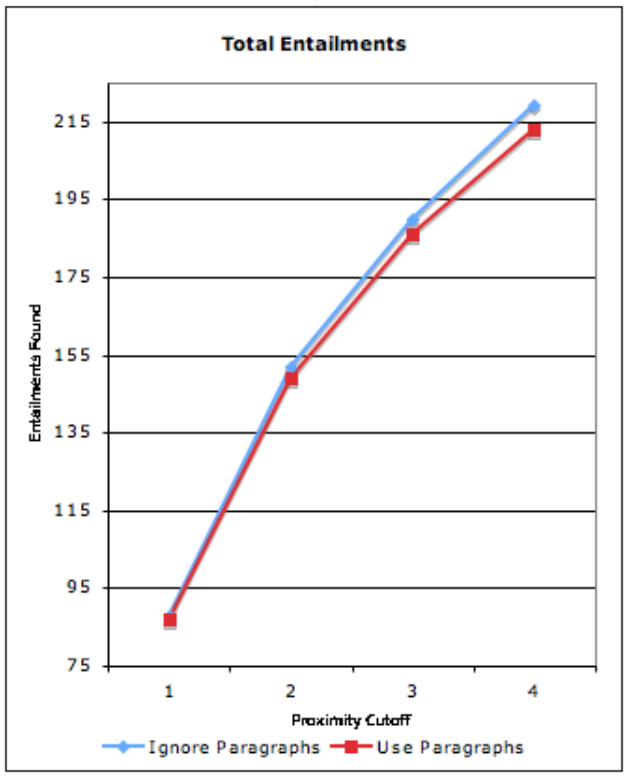
6 Results

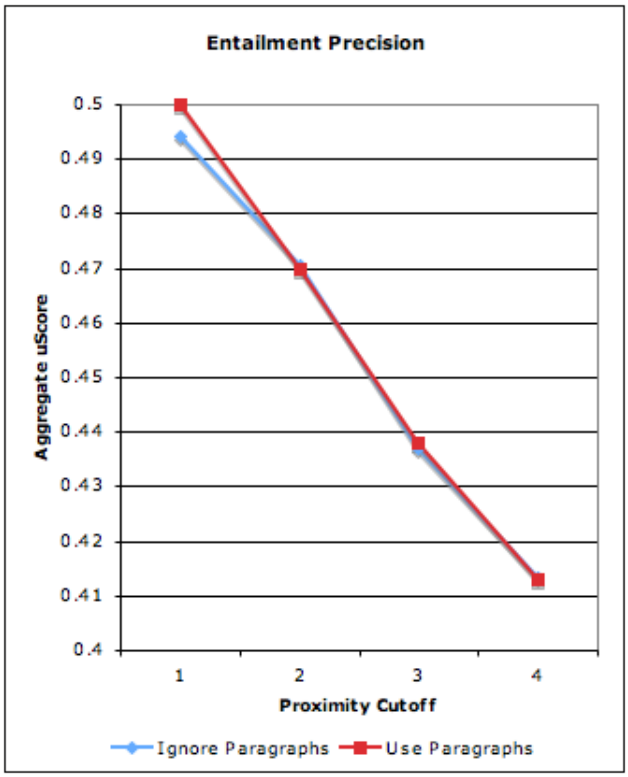
6.1 Accuracy and Precision

To get some sort of idea of the strength of our system, we ran EMTAIL on data we had not developed on (a different set of blogs), and hand-scored the results with uScores. Because of the time consuming nature of calculating uScores, we only experimented with two parameters: proximity cutoff (how many sentences can two clauses be apart for them to have an entailment) and a flag for whether sentences in different paragraphs could entail each other. Other parameters that we tweaked, but could not go through the scoring process for, were the count cutoffs for whether an entailment is recorded, whether sentences from different blog posts could entail each other, which simplification techniques we used, which arguments we kept as constants, variables, or neither, and what sort of data we trained on.

Below are the results for the number of entailments for each parameter, and the "aggregate uScore" for each parameter, which is calculated by taking the sum of the score-2 entailments, half the sum of the score-1 entailments, and dividing by the number of total entailments. This is effectively a precision measure, because if every entailment we find is correct, we get 1.0, if none are good at all, we get 0.0, and depending on how good the entailments are that we find, we get something in between. The number of entailments we find is effectively an accuracy measure, because for the entire data set, there exists a complete set of entailments that are possible for us to find, and if we knew the number of those, dividing total entailments we found by that number would yield accuracy. Thus, we are graphing a linear scale of accuracy. (Of course, there are technically infinitely many entailments, but the general idea still holds that the more correct entailments we find, the

better coverage we have.)





Since the score-1 results are generally correct entailments given the context of the document, we consider them as successful results of our system. With that intuition, the best precision we reached was 72.4% 1's and 2's, using a proximity cutoff of 1 and removing cross-paragraph entailments. Since this score is consistent with the true/false measure we used in error analysis (where true corresponded to 1's and 2's), we do even better on the new data than we did on the data we developed on. Even when we hold the proximity and paragraph-flag constant, we get 63% 1's and 2's, compared to 53% in the development data set. This shows two things: one, we aren't getting ahead by writing our specific parameters and techniques for specific data, and two, either the data or the scoring technique isn't very consistent.

6.2 Example Entailments

We took note of a few entailments that stood out when we scored the new data set (paraphrasing the link structure). Ternary relations were pretty rare unfortunately.

1. "X feels Y" → "Y sweeps over X"
2. "X is great friend to Y" → "X helps Y"
3. "I give Y X" → "Y gets X"
4. "Y keeps an eye on X" → "Y sees X"
5. "Y learns secret from X" → "X tells Y"
6. "Y didn't read X in school" → "Y is curious about X"

7. "X misses Y" → "X loves Y very much"
8. "Y shows off X" → "Y takes X everywhere"
9. "Y uses X" → "Y has X"
10. "Y writes X" → "X gets message from Y"

Of course, only about 27% of our entailments were this good (uScore=2) with our best parameters. Many of our entailments were pretty bad. For example, we read "he missed 123 games" and "he played 145 games" in close proximity, and one can guess what kind of contradiction we obtain.

6.3 Higher Order Entailments

In addition to the normal entailments we generated, we also tested generating second order entailments. These make use of the transitive property we assume holds between entailments: if X implies Y and Y implies Z then X implies Z. This technique worked relatively well, however, as would be expected, it was dependent on the quality of the first order results. We did however get a few good results, such as:

X enjoys Y $\longrightarrow$ X looks for Y
and
X looks for Y $\longrightarrow$ Y excites X
Therefore
X enjoys Y $\longrightarrow$ Y excites X

We also got: Y stays up with X $\longrightarrow$ X asks Y
and
X asks Y $\longrightarrow$ X wakes up Y
Therefore:
Y stays up with X $\longrightarrow$ X wakes up Y

These results appear to indicate that our higher order entailments work to some extent. Also, as we are able to improve the accuracy of our first order entailments, we will generate more meaningful second order entailments.