

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.863J/9.611J, Natural Language Processing
Laboratory 3: Advanced Parsing & Lexical Semantics

Handed out: March 20, 2009

Due: April 6, 2009

Goals of the Laboratory. Laboratory 2 and Competitive Grammar Writing introduced you to probabilistic context-free parsing. In this laboratory you will learn how well state-of-the-art statistical context-free parsers work – their strengths and weaknesses.

In Part I, you will learn about the following:

1. An introduction to current state-of-the-art probabilistic parsing. These systems use machine learning methods to acquire their own rules, trained on a large number of pre-parsed sentences, using a more sophisticated method than the one you used in Laboratory 2. After your own work on grammar writing, you might appreciate the effort that such machine learning saves. In the first few questions we'll have you do some basic warm-up in running these systems, so as to get used to the particular interfaces we have set up for them.
2. What are the strengths and weaknesses of current state-of-the-art statistical natural language parsers? If they were perfect, we'd be done, at least for the syntax part of natural language processing. But the parsers are not perfect. We will look at ambiguity (again). One of the things modern statistical parsers do better is to add information about particular words in order to figure out how to prune parsing possibilities. Classic examples would be those such as *I saw the guy on the hill with the telescope*. Does *with the telescope* associate more strongly with *the hill* or *saw*?

Part II will give deeper understanding of how statistical parsers work and how they interact with, lexical frequencies, syntactic, and semantic regularities. In particular, you will:

1. Investigate the connection between lexical (word level) semantics and parsing, using the Penn Tree Bank (PTB) as a concrete test bed.
2. Explore how a state-of-the-art probabilistic parser will handle these issues.

What you must turn in. You will need to turn in two (or more) report files.

1. a writeup of Part I: `lab3a.pdf`
2. one or more reports for each verb for part II as `[your verb].pdf`.

As before, please email your write-ups as pdf files to `6.863-graders@mit.edu`. You may use the write-up templates provided here:

<http://web.mit.edu/6.863/spring2009/writeups/lab3/>

Please rename `verbN.pdf` to `[your verb].pdf` when you submit. You will be assigned multiple verbs to analyze, you are required to report on one. Extra reports you submit will count as extra credit. In your email include **6.863 Lab 2** as your "Subject:". As usual, you may collaborate with whomever you wish; just note the names of your collaborators in your report. Your report should be recognizably your own work.

Part I

Initial Preparation:

- **Background reading:** Please read our semi-Google books version of Chapter 14 of the 2nd edition JM text, here:
<http://www.mit.edu/~6.863/spring2009/jmnew/ch14.pdf>
- **Software:** You will need to run the tools in this lab either on Athena, by logging in to `linux.mit.edu` and setting up your environment properly, as described in Laboratories 1 and 2. Alternatively, you can download the java program to run the parsing tools here, along with installation instructions:
<http://web.mit.edu/~6.863/spring2009/code/lab3.zip>

1 Using modern probabilistic parsers

1.1 Running statistical parsers on Athena

You created a simple probabilistic parser using NLTK in Laboratory 2. For this lab, on Athena we've set up installations of two widely-used statistically-based parsers on Athena. Each is pre-trained on the Wall Street Journal (WSJ) section of the Penn Treebank (PTB), a collection of approximately one million words of running text from, ah, the Wall Street Journal, which was then converted, partially by hand, into parse trees.¹

Such parsers require their input to be *tagged* – their rules do not go down to the level of individual words, only their parts of speech (POS). These part of speech names, which you first saw in Lab 2, are partly an historical artifact, derived and elaborated from much older corpus work first done at Brown University in the 1960s. For example, the tag IN denotes any Preposition, while VBD stands for a past-tense verb, sometimes ending in *ed* or *en*, such as *taken* (but VBD could also be the tag associated with an irregular past tense verb such as *sung*). A list of 48 of the most important tags used for the Penn Tree Bank is given just below. You'll soon grow to know and love or hate this list. (For example, what information do these tags include? What information do they omit?)

<http://bulba.sdsu.edu/jeanette/thesis/PennTags.html>

Since the sentences don't come with their tags on them, and since the same word can have more than one tag, parsing systems must pre-process sentences and assign tags to each word. We will be using these parsers with the MXPOST maximum entropy tagger, which combines information about the endings of words plus some local context, such as the preceding word tag, and combines this information probabilistically to figure out the most likely tag to assign to the word it is currently looking at. We will have more to say about this tagging method in lecture, but for now if you want you can download it yourself and read about it here:

http://www.inf.ed.ac.uk/resources/nlp/local_doc/MXPOST.html

and read about the method via the following paper as described there:

http://www.mit.edu/~6.863/spring2009/readings/mxpost_doc.pdf

1.1.1 The Java installation for the Bikel-Collins & Stanford parsers

For a warm-up introduction to these parsers, we will use a java-based “wrapper” implementation that lets you switch between either the Bikel-Collins or the Stanford statistical parsers, and automatically loads in the entire training set of Wall Street Journal sentences. (It also lets you load in the entire test set of Wall Street Journal sentences, and has other useful functionality that we'll run through as we go, see below.)

¹As is standard, specifically this means the systems are trained on sections 02 to 21 of the Wall Street Journal, approximately 40,000 parsed sentences.

This parser wrapper comes packaged as a .jar file `parsers.jar` so it should be possible to install and run on your local linux, Windoze, or (with more difficulty) Mac OS X machine. See the `readme` file for how to do this, here:

`http://web.mit.edu/~6.863/spring2009/code/lab3/parsers-install.txt`

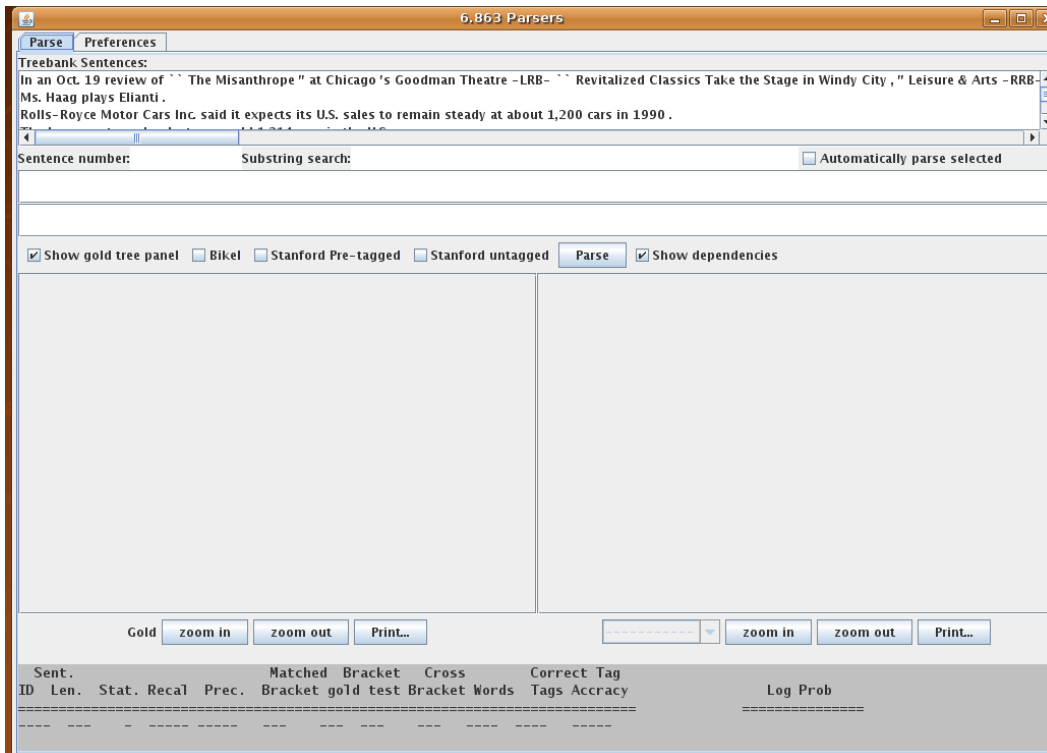
To run the program on Athena, `ssh` using X-windows forwarding:

```
ssh -Y mylogin@linux.mit.edu
```

and make sure you have set up your 6.863 environment properly in the usual way as in Labs 1 and 2. Then do:

```
java -Xmx800M -jar parsers.jar
```

The first time you run this, the program will add a directory of resources to your home dir, which will take about a minute or so (you will see the loading messages on the terminal console). This will make subsequent parsing very fast. After it has finished this, the program will pull up the following window back on your machine, with all the training sentences from the Wall Street Journal pre-loaded. You can select any of them to parse, or you can type new sentences in. Let's give a brief overview of what the layout of this gui, and what you can do with it.



Here's how the gui layout divides up. Top to bottom, there are roughly seven regions to note:

1. **Treebank sentences:** An area that displays sentences loaded into the system to select in parse. Initially, this is set to the training sentences of the PTB for these parsers, sections 02-22 of the Wall Street Journal treebank, one sentence per line. So you'll see the first sentence *In an Oct. 19 review of "The Misanthrope" at Chicago...*, followed by a few others that you can see. To see more, use the scroll bar on the left. You select a sentence to parse by click-selecting it. If the "Automatically parse selected" box is checked, then this will, surprise, automatically parse the selected sentence without you having to do anything else.

2. **Sentence number** and **Substring search**: Next, there are two boxes that let you select a particular sentence number in the corpus (e.g., sentence 9), rather than selecting by highlighting a sentence. Substring search lets you look for a particular substring of a sentence, e.g., the word *remain*.
3. **Sentence input region**: The third region of the gui provides a text type-in area if you want to type in your own sentence instead of selecting an existing treebank sentence. You can do this any time you want, even if sentences are already loaded in above. Please see the remarks immediately below about how sentences must be tokenized – you can get a good idea of this by simply observing how the treebank sentences are tokenized. You have to turn off “Automatically parse selected” to use your own sentences this way.
4. **Tagger output region**: The output as tagged by MXPOST will be produced immediately below the sentence, in the fourth, next panel. This will be in an s-expression format, (`token (TAG)`). If you want, you can correct any incorrect tags here by simply selecting an input point with the cursor and typing. You will see as you gain familiarity with the system that in fact the Bikel parser can *change* the tags that MXPOST provides, usually correcting such mistakes (but not always).
5. **Parser control**: Fifth, we have six horizontally placed buttons that control various aspects of parsing: Show gold tree panel; Bikel; Stanford Pre-tagged; Stanford untagged; Parse; and Show dependencies. These selection boxes operate as you would expect. If you uncheck the “Show gold tree panel” box, then you’ll see only the parser output. Checking/unchecking the “Bikel” and the “Stanford untagged” boxes allows you to select one of the two parsers, either the Bikel parser or the Stanford parser. (You should not really need to use the “Stanford Pre-tagged” box in this Lab, which will by-pass MXPOST.) If you select both parsers, then you can choose between which parser’s output is displayed below by using a pull-down menu later. Proceeding, the “Parse” button tells the system to parse the sentence that is in the input sentence box, if one is not using automatic parsing; this would be used if you typed in your own sentence. Finally, the “Show dependencies” check-box applies to the Stanford parser only. When checked, it will add links between the words at the fringe of the parse tree to display “grammatical” dependencies such as `nsubj`, the noun Subject of a sentence, or `amodnonsbj`, an “adjective modifier” of a noun Subject. See below for an example.
6. **Gold tree and parse tree output**: The region below is divided into left and right panels where the “gold standard” training parse and then the trained parser’s output are displayed. These are scrollable, with zoom-in and zoom-out buttons. The right-hand panel has a pull-down menu for selecting which parse tree to display, either Bikel or Stanford, if you’ve earlier chosen both parsers. The “print” button will bring up a menu selection for printing either the gold standard or the parse trees to a postscript file of your choice (or a printer).
7. **Evalb scoring**: Finally, there is a gray background region that displays the results of running the precision/recall `evalb` program using the gold tree (if there is one) against the parser’s output. The most important numbers here are the Recall and Precision values, along with the negative log probability score for the parse tree (on this scale, closer to 0 is a higher probability, with logs taken to base e). (See the file `readme-evalb.txt` in the code package for additional discussion on the format and details of `evalb`.)

To see an example of what a parse tree looks like, in the snapshot below we have selected the second sentence, *Ms. Haag plays Elianti .*, a sentence with 5 tokens, and parsed it with the Bikel parser. You can see the gold standard, “true” parse on the left, and what the parser produces on the right. Note that the match between the two is exact, so `evalb` returns, as expected, 100% precision and recall values. The log probability score is approximately -23.723 .

The screenshot shows the 6.864 Parsers interface. The sentence being parsed is "Ms. Haag plays Elianti." The parse tree on the left (Gold) shows a root node S branching into NP-SBJ and VP. NP-SBJ branches into NNP (Ms.) and NNP (Haag). VP branches into VBZ (plays) and NP, which further branches into NNP (Elianti). The parse tree on the right (Bikel) shows a root node S branching into NP and VP. NP branches into NNP (Ms.) and NNP (Haag). VP branches into VBZ (plays) and NP, which further branches into NNP (Elianti). The interface includes a table at the bottom with the following data:

Sent. ID	Len.	Stat.	Recal	Prec.	Matched Bracket	Bracket gold test	Cross Bracket Words	Correct Tags	Tag Accracy	Log Prob
3	5	0	100.00	100.00	4	4	4	0	5	100.00

To see what a slightly more complicated example sentence looks like, here is the fourth sentence, *The luxury auto maker last year sold 1,214 cars in the U.S.* ., as parsed by the Stanford system. This also shows you what dependencies look like; they are the labeled arcs from one word to another displayed at the bottom of the parse tree on the right. You should take note of common dependencies such as *det*, that links *the* with the noun *maker*, as well as the *nsubj* or Subject link from the verb *sold* to *maker*, and *dobj*, the “direct object” link from the verb to the NP object *1,214 cars*. Note finally that the log prob score is much lower, in part because the sentence is longer. The *evalb* score is perfect, since again the parser has done a perfect job of parsing the sentence as compared to the training example.

The screenshot shows the 6.864 Parsers interface. The sentence being parsed is "The luxury auto maker last year sold 1,214 cars in the U.S." The parse tree on the left (Gold) shows a root node S branching into NP-SBJ, NP-TMP, and VP. NP-SBJ branches into DT (The), NN (luxury), NN (auto), and NN (maker). NP-TMP branches into JJ (last) and NN (year). VP branches into VBD (sold), NP (1,214 cars), and PP-LOC (in the U.S.). The parse tree on the right (Stanford untagged) shows a root node S branching into NP, NP, and VP. The first NP branches into DT (The), NN (luxury), NN (auto), and NN (maker). The second NP branches into JJ (last) and NN (year). The VP branches into VBD (sold), NP (1,214 cars), and PP (in the U.S.). The PP branches into IN (in), DT (the), and NNP (U.S.). The interface includes a table at the bottom with the following data:

Sent. ID	Len.	Stat.	Recal	Prec.	Matched Bracket	Bracket gold test	Cross Bracket Words	Correct Tags	Tag Accracy	Log Prob
15	12	0	100.00	100.00	7	7	7	0	12	100.00

When you type your own sentences in for the system to parse, you need to give the parser tokenized input, as described in Laboratory 2 and as shown in the input sentences panel above: There need to be spaces between everything that the parser considers as a separate token, not just where spaces would be used in natural English.

1.2 Basic questions about the statistical parsers

Problem 1.1. First, a question about tagging. The simple NLTK parser you used before in Lab 2 was also assigning parts of speech to words, but it was doing it using the rules it learned such as:

```
VBN -> 'classified' [p=0.0004766]
```

This could be considered a kind of *unigram tagger*, because it does not use any surrounding context to assign parts of speech. The parsers' MXPOST maximum-entropy tagger, on the other hand, uses surrounding context to make its decisions. But there is a bit of information lost at the boundary between MXPOST and the Bikel parser. What information was present in your NLTK parser's part-of-speech assignments that is *not* conveyed between MXPOST and the Bikel parser?

We would expect the trained parser to do well on the sentences it has already seen — the sentences trained on. But does it always do perfectly on training data and return a parse tree that is identical to the ones it was trained on? Using the java-based parser, check the box labeled, “Automatically parse selected,” as well as the box labeled, “Show gold tree panel” and “Bikel”. Leave all the other boxes unchecked (“Stanford Pre-tagged,” “Stanford untagged,” and “Show dependencies”). Now you can step through each sentence of the training data, one by one, to see how the parser does on them. As you select each one, the system will automatically tag and then parse that sentence. The left-hand panel will display the original training tree, while the right-hand panel will show the parser's output. The very first sentence is this one, 49 tokens long:

```
In an Oct. 19 review of ‘ ‘ The Misanthrope ’ ’ at Chicago ’s Goodman  
Theatre -LRB- ‘ ‘Revitalized Classics Take the Stage in Windy City , ’ ’  
Leisure & Arts -RRB- , the role of Celimene , played by Kim Cattrall ,  
was mistakenly attributed to Christina Haag .
```

The parser does perfectly on this first training example and its output tree structure matches the training data. (We put to one side some of the tag details in the original training data for nonterminals like NP-TTL, which denotes an NP that is a ‘title’ that are not meant to be learned by the parser; this is also true of any ‘empty’ nodes in the training data trees that dominate null words, labeled with a 0 – see training sentence number 5 for an example.) Note that If you look at the bottom of the gui panel, which shows the results from the standard precision/recall `evalb` program, it says that precision and recall are both 100%, with 31 brackets matched perfectly.

Scrolling down and selecting, the second sentence in the training data is an easy one for the parser (we showed the output earlier, above), and the `evalb` recall and precision is also 100%.

```
Ms. Haag plays Elianti .
```

Problem 1.2. What is the first sentence in the training set where the (Bikel) parser makes an error with Recall < 100%? Report the sentence number and the actual sentence. (The easy way to find the example is to keep scrolling down the list of training sentences, and as you select each one, it will be automatically parsed. Look for the first sentence where the Recall value at the bottom turn out to be less that 100.00%.) How would you describe the kind of error that the parser makes? Would this error have any effect on the meaning of the sentence? That is, would the error the parser make have an effect on the semantic interpretation of the sentence, as it does in cases where a parser can decide to attach a phrase at once place versus another, as in *I saw the guy on the hill with the telescope*, where *with the telescope* might be part of the NP *the hill* or part of the NP *the guy* or the VP *saw*. Why or why not? (Include a snapshot in your report.)

2 Current issues for statistical parsers

2.1 Assumptions

Statistical parsers must make statistical assumptions. As you have seen from the parsers' output, one of the assumptions is that we can measure the likelihood of sentences by a negative log score. The more negative the score, the less likely the sentence.

Problem 2.1. This problem asks you to consider some simple properties of this score. How are the likelihoods of sentences distributed? To examine this, use the Bikel parser and sentences following a pattern like this, where we have used braces and a | following a regular expression notation:

The {witch | broker} is dead .

The wicked {witch | broker } is dead .

The wicked wicked {witch | broker } is dead .

The wicked wicked wicked {witch | broker} is dead .

The wicked wicked wicked wicked {witch | broker} is dead .

Using the Bikel parser, analyze these sentences and plot the sentence length in number of tokens (don't forget the period) vs. the log probability score that is returned. What do you observe? What does this tell you about the relationship between sentence length and sentence likelihood, all other things being equal? Is there any difference between the scores when you substitute *broker* for *witch*? Why or why not? Try substituting a few other (possibly common) nouns and see what their effect is on the log probability score. What is your conclusion?

Problem 2.2. Some researchers have suggested using the log probability score as a substitute for that awkward notion of “grammaticality” that linguists like to use. The idea is that less grammatical sentences should have a lower likelihood. (Hmm... something like in your Reading and Response 1.) Now we have a parser to test this idea. Keeping in mind your answer from the previous problem, and the “all other things being equal” caveat, below are a set of sentences. Those marked with asterisks are generally considered as “ungrammatical,” and therefore highly unlikely, to varying degrees. (Let's accept this as a given.) Your job is to run these sentences through the parser, and examine the log probability scores to see how well the notion of “grammaticality” aligns with that of “low likelihood,” at least for these examples. (You might also have to examine the parse trees to make sure the parser is not going wildly astray.) Note that you also may have to construct new, similar examples to adjust for what you discovered in the previous question. Do you think the alignment between grammaticality and likelihood is accurate? Please justify your conclusion with the results of your experiments.

- (1) a. *John slept .*
- b. * *John slept Bill .*
- c. *Bill was arrested .*
- d. * *It was arrested Bill .*
- e. *I am proud of John .*
- f. * *I am proud John .*
- g. * *Who does Mary like John .*
- h. *I tried to leave .*
- i. * *I tried John to leave .*
- j. *I promised John to leave .*

2.2 How well do lexicalized parsers work?

If you haven't already done so, please read the JM chapter 14 posted pdf mentioned at the start of the lab, and then come back here when you are done. OK, if you've read that chapter, you'll recall that a key advance claimed for these state-of-the-art statistical parsers is that they use information about the *head* of a phrase, encoded as part of the nonterminal information in context-free rules. This allows a parser to take into account distinctions between different words and condition the parse on them, what are called *lexical dependencies*. For example, the *head* of a verb phrase (VP) is its verb, the head of a noun phrase (NP) is² its noun, and the head of a prepositional phrase (PP) is its preposition. What the rest of the phrase looks like often greatly depends on which head this is. For example, if it is the verb *saw*, then we would expect that a Noun Phrase (NP) follows (as in *saw the guy*, but if the verb is *sleep*, then the parser should not expect an NP to follow. (We can think of the verb as a function taking some number of arguments.) In fact, since the NP that follows will itself have its own "head", e.g., *guy*, the parser can take that information into account as well. This kind of information is often helpful in disambiguating examples such as the following (from the JM book, chapter 14, figures 14.5 and 14.6):

Workers dumped sacks into a bin

Here, the Prepositional Phrase, *into a bin*, with a head *into* might modify either the VP with the head verb *dumped*, specifying where the sacks were dumped; or an NP with a head noun *sacks*, specifying the type of sacks that are being dumped. The first interpretation, revealing a closer "affinity" between *dump* and *into* rather than *sacks* and *into*. In short, we must condition the parse on the precise words that occur – in particular, the heads of the phrases involved. We do this by adding the head information to the individual rules expanding phrases (details given in Chapter 14 of JM), e.g., instead of 'bare' phrase structure rules with probabilities:

```
VP -> VBD NP      0.8
VP -> VBD NP PP 0.2
NP  -> NNS         0.8
NP  -> NNS PP     0.2
```

We have instead the rules annotated with their corresponding heads, indicating the affinity between particular verb, noun, and prepositional heads directly (here we have made up the probabilities to show that we have 'boosted' the probability of having the PP with *into* tied more closely to *dumped* than with *sacks*):

```
VP-dumped -> VBD-dumped NP-sacks          0.4
VP-dumped -> VBD-dumped NP-sacks PP-into 0.6
NP-sacks  -> NNS-sacks PP-into 0.4
NP-sacks  -> NNS 0.6
```

The JM book uses this notation:

```
VP(dumped) -> VBD(dumped) NP(sacks) PP(into)
```

Of course, we now have a great many more rules, as noted in lecture: assuming binary branching rules, if there are N nonterminals, and we have an alphabet Σ , then since we can have any one of 3 possible nonterminal choices in a rule, and two alphabet (word) symbols, one for the head, and one for the non-head, then there are $|N|^3 \times |\Sigma|^2$ possible rules, a huge number. Since the CKY algorithm runs in time at worst proportional to the cube of the length of the sentence, n^3 , times the grammar size, this not such good news! Fortunately, the number of distinct words in a sentence of length n can be at most n , so we can reduce this grammar size factor to $|N|^3 \times n^3$, for a worst case running time of $O(n^5 |N|^3)$. That is still not great.

Further, we have many more parameters to estimate. In practice, the particular "affinity probabilities" are estimated by examining their frequencies of occurrence in an actual corpus — a count of how many times

²On some accounts. There is a good argument that the head of an NP is in fact the Determiner, or Det, but we put this aside here.

we actually see a PP with the head *into* related to a VP with the head *dumped* as opposed to appearing after an NP with the head *sacks*. As you can imagine, since there are many, many different nouns and verbs, this leads to a challenging estimating problem. Even in a corpus of 49,000 sentences, we would rarely expect to see a particular verb appear in the same context with a particular noun or preposition. Thus, as discussed in JM, what such systems must do is (1) break the total conditional probability chain down into simpler Markov-like parts; and (2) employ some method of smoothing or back-off for missing data. For example, in the Bikel-Collins parser, given some particular verb, e.g., *dumped*, if a specific head noun right context is not found, e.g., *dumped-sacks*, then the system will first back-off to use just the tag of the head, NNS, and if there is no estimate in the corpus for that conditioning value, then a vanilla un-lexicalized context-free rule will be used.

Let's take a quick look at how well this works in practice to resolve the Prepositional Phrase attachment ambiguities that make natural language processing so challenging.

Problem 2.3. We will consider just a single kind of sentence, of this general sort:

(2) *John mixed the water with the milk .*

The Bikel parse for this is as follows. Note that the Prepositional Phrase (PP) *with the milk* is attached “high,” that is, directly under the VP, rather than “low,” as part of the Noun Phrase *the water*. This high attachment means that *the milk* is considered to modify the mixing event, rather than being part of the water, and we are mixing both together. This is probably a sensible result. On the other hand, if we had a sentence such as, *John ate the ice-cream with vanilla* then it seems more sensible to think of the vanilla as being part of the ice-cream. In this problem we'll see how subtle this relationship can be, and whether the statistical parsers successfully “learn” about it from their training data.

Let's see what happens when we interchange *water* and *milk*, as well as try other combinations of verbs and nouns. That is, parse the following sentence and see whether the PP *with water* is attached “high” or “low” by examining the resulting parse tree.

(3) *John mixed the milk with the water .*

In fact, we should be a bit more systematic: try the same experiment and see whether attachment is HIGH or LOW by constructing all the variants of this sentence type with the verbs *mixed* and *sold*, and the nouns *milk*, *water*, and *stock*. We've begun the list below. Your job is to fill out a table that describes whether the PP *with the ...* is attached “high” or “low” for each case. We have placed LOW into the table as your first entry, as per the results from our example *mixed the water with the milk*. You are to fill out the rest of the table, with either the entries HIGH OR LOW. first recording the results from your parse of *mixed the milk with the water*, and then the rest of the table entries, which are blank.

Verb	Noun	Noun with milk	milk with Noun
mixed	water	HIGH	???
mixed	stock	???	
sold	water		
sold	stock		

Problem 2.4. What explains the pattern of HIGH and LOW attachments that you got in the previous problem? To begin to investigate this, we should think about why there should be any differences between the verbs and nouns you select and their attachment preferences – that is, how do such systems learn any preferences in the first place? See if you can figure this out by examining the training corpus file `wsj02-22.mrg` (yes, it's a large file, you can emacs, vi, less or `tgrep`³ to search through it): look for sentences that contain *milk* as an object, and see whether they also have PPs that modify *milk* or not, as compared to the other verbs and nouns above. (You are right: it would be good to have a tool like `grep`

³You can read about `tgrep` in part II of this lab.

to do this, but based on tree structure, but for the purposes of this question, even a text editor will suffice. However, people have built tools to do tree searching of this kind.) Please explain in a few sentences what evidence you can find for why the Bikel parsing system follows the pattern of HIGH and LOW attachments that it does for the cases in your table. What does this imply about the nature of the training data and learning from corpora?

This concludes Part I!

Part II

You have seen in lectures that one of the goals of statistical parsing methods has been to resolve ambiguity by using training examples to select which of several alternatives is most likely. To do this, they have incorporated lexical information in the form of, for example, the ‘heads’ of phrases projected up from lexical entries. For instance, in a Verb Phrase (VP) form like *eat ice-cream*, the verb *eat* is in effect copied up to the VP so that it can play a role in conditioning the expansion of the VP, as to whether, e.g., an NP with a particular head can follow or not.

Such a method seems to capture at least the **syntactic** regularities that accompany different verb types, but the question remains as to whether **semantic** regularities are captured this way. By ‘semantic regularities’ we mean regularities that are reflected in possible/impossible ‘alternations’ as discussed in Beth Levin’s *English Verb Classes and Their Alternations* (1973). For example, we have the following differences between pour and fill:

- (4) *John filled the glass with water.*
- (5) * *John poured the glass with water.*
- (6) *John poured water into the glass.*
- (7) * *John filled water into the glass.*

Such examples are called **verb alternations**. Since these differences are reflected in alternative Prepositional Phrases that are possible or impossible with *fill vs. pour*, one might at least hope that such differences might be picked up by training over some set of examples like the PTB, but of course there are issues as to the sparseness of the training data. And even if such data are present, there is a question as to whether such parsers ‘learn’ the preferences that people (i.e., you) seem to have about such examples.

To make this all concrete, we will assign each of you **three** particular verbs from Levin’s examples. Everybody will have verbs of their own. You will also be able to access in pdf the relevant section of Levin’s book that pertains to the alternation analysis of these two verbs, providing you with Levin’s examples and her analysis of what verb subcategorization frames ought to appear with such verbs. Through out the course of this lab you will collect and analyze four aspects of each verb:

1. The frequency with which each verb appears in the PTB, along with its associated syntactic argument frames (e.g., NP, NP PP, etc.);
2. Levin’s analysis of what frames ought to appear with each verb, according to semantic regularities, along with her examples;
3. Your intuitions as to which verb subcategorizations are more likely than others, for each of your verbs;
4. How the Collins parser ranks each possible subcategorization frame (using negative log probability scores), according to the parsing of test sentences that you construct for your verbs.

Using your findings from 1–4, above, in your lab report we want you to address how the syntactic and semantic regularities line up, comparing Levin, your intuitions, the PTB, and the Collins parser.

We will use the verb *abandon* in what follows as a running example so you can see explicitly the workflow of what you must do and the questions you must address.

Required reading, software, and data

As usual, you can run all the required analyses via an ssh connection to our favorite Athena machine, `linux.mit.edu`. We will cover particulars below. It is also possible to download all the required software and data for use on your own machine, or use a combination of both Athena and your own machine.

Reading:

1. The Levin book, *English Verb Classes and Alternations* (EVCA), 1993. This is available in 3 sections, in pdf format, covering the entire book (though you'll only need to look at parts), as follows, where the numbering refers to sections in the Levin book:
<http://www.mit.edu/~6.863/spring2009/readings/levin1-5.pdf>
<http://www.mit.edu/~6.863/spring2009/readings/levin6-29.pdf>
<http://www.mit.edu/~6.863/spring2009/readings/levin30-57.pdf>
2. The index for the Levin book, which cross-references verbs by section numbers,
<http://www.mit.edu/~6.863/spring2009/readings/evca93.index>
3. The Penn TreeBank file in its sentence and 'merged' forms; a compressed, hashed version used for searching; and sections 2-21 of the PTB used for training the Collins parser. These four files are: `wsj.txt`, `wsj2.mrg`, `wsj2.t2c`, and `wsj2-21.mrg`. We have placed these in `lab3.zip` downloadable from <http://web.mit.edu/6.863/spring2009/code/lab3.zip>
4. The PTB search program, a "tree grep" program, `TGrep2`. This is installed on `linux.mit.edu` under `/mit/6.863/tools/TGrep2`. If you wish to work with your own copy, instructions for downloading and installation is found here.
<http://tedlab.mit.edu/~dr/Tgrep2/>
5. A PTB tree viewing program, `TreeBankViewer`. Installed on athena under:
`/mit/6.863/tools/treebankviewer/viewer`
6. The Collins parser. Installed on `linux.mit.edu`: `/mit/6.863/tools/collins-parser` or available for download here:
<ftp://ftp.cis.upenn.edu/pub/mcollins/PARSER.tar.gz>

Now on to your particular tasks. Grab your verbs and let's go...

Task 1. Collecting and analyzing subcategorization frames

First, we want you to compare the occurrences of your verbs in particular syntactic frames in the PTB with the descriptions of the occurrences of these verbs in Levin (1993) (i.e., your copy of the section of her book for your verbs). This task has the following sub-parts:

1. Retrieve all the 'subcategorization frame' occurrences of your verbs in the Penn Tree Bank, by using the `tgrep2` tool to find the sentence # of these occurrences in the PTB.
2. Compare these retrieved contexts to Levin's possibilities for your verbs and note any differences. Are there contexts that don't occur in the PTB but do occur in Levin, and viceversa? Tabulate the # of occurrences of each frame in the PTB, e.g., VP NP vs. VP NP PP, and the type of each PP (locative, temporal, etc.), noting any semantic differences from what Levin says should occur.

Let's now go through this part of the exercise with our concrete example, *abandon*.

Locating all verb occurrences

To find all the occurrences of *abandon* (including basic inflections) as a verb, we can invoke the `tgrep2` program in the following way,⁴

```
~> tgrep2 -i -a -x -c wsj2.t2c 'VP < (__(/abandon[esi]|abandon$)'
```

After the command there are a set of flags, then the hashed file to search, followed by a tree search pattern in single quotes.

- `-i` flag causes `tgrep2` to be case insensitive. Without this flag, `tgrep2` would miss a target verb beginning with a capital letter.
- `-a` flag enables `tgrep2` to match more than one occurrence of a target pattern in a single tree. Some sentences may have multiple instances of *abandon*
- `-x` flag tells `tgrep2` to print the sentence number of a tree that contains a match and the position of the match within the tree, in the form `s:m`, where *s* represents the sentence number (starting with 1), and *n* is the node number according to a depth-first search order.
- `-c` flag sets the corpus file used by `tgrep2`.

The regular expression that follows searches for a string that is dominated by anything, signified by the wild card symbol `_` and then dominated (`<`) by a VP. In turn, this wild card symbol matches any label that is immediately dominated by a VP node, hence the `<` after the VP node.

Instead of the wild card symbol, the VB symbol could also be used. Each verb has a category label beginning with VB, following the Penn Treebank tags format.⁵

However, use of the wild card symbol is advantageous in that it can match incorrectly labeled verbs. For example, `tgrep2` matched an instance of *awe* that was marked with the label JJ to indicate an adjective. Yet this label was directly dominated by a VP node. For example, inspection of the sentence, “Ms. Johnson, who works out of Aetna’s office in Walnut Creek, an East Bay suburb, is awed by the earthquake’s destructive force” reveals that despite the labeling as an adjective, *awe* is used as a verb in this construction. If `tgrep2` searched for a VB labeled target item, it would not match *awe* in this construction.

In order to match a particular target verb, we use a regular expression. For *abandon*, this is the regular expression:

```
/^abandon[esi]|abandon$/
```

The `abandon[esi]` pattern will match any inflected forms of the verb *abandon*, such as *abandoned*, *abandons*, or *abandoning*. To match the uninflected form of *abandon* we use the pattern `^abandon$`. For additional information on `tgrep2`, you can refer to the pdf manual here:

<http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf>

Executing `tgrep2` using the command args above we get a list of 55 matching PTB sentence numbers with their associated depth levels.

```
1979:54
2370:86
...
48528:90
49163:79
```

⁴We assuming that both the `tgrep2` binary and the PTB hash file `wsj2.t2c` are in the same directory. If you are on an Athena linux machine, you should `cd` to `/mit/6.863/tools/TGrep2`.

⁵For reference on the POS labels, please see the Treebank page: <http://web.mit.edu/6.863/www/PennTreebankTags.html>

Collecting subcategorization frames

Given the sentence numbers you can view the matching parse trees by running the treeviewer. To run the treeviewer program, cd into `/mit/6.863/tools/treebankviewer/` and run the program `./viewer`.⁶ Once you have the treebankviewer TK window up, you must load in the txt file `wsj.txt` and the prolog file `wsj.pl` (located in the same directory). Then enter the sentence number to view the tree.

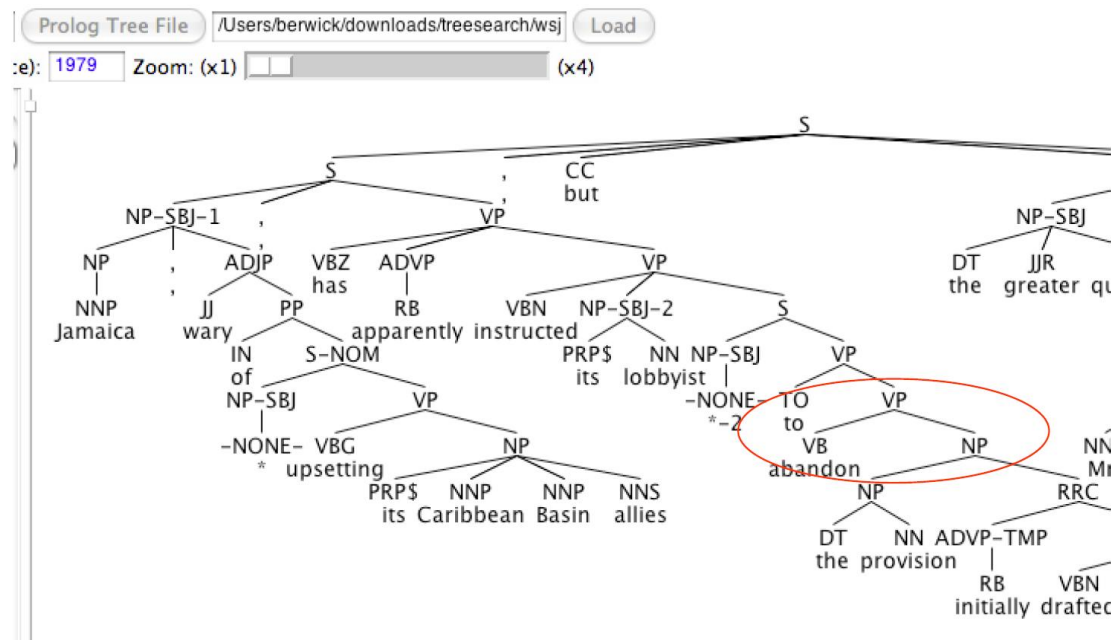


Figure 1: Screen capture of the treebank viewer showing sentence 1979. Here `abandon` is followed by an NP (circled in red)

Certainly you are more than welcome to extract the subcategorization frames manually. But this is after all, computational linguistics, so we will extract the frames programmatically. Running `tgrep2` without the `-x` switch will return all subtrees that matched the search regular expression in the form of s-expressions, such as:

```
(VP (VB abandon) (NP (NP (DT the) (NN provision))) (RRC (ADVP-TMP (RB initially)))
(VP (VBN drafted) (NP (-NONE- *)) (PP (IN by) (NP-LGS (NNP Mr.) (NNP Gray))))))
```

The format of a subcategorization frame is typically: `[XP X args]`. So for the first match above, the extraction program should output: `[VP (VB abandon) NP]`. You may use the provided script: `frame_extractor.py` to help you extract the frames, or you can roll your own. You may also want to print out the verb arguments as a string. You will need to analyze the contents of verb arguments to check whether they agree with Levin. (For example, to determine whether the argument is a location or an event.)

You should include in your report a tabulation of subcategorization frame counts like in table 1. (You should ignore case when counting).

Now, compare the tabulated results to what Levin says. According to Levin, `abandon` is a verb that patterns with `leave` and occurs in the frame `[VP NP]` where the direct object is, the location that has been left.

⁶Remember to login in with XSession forwarding by specifying `-X` or `-Y` flag in your ssh session. `ssh -Y yourlogin@linux.mit.edu`.

	subcategory	count
a)	[VP (VB abandon) NP]	15
b)	[VP (VBD abandoned) NP]	12
c)	[VP (VBN abandoned) NP]	9
d)	[VP (VBG abandoning) NP]	7
e)	[VP (VBN abandoned)]	4
f)	[VP (VBZ abandons) NP]	2
g)	[VP (VBD abandoned)]	1
h)	[VP (VBN abandoned) PP-CLR]	1
i)	[VP (VBN abandoned) PP]	1
j)	[VP (VBP abandon) PP]	1
k)	[VP (VB abandon) NP PP-CLR]	1
l)	[VP (VB abandon) NP S-CLR]	1

Table 1: Table of Subcategorization Frames for abandon

[V NP] as in, *We abandoned the area* (Levin, p. 264)

In 45 out of the 55 matches, examples (a,b,c,d,f), some form of the verb abandon occurs in a verb frame with a direct object NP alone. A look at these sentences also shows that the direct object of *abandon* in the Penn Treebank constructions is **not** a location. For example, in sentence 1979, the direct object of *abandon* is *the provision* and in sentence 3191 the direct object is *the president's proposal for a cap on utilities' sulfur-dioxide emissions*. The other verb frames, (e,g,h,i,j,k,l), that appear in the PTB with abandon in are not discussed in Levin's work.

Now you are to do the same analysis for your assigned verbs. Please write up your findings and along with relevant supporting data. Pay special attention to any semantic 'divergences' as with the example of *abandon* – Levin says that this will be a location, but this is not usually true in the PTB.

Task 2. Comparison of the Collins parser to the PTB results and your intuitions

Now that we have a good grasp of what the PTB says about your verbs, we'll run example sentences based on them through the statistical Collins parser, and see how the log probabilities that the Collins parser for such examples compares to your intuitions and to Levin's analysis.

What might you expect the Collins parser to say about verb alternations? Since it is trained on the PTB, we might expect its logprob rankings to reflect frequencies of the various VP frames in the PTB.

Your key task in this section is to find out whether this is true or not for your verbs. If not, why not? Do the log probability scores reflect your personal intuitions or your expectations about the sentences? Further, you should take into account the following:

1. Are the differences between the log prob values noise or significant? E.g., if we are comparing on a log scale, -99.2 is not very different from -99.7, but -99.2 is very different from -150. This turns out to be a challenging question to answer even for the statistical professionals, so for our purposes we will say that a difference of 1 or less does not matter (it is noise), while a difference of 5 or more does matter.
2. Are 'ungrammatical' sentences (according to you or Levin) assigned radically different log probabilities from grammatical sentences, or do the rankings make sense?
3. Can the Collins parser distinguish valid and invalid alternations for a given verb?

In short, we want you to focus on discrepancies: Collins vs. PTB; Collins vs. linguistic intuition; and PTB vs. linguistic intuition. Are the discrepancies regular or not?

To carry out this task you will need to do the following:

1. Look at Levin's analysis for your verbs and select some sample grammatical and 'ungrammatical' sentences (according to her analysis)
2. Run these examples through the Collins parser and look at the log probability outputs from the parse. You will have to take some care in constructing the sentences to make sure the frequency count for the words you use is > 5 in the PTB, otherwise the resulting word's part of speech will be classified by the parser as 'unknown,' which will spoil the results. You will also have to tag the words you do use, using the PTB reference. You can do that in one of two ways. The quick and dirty way is just to look at how the word is tagged in the PTB and make a sensible choice. Your sentences will be short, so this shouldn't be difficult. The other way is to run it through any of the POS taggers you have used before. If you feel an urgent need to use a more sophisticated tagger, email the TAs and me (6.863-graders@mit.edu), but this shouldn't be necessary.
3. Now use your results to carry out the analysis and answer the questions sketched above. Put the results into your Lab report.

Again, let's run through this exercise using our example *abandon*.

Levin's analysis

Repeating what we noted earlier, according to Levin (1993:264):

abandon is a verb that patterns with *leave* and occurs in the frame [VP NP] where the direct object is "... the location that has been left."

An example is shown in (a). Compare this with the ill-formed example in (b) which shows that the verb frame [V PP] is not allowed.

(a) [V NP] as in, *We abandoned the area.* (Levin, p. 264)

(b) *[V PP] as in, *We abandoned from the area.* (Ibid., p. 264)

Constructing sentences from the Levin examples to run through the Collins parser

Taking the sentences from Levin above as our examples, we next construct the sentence format usable by Collin's parser.

First, we need to check that the words used are frequent enough in the PTB. You can check the word frequency by running `egrep` command over the Collins parser training set to obtain approximate word counts:

```
egrep -ic PATTERN wsj-02-21.mrg
```

Here **PATTERN** is a regular expression for matching the desired word. We might want the pattern to account for different endings (recall `abandon[esi]` as in the `tgrep2` example).

We have also included a python script that will run a word count over a text file of words.

```
word_checker.py -c wsj-02-21.mrg -s abandon.sen
```

`abandon.sen` would be a file containing your example sentence. The script outputs:

```

we      793      (NP-SBJ (PRP we) )
abandoned 30      (VP (VBD abandoned)
the     41185    (NP (DT the) (NN Stage) )
area   221      (NP (DT the) (NNP Pacific) (NN area) ))
.      39563    (. .) ))
...

```

Clearly, these words all contain large enough counts to use.

Next, we need to assign words PTB parts of speech. This is easy to do just by looking at the `egrep` or `word_checker.py` output. So for our example sentence, *we*, the tag is PRP since we see (NP-SBJ (PRP we)). The other tags can be determined similarly. So we have: *abandoned* is a VBD (ed ending on a verb); *we* is a PRP (personal pronoun); *the* is a DT; *area* is an NN; and *from* has the tag IN.

Constructing input sentences to parse

Now we're ready to construct the sentence input file for the Collins parser. The format for each input sentence is as follows:

```
n word1 tag1 word2 tag1 . . . wordn tagn . .
```

where *n* begins the line and is the number of **words** in the full sentence, including the final period (not including the tags!); each word is the actual word token then followed by a blank space and then its part of speech tag; and the final two items are the final period, a space, and then a period (which is also the tag for the final period). A set of sentences is just a collection of these sentences, one per line. So, our two sentences look like this:

```

5 we PRP abandoned VBD the DT area NN . .
6 we PRP abandoned VBD from IN the DT area NN . .

```

Save these Collin's format sentences in a text file, e.g., `abandon.txt`, and we are ready to run the parser.

Running the parser

To parse the sentences, you must either use the parser we have on Athena or else unpack and make the parser on a Macintosh or *nix machine (follow the instructions in the README file that you get). We then `cd` to the directory where the parser is located, on Athena: `/mit/6.863/tools/collinsparser/`

Assuming you are now in the parser directory, you can run the parser in the following way. We will run the so-called "Model 1" version of the parser, which includes just head information. Here, `<tagged_file>` is the file of example sentences that you built from the previous step, e.g., `abandon.txt`, while `<output_file>` is where the detailed parse and probability information will get written (so make sure it's to a directory of yours that you can write to). Some quick things to note: the parser is not very forgiving and if you have miscounted the number of words in a sentence or forgotten a tag, it will just fail. You should get something like the output we show below – it should tell you immediately how many sentences it will parse, and if that doesn't look right, something is probably wrong with your input file.

```

gunzip -c models/model1/events.gz | ./code/parser <tagged_file> \
models/model1/grammar 10000 1 1 1 1 > <another_output_file>

```

Here is the terminal output we get when we run `abandon.txt` through the parser.

```

gunzip -c models/model1/events.gz | ./code/parser abandon.txt \
      models/model1/grammar 10000 1 1 1 1 > abandon-out1.txt
Initialised lexicons
Initialised grammar
Loaded non-terminals
Loaded lexicon
Loaded grammar
NUMSENTENCES 2
Hash table: 100000 lines read
. . .
Hash table 3600000 lines read (Now the parse will actually start - it will be quite fast.)

```

The output file `abandon-out1.txt` looks like this:

```

PROB 298 -25.2695 0
TOP -25.2695 S -16.3069 NP -0.0142832 NPB -1.24178e-05 PRP 0 we
  VP -10.351 VBD 0 abandoned
    NP -2.10062 NPB -1.80633 DT 0 the
      NN 0 area
(TOP~abandoned~1~1 (S~abandoned~2~2 (NPB~we~1~1 we/PRP )
(VP~abandoned~2~1 abandoned/VBD (NPB~area~2~2 the/DT area/NN ./PUNC. ) )
) )
TIME 0
PROB 467 -30.8477 0
TOP -30.8477 S -21.8852 NP -0.0142832 NPB -1.24178e-05 PRP 0 we
  VP -15.9292 VBD 0 abandoned
    PP -8.93852 IN 0 from
      NP -2.10062 NPB -1.80633 DT 0 the
        NN 0 area
(TOP~abandoned~1~1 (S~abandoned~2~2 (NPB~we~1~1 we/PRP )
(VP~abandoned~2~1 abandoned/VBD (PP~from~2~1 from/IN (NPB~area~2~2 the/DT area/NN ./PUNC. ))))
) )
TIME 0

```

Let's look at this output in more detail so you can see how to pull the log probabilities out of it. The first line before each sentence parse gives a number you can ignore, and then the total log probability for the sentence (in negative logarithms, so closer to 0 is better). That is followed by a detailed breakdown of the log probability for each phrase in the sentence. For our first sentence then, we abandoned the area, the total log probability is **-25.2695**. The next lines break down this probability by phrases. The VP log probability for *abandon the area* is:

```

VP -10.351 VBD 0 abandoned
  NP -2.10062 NPB -1.80633 DT 0 the
    NN 0 area

```

In other words, the negative log probability for the whole VP is -10.351 while that for a VBD label for abandon is 0 (i.e., probability e^0 or 1); the logprob score for the NP *the area* is **-2.10062** (The next line, headed by TOP `abandoned`, shows the structure of the parse, with the heads included, and the last line, **TIME**, gives the total time to find this parse.)

Turning to the second sentence, we see that it has a higher negative score of **-30.8477**, hence a lower probability. This lower probability reflects in turn the lower probability of the VP -15.9292, which in turn results from the lower probability of the PP, **-8.93852**, in this particular verb context. This difference of -5 compared to to the first sentence is quite **large**.

Analysis of results

These results show that the probability of the [V NP] frame for *abandon*, as in sentence (a) above, is significantly greater than the probability of [V PP] as in (b). This holds for the probabilities in each sentence and each VP. These results certainly are in line with intuitions and Levin: Example (a) should be better than (b), as (b) is ungrammatical according to Levin (1993), and according to my intuitions. Also, in Task 1 in the PTB, we found that 45 (out of a total of 55) instances of *abandon* occurring in a [V NP] verb frame. Further, even though the (b) sentence is longer, which could add to its negative probability, we note that the PP probability as a whole is lower, hence we may tentatively say that there is not really a length effect. However, to be complete, one should really try to assess this effect by constructing (a) type sentences of similar length to (b) type sentences. That said, overall, we conclude that the parser does a reasonable job of reflecting the alternation contrasts for this verb.

Your task

Now go and do likewise for the two verbs that you have been assigned. Be sure to include an analysis of the outcome in your Lab report. Pay special attention to the questions posed at the beginning of this section, summing up with your assessment of whether the Collins parser has captured the lexical alternation effects for your verbs. Be sure to do a better analysis of the possible effect of sentence length than we have done above.

This concludes Laboratory 3!