CS 224N: Natural Language Processing
Prof. Christopher D. Manning
Spring 2007-2008

Valentin I. Spitkovsky
Final Programming Project
2008-06-05

An *Attempt* at Unsupervised Learning of Hierarchical Dependency Parsing

via the

Dependency Model with Valence (DMV)

(code submitted before 10AM on 2008-06-05: 0 days late)
(text submitted before 10AM on 2008-06-06: 1 days late)

1 Summary

In this final assignment, we attempted to implement an algorithm for unsupervised learning of hierarchical dependency parsing, based on the Klein & Manning's Dependency Model with Valence (DMV) of 2004. The nominal goal, which sadly did not meet with success, was to match their numbers on *The Wall Street Journal* section of the Penn Treebank Project. However, we are more satisfied with the real educational goals of learning the Inside-Outside algorithm (building on what we learned about the Expectation Maximization in the second programming assignment), re-deriving Klein's model (some formulae were omitted from the appendix of his thesis, while others appeared to be incorrect), implementing a probabilistic dependency parser for the DMV (building on what we learned about [P]CFGs in the third programming assignment), solidifying our understanding of hierarchical dependencies (from lectures) as well as their pseudo-isomorphism with constituency grammars (by means of head percolation rules, from readings). Given a bit of extra time, we intend to debug our implementation and go on to learn dependency parsing on significantly larger quantities of data. Until then, we short-circuited the problem by computing the surface level statistics for the probabilities tracked by the DMV, evaluating the faked model's performance, and considering its linguistic short-comings and potential improvements.

2 Motivation

Klein points out several reasons to focus on dependency parsing. One is that "A central motivation for using tree structures in computational linguistics is to enable the extraction of dependencies — function-argument and modification structures — and it might be more advantageous to induce such structures directly." Another is that "For languages such as Chinese, which have few function words, and for which the definition of lexical categories is much less clear, dependency structures may be easier to detect."

For us, our connection to Google dictates at least two obvious applications (most likely, there will be many others): 1) search query refinement may improve if certain dependent words or clauses were dropped in a clever way; and 2) statistical machine translation could perhaps benefit from having a dependency structure imposed on a sentence, allowing for somewhat more modular translation of clauses. Google's raw processing power and sheer volumes of unannotated data would eagerly make heavy use of a good unsupervised learning technique.

Contents

3 History

3.1 Original Project Proposal

```
CS 224N: Natural Language Processing
Prof. Christopher D. Manning
Spring 2007-2008


Valentin I. Spitkovsky (vals@stanford.edu)
Final Project Abstract
2008-05-07


Problem Being Investigated
======= ===== ============

Unsupervised learning of hierarchical dependency parsing.

This is a joint project that I am undertaking with Prof. Daniel
Jurafsky at Stanford and Dr. Hiyan Alshawi at Google.  Presently,
my understanding is that most successful parsers are supervised.
But if we could devise some useful novel unsupervised techniques,
then we could unleash Google's supply of data and raw power at
the problem, possibly allowing us to climb to greater heights...

We see at least two obvious applications (most likely, there
will be many others): 1) search query refinement may improve if
certain dependent words or clauses were dropped in a clever way;
2) statistical machine translation could perhaps benefit from
having a dependency structure imposed on a sentence,
allowing for somewhat more modular translation of clauses...


Approach / Plan
======== = ====

I am still very vague on the potential implementation details,
but I see at least two clear options: 1) learn about the state
of the art unsupervised methods for parsing, implement them,
and test how they scale with more data, using Google's machines;
2) somehow make use of parallel data -- assuming that this isn't
something that's already being done by state-of-the art parsers:
we could use the bi-texts available at Stanford, or Google's
stores of parallel data, or even the sentence-aligned
intermediates from Google's MT systems (before
they are fed into word-alignment training).

In terms of testing, I suppose we could simply reuse some
data from the TreeBank bake-offs (I've been told that
Ryan McDonald, who was closely involved with these,
now works for Google New York).


Achievements So Far / Relevant References Found
============ == === = ======== ========== =====

Not much to report in terms of achievements thus far: the term
has really kept me busy as it was...  Also, we just started going
over parsing in general this week, so I am very much a newbie,
but hope to know much more by the time I submit the third
programming assignment.  In any case, I was very happy
to see dependency parsing in Monday's lecture.  :)
```

```
I've been pointed to several references so far:

A Tutorial on Dependency Parsing:
  - http://dp.esslli07.googlepages.com/

A Paper on Isomorphism:
  - Marie-Catherine de Marneffe, Bill MacCartney and Christopher D.
    Manning. 2006.  Generating Typed Dependency Parses from Phrase
    Structure Parses. In LREC 2006.

A Nivre Paper:
  - http://www.msi.vxu.se/users/nivre/papers/acl05.pdf

Dan Klein's Work on Unsupervised Learning:
  - Corpus-Based Induction of Syntactic Structure: Models of
    Dependency and Constituency, Dan Klein and Chris Manning,
    In Proceedings of (ACL) 2004.
  - The Unsupervised Learning of Natural Language Structure,
    Dan Klein, Ph.D. Thesis, Stanford University 2005.

Other Recent Theses:
  - Head-Driven Statistical Models for Natural Language Parsing,
     Michael Collins, Ph.D. Thesis, MIT 1999.
  - Discovery of Linguistic Relations Using Lexical Attraction,
    Deniz Yuret, Ph.D. Thesis, MIT 1998.

... this will probably be too much to absorb during the term
(and still have time to build a system), but I hope that we
could narrow it down to a small set of key references once
we prune down the scope of the project and have have a
face-to-face conversation or two.


Plan for the Remainder of the Quarter
==== === === ========= == === =======

Focus on lectures and the remaining programming assignment to
learn about parsing in general, then get acquainted with the
literature and have some in-person conversations with the
staff, then get to work -- the term is short!

The longer-term plan is to make use of my time in the NLP
class to get an introduction to the field, have the final
project turn into a base-line for judging the full time
summer project, then before next year decide whether or not
this sort of collaboration would make sense going forward
(if not, I would leave Google and find another project at
Stanford; if yes, I would remain at Google on a one-day-a-week
basis, which would still allow me access to data and machines).

One very important question: if I were to do a lot of
the computation at Google, I probably couldn't share the code
in its entirety, since it is bound to touch some proprietary
stuff...  how big of a problem would this be?  Would it be
okay to isolate the relevant classes and submit those,
even if they wouldn't run stand-alone for the course staff?


P.S.  In addition to the above, I will also be doing a final
project in a class on nonparametric statistics.  I am toying
with the idea of perhaps coaxing some of the methods we learned
there into an algorithm for judging translation similarity, as
a potential competitor to BLEU.  If that ends up working out,
I'd be happy to report on those results as well.  :)
```

3.2 Instructor's Feedback / Clarifications

CC: cs224n-spr0708-staff@mailman.stanford.edu, vspitkovsky@yahoo.com
From: "Christopher Manning" <manning@stanford.edu>
To: "Valentin I. Spitkovsky" <valentin@google.com>
Subject: Re: [cs224n-spr0708-staff] Final Project Abstract
Date: Tue, 27 May 2008 08:32:17 -0700

Hi, here's some (very slow - sorry!) feedback on your cs224n project proposal:

SCOPE It's obviously something you can keep working on, but for cs224n, you should be careful that the scope doesn't become too large. Getting an existing unsupervised dependency parser reimplemented, working and evaluated on some data is already I think a large project for a single person. Doable. For instance, someone (a current PhD student, Chuong Do) did reimplement Klein's CCM model for a cs224n final project, and got it working. But there probably isn't also a ton of time to then investigating scaling. In particular, Klein never ran his stuff on anything bigger than the Penn treebank. To run it on 10 million words of data would already be an order of magnitude more data. You don't immediately need terabytes. (I do believe that dependency parsing accuracy should be able to improve with a lot of data, whereas it's not so clear that is true for the CCM model.) So, doing less than you suggest for 1) would be quite sufficient. It seems to me a little harder to see how to keep (2) to an appropriate scale.

LITERATURE The tutorial you cite is probably the best way to get up to speed on the active recent thread of work on supervised dependency parsing. I don't think it covers unsupervised work, though. I may be biased, but I really think there is still no better place to start than the Klein and Manning 2004 DMV model.

There has been some work on dependency-based models in MT, most noticeably a whole bunch of papers at Microsoft Research, and also some at BBN. But I see this as beyond the scope of the cs224n project.

Here's one paper that uses bilingual data for unsupervised (PCFG) syntax induction that looked quite interesting:

http://acl.ldc.upenn.edu/P/P04/P04-1060.pdf

EXCITING PROBLEM Yes.

EVALUATION Straightforward, as you say, within the limits of unsupervised systems in principle being hard to evaluate.

CLEAR PLAN I think there now is.

OVERALL Fine.

> One very important question: if I were to do a lot of
> the computation at Google, I probably couldn't share the code
> in its entirety, since it is bound to touch some proprietary
> stuff... how big of a problem would this be? Would it be
> okay to isolate the relevant classes and submit those,
> even if they wouldn't run stand-alone for the course staff?

It would be. The best case for us is if we can just run things, but we've dealt with situations before where we've done similar compromises. But I rather think that in this situation, you'd be genuinely better off writing something standalone and using it as the cs224n project, and leaving an integrated model until later. That is, if at some point you wanted to run something large scale and using parallel texts then you may well want/need to have a distributed MapReduce implementation and interface to other google goo. But I

don't see how this would do anything other than distract you and slow
you down for the purposes of writing a cs224n project.

> P.S. In addition to the above, I will also be doing a final
> project in a class on nonparametric statistics. I am toying
> with the idea of perhaps coaxing some of the methods we learned
> there into an algorithm for judging translation similarity, as
> a potential competitor to BLEU. If that ends up working out,
> I'd be happy to report on those results as well. :)

DanJ and me would be interested to hear what you do. We've actually
been given money to come up with more sophisticated approaches to MT
evaluation ... though the direction of that work is more in the line
of exploiting parsing, and semantic equivalence (RTE-style
equivalanece/semantic similarity).

Chris.

On May 7, 2008, at 10:49 PM, Valentin I. Spitkovsky wrote:

> For some reason Yahoo! Mail is giving me trouble, so
> I am sending out this proposal from my work address,
> just to make sure that it will be time-stamped on time.
>
> Best,
>
> Val.
>
> P.S. Page two of the hand-out still lists a pervious
> year's mailing list, so you may want to be on a
> look-out for submissions to cs224n-spr0607-staff...
> <abstract.txt>____
> cs224n-spr0708-staff mailing list
> cs224n-spr0708-staff@lists.stanford.edu
> https://mailman.stanford.edu/mailman/listinfo/cs224n-spr0708-staff

4 Investigation / Research Question

Although we outlined a number of potential applications when motivating the subject, for the purposes of this project, we limited our scope to simply replicating the work of Klein and Manning. The clear question we set out to answer was whether or not we could reproduce their results.

We feel that in the short amount of time given to work on this project we have arrived at a fairly good understanding of their methods. It would have been nicer if our implementation of the algorithm had also worked as advertised, but thankfully the project guidelines state that "You will not be penalized if your system performs poorly, providing your initial design decisions weren't obviously unjustifiable, and you have made reasonable attempts to analyze why it failed, and to examine how the system might be improved." And while we *will* make every effort to debug our system after the quarter is over, given that there is a 10% daily penalty for late submissions, it made little sense to delay writing up this report, since we failed to secure an extension...

Why our implementation fails to exhibit the expected behavior is still an open question. It's possible, though we imagine it to be extremely unlikely, that the model simply doesn't work. A more plausible explanation is that its performance is highly sensitive to tiny little (e.g. off-by-one) bugs, which are very likely to creep into any algorithm that's complicated enough to drum up sufficiently many inner loops to score an $O(n^5)$ running time for itself... It could also be that we either misunderstood the model or made errors in our derivations (we are fairly certain that some of the few formulae presented by Klein himself are incorrect). And while we made every effort to `assert` all of our invariants (e.g. multiple ways to compute the probability of a sentence, given the inside and outside charts), there are no guarantees from silly (or not so silly) errors...

5 Algorithms / Methods Used

Klein mentions several existing generative dependency models intended for unsupervised learning. However, these models — even when trained on large amounts of data, "using EM, in an entirely unsupervised fashion, and at great computational cost" — "... resulted in parsers which predicted dependencies at below chance level (measured by choosing a random dependency structure)." Klein partly attributes this below-random performance to the models' linking word pairs which have high mutual information (e.g. occurrences of *Congress* and *bill*), regardless of whether their syntactical relation is plausible. Klein also criticizes these models for lacking a means to encode valence, a term he uses to broadly refer to the regularities in number and type of arguments a word or word class takes (e.g. not all occurrences of *New* and *York* should be attached, especially those which are not adjacent).

5.1 Dependency Model with Valence

Assume a reserved non-terminal `ROOT`, whose sole dependent is the head of the sentence (this simplifies the notation, math and the evaluation metric). Consider a simple head-outward dependency model over word classes, now including a model of valence. It begins at the `ROOT`. Each head generates a series of non-`STOP` arguments to one side, then a `STOP` argument to that side, then non-`STOP` arguments to the other side, then a second `STOP`. The deciding left-first versus right-first head choice incurs a cost according to $\mathrm{P}_{\texttt{ORDER}}(w)$, the probability that in this derivation the head will attach its left arguments before its right arguments.

In this process, there are two kinds of derivation events, whose local probability factors constitute the model's parameters. First, there is the binary decision whether or not to stop: $\mathrm{P}_{\texttt{STOP}}(h, dir, adj)$, conditioned on the head h (the word class), the direction dir (generating to the left or right of the head), and the adjacency *adj* (whether or not an argument has already been generated in the current direction). This stopping decision is estimated directly, without smoothing.

If a stop is generated, then no more arguments will be produced for the current head to the current side. Otherwise, another argument is chosen, according to probability $\mathrm{P}_{\texttt{ATTACH}}(h, dir, arg)$, where the argument *arg* is picked conditionally on the identity of the head h (again, a word class) and the direction *dir*. This term too is not smoothed in any way, only now adjacency has no effect on the identity of the argument (it only affects the likelihood of termination). Every new argument's sub-tree in the dependency structure is generated recursively.

5.1.1 A Mathematical Formalism

For a dependency structure S, with each word h having left dependents $D_S(h, \texttt{L})$ and right dependents $D_S(h, \texttt{R})$, the probability of the fragment $S(h)$ of the dependency tree rooted at h is defined as follows:

$$\mathrm{P}(S(h)) = \prod_{dir \in \{\texttt{L},\texttt{R}\}} \mathrm{P}_{\texttt{STOP}}(h, dir, adj) \prod_{arg \in D_S(h,dir)} \mathrm{P}(S(arg))\mathrm{P}_{\texttt{ATTACH}}(h, dir, arg)(1 - \mathrm{P}_{\texttt{STOP}}(h, dir, adj)).$$

5.1.2 A Context-Free Grammar

The structure generated by this derivational process can be viewed as a lexicalized tree composed of the following local binary and unary context-free configurations:

$$\begin{array}{rrcccc}
\text{Right-first} & \text{Head Choice} & \vec{w} & \rightarrow & & w \\
\text{Left-first} & \text{Head Choice} & \overset{\leftarrow}{w} & \rightarrow & & w \\
\\
\text{Right-first} & \text{Right Attachment} & \vec{h} & \rightarrow & & \vec{h}\ \ \overline{a} \\
\text{Right-first} & \text{Left Attachment} & \overset{\leftrightarrows}{h} & \rightarrow & \overline{a} & \overset{\leftrightarrows}{h} \\
\\
\text{Left-first} & \text{Left Attachment} & \overset{\leftarrow}{h} & \rightarrow & \overline{a} & \overset{\leftarrow}{h} \\
\text{Left-first} & \text{Right Attachment} & \overset{\rightleftarrows}{h} & \rightarrow & & \overset{\rightleftarrows}{h}\ \ \overline{a} \\
\\
\text{Right-first} & \text{Right Stop} & \overset{\leftrightarrows}{h} & \rightarrow & & \vec{h} \\
\text{Left-first} & \text{Left Stop} & \overset{\rightleftarrows}{h} & \rightarrow & & \overset{\leftarrow}{h} \\
\\
\text{Right-first} & \text{Seal} & \overline{h} & \rightarrow & & \overset{\leftrightarrows}{h} \\
\text{Left-first} & \text{Seal} & \overline{h} & \rightarrow & & \overset{\rightleftarrows}{h}
\end{array}$$

Each of the four configurations equivalently represents either a head-outward derivation step or a context-free rewrite rule. The terminal symbols are $w \in W$ (here, the terminal vocabulary W is the set of word classes). The non-terminal symbols are $\vec{w}$, $\overset{\leftarrow}{w}$, $\overset{\leftrightarrows}{w}$, $\overset{\rightleftarrows}{w}$ and $\overline{w}$, for $w \in W \cup \{\diamondsuit\}$.

Sentences are imagined to end with the `ROOT` symbol $\diamondsuit$, with $\overline{\diamondsuit}$ always serving as the start symbol. Trees rooted at $\overline{h}$ are called *sealed*, since their head h cannot take any further arguments to either side, in this grammar's topology. Trees with roots $\overset{\leftrightarrows}{h}$ and $\overset{\rightleftarrows}{h}$ are called *half-sealed*, since they can only take arguments to the final attachment side (L and R, respectively). Trees rooted at $\vec{h}$ and $\overleftarrow{h}$ are called *unsealed*, since they can take arguments to either side (initially R and L, but eventually L and R, respectively).

5.2 Inside-Outside Algorithm

The model can be re-estimated using the basic inside-outside algorithm, which gives for each sentence $s \in \Omega$ the expected fraction of parses of s with a node labeled x extending from position i to position j:

$$c_s(x : i, j).$$

Klein states that the inside and outside recurrences over items $(x : i, j)$ are defined in the standard way. We expand on his notation, to make things more precise: let s be a sentence $s_0 \ldots s_{l-1}\diamondsuit$ (originally containing length l words $s_i \neq \diamondsuit$) and recall the various derivation styles (or types) $t \in \{\rightarrow, \leftarrow, \leftrightarrows, \rightleftarrows, -\}$. Then the items of interest are

$$(i, \overset{t}{h}, j), \text{ for } i \in [0, l),\ j \in (i, l] \text{ and } h \in [i, j),$$

representing $s_i \ldots s_{j-1}$ derived from $\overset{t}{s}_h$. The standard unknowns of interest are the inside probabilities

$$\mathrm{P}_{\mathcal{I}}(i, \overset{t}{h}, j) \equiv \mathrm{P}(\overset{t}{s}_h \text{ derives } s_i \ldots s_{j-1})$$

and the outside probabilities

$$\mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j) \equiv \mathrm{P}(\overline{\diamondsuit} \text{ derives } s_0 \ldots s_{i-1}\overset{t}{s}_h s_j \ldots s_{l-1}\diamondsuit),$$

whose product yields the desired fractions of parses, from the inherent assumptions of independence:

$$c_s(i, \overset{t}{h}, j) = \mathrm{P}_{\mathcal{I}}(i, \overset{t}{h}, j)\mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j).$$

5.2.1 The Inside Chart (Bottom-Up)

- Base Cases: projections of the terminals, which represent the point where it is decided whether a word w will take its arguments to the left, then right, or to the right, then left; using our notation,

$$\mathrm{P}_{\mathcal{I}}(i, \overleftarrow{i}, i+1) = \mathrm{P}_{\mathtt{ORDER}}(s_i),$$

$$\mathrm{P}_{\mathcal{I}}(i, \vec{i}, i+1) = 1 - \mathrm{P}_{\mathcal{I}}(i, \overleftarrow{i}, i+1),$$

  for $i \in [0, l)$; to simplify what follows, assume also that

$$\mathrm{P}_{\mathcal{I}}(i, \overset{t}{i}, i+1) = 0$$

  for $t \notin \{\rightarrow, \leftarrow\}$, i.e. for $t \in \{\leftrightarrows, \rightleftarrows, -\}$.

- Inductive Updates (Binary Propagation): spans greater than one ($j > i + 1$) arise from summing over various shorter decompositions. Half-sealed scores are expressed in terms of smaller half-sealed scores, and similarly for the unsealed scores:

$$\mathrm{P}_{\mathcal{I}}(i, \overset{t}{h}, j) = \sum_{k=i+1}^{h} \mathrm{P}_{\mathcal{I}}(k, \overset{t}{h}, j)(1 - \mathrm{P}_{\mathtt{STOP}}(s_h, \mathtt{L}, 1_{k=h})) \sum_{a=i}^{k-1} \mathrm{P}_{\mathtt{ATTACH}}(s_h, \mathtt{L}, s_a)\mathrm{P}_{\mathcal{I}}(i, \overline{a}, k), \text{ for } t \in \{\leftarrow, \leftrightarrows\};$$

$$\mathrm{P}_{\mathcal{I}}(i, \overset{t}{h}, j) = \sum_{k=h+1}^{j-1} \mathrm{P}_{\mathcal{I}}(i, \overset{t}{k}, j)(1 - \mathrm{P}_{\mathtt{STOP}}(s_h, \mathtt{R}, 1_{k=h+1})) \sum_{a=k}^{j-1} \mathrm{P}_{\mathtt{ATTACH}}(s_h, \mathtt{R}, s_a)\mathrm{P}_{\mathcal{I}}(k, \overline{a}, j), \text{ for } t \in \{\rightarrow, \rightleftarrows\}.$$

- Inductive Updates (Unary Propagation): all spans, regardless of their length, must be adjusted by summing over further decompositions of equal length. Sealed scores are expressed in terms of half-sealed scores; analogously, half-sealed scores are expressed in terms of smaller half-sealed scores:

$$\mathrm{P}_{\mathcal{I}}(i, \overset{\leftrightarrows}{h}, j) \overset{+}{=} \mathrm{P}_{\mathcal{I}}(i, \vec{h}, j)\mathrm{P}_{\texttt{STOP}}(s_h, \texttt{R}, 1_{j=h+1}),$$

$$\mathrm{P}_{\mathcal{I}}(i, \overset{\rightleftarrows}{h}, j) \overset{+}{=} \mathrm{P}_{\mathcal{I}}(i, \overset{\leftarrow}{h}, j)\mathrm{P}_{\texttt{STOP}}(s_h, \texttt{L}, 1_{i=h});$$

  in turn (order matters here),

$$\mathrm{P}_{\mathcal{I}}(i, \overline{h}, j) \overset{+}{=} \mathrm{P}_{\mathcal{I}}(i, \overset{\rightleftarrows}{h}, j)\mathrm{P}_{\texttt{STOP}}(s_h, \texttt{R}, 1_{j=h+1}) + \mathrm{P}_{\mathcal{I}}(i, \overset{\leftrightarrows}{h}, j)\mathrm{P}_{\texttt{STOP}}(s_h, \texttt{L}, 1_{i=h}).$$

In the end, the probability of this sentence is $\mathrm{P}_s = \mathrm{P}_{\mathcal{I}}(0, \overline{\diamondsuit}, l+1)$.

5.2.2 The Outside Chart (Top-Down)

Klein simply states that the outside recurrences are similar. Nevertheless, we found this exercise for the reader quite tedious, labor-intensive and worse error-prone — we spotted several typos in Klein's inside recurrences and therefore now proceed to rederive the outside recurrences anew as well, decorating all sums with their limits of summation throughout, since this work may be of future use to ourselves or others:

- Base Cases: enforce the structural assumption that the `ROOT` symbol derives all possible parses.

$$\mathrm{P}_{\mathcal{O}}(0, \overline{\diamondsuit}, l+1) = 1;$$

  to simplify what follows, assume also that every other cell of the chart has been zeroed out...

- Inductive Cases: for every symbol's appearance on the right-hand side of a rule, work back to the production which generated it (most aren't too bad, but sealed parses appear in four places):

$$\mathrm{P}_{\mathcal{O}}(i, \overline{h}, j) = \sum_{k=0}^{i-1}\sum_{a=k}^{i-1}(1 - \mathrm{P}_{\texttt{STOP}}(s_a, \texttt{R}, 1_{i=a+1}))\mathrm{P}_{\texttt{ATTACH}}(s_a, \texttt{R}, s_h) \sum_{t\in\{\rightarrow,\rightleftarrows\}} \mathrm{P}_{\mathcal{I}}(k, \overset{t}{a}, i)\mathrm{P}_{\mathcal{O}}(k, \overset{t}{a}, j)$$

$$+ \sum_{k=j+1}^{l}\sum_{a=j}^{k-1}(1 - \mathrm{P}_{\texttt{STOP}}(s_a, \texttt{L}, 1_{j=a}))\mathrm{P}_{\texttt{ATTACH}}(s_a, \texttt{L}, s_h) \sum_{t\in\{\leftarrow,\leftrightarrows\}} \mathrm{P}_{\mathcal{I}}(j, \overset{t}{a}, k)\mathrm{P}_{\mathcal{O}}(i, \overset{t}{a}, k),$$

$$\mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j) = (1-\mathrm{P}_{\texttt{STOP}}(s_h, \texttt{R}, 1_{j=h+1})) \sum_{k=j+1}^{l} \mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, k) \sum_{a=j}^{k-1} \mathrm{P}_{\texttt{ATTACH}}(s_h, \texttt{R}, s_a)\mathrm{P}_{\mathcal{I}}(j, \overline{a}, k), \text{ for } t \in \{\rightarrow, \rightleftarrows\},$$

$$\mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j) = (1 - \mathrm{P}_{\texttt{STOP}}(s_h, \texttt{L}, 1_{i=h})) \sum_{k=0}^{i-1} \mathrm{P}_{\mathcal{O}}(k, \overset{t}{h}, j) \sum_{a=k}^{i-1} \mathrm{P}_{\texttt{ATTACH}}(s_h, \texttt{L}, s_a)\mathrm{P}_{\mathcal{I}}(k, \overline{a}, i), \text{ for } t \in \{\leftarrow, \leftrightarrows\};$$

  lastly (once again, order matters),

$$\mathrm{P}_{\mathcal{O}}(i, \overset{\rightleftarrows}{h}, j) \overset{+}{=} \mathrm{P}_{\texttt{STOP}}(s_h, \texttt{R}, 1_{j=h+1})\mathrm{P}_{\mathcal{O}}(i, \overline{h}, j) \text{ and } \mathrm{P}_{\mathcal{O}}(i, \overset{\leftrightarrows}{h}, j) \overset{+}{=} \mathrm{P}_{\texttt{STOP}}(s_h, \texttt{L}, 1_{i=h})\mathrm{P}_{\mathcal{O}}(i, \overline{h}, j),$$

$$\mathrm{P}_{\mathcal{O}}(i, \vec{h}, j) \overset{+}{=} \mathrm{P}_{\texttt{STOP}}(s_h, \texttt{R}, 1_{j=h+1})\mathrm{P}_{\mathcal{O}}(i, \overset{\leftrightarrows}{h}, j) \text{ and } \mathrm{P}_{\mathcal{O}}(i, \overset{\leftarrow}{h}, j) \overset{+}{=} \mathrm{P}_{\texttt{STOP}}(s_h, \texttt{L}, 1_{i=h})\mathrm{P}_{\mathcal{O}}(i, \overset{\rightleftarrows}{h}, j).$$

Now the probability of this sentence can be re-computed in multiple alternative — yet equivalent! — ways:

$$\mathrm{P}_s = \mathrm{P}_{\mathcal{O}}(i, \overset{\leftarrow}{i}, i+1)\mathrm{P}_{\texttt{ORDER}}(s_i) + \mathrm{P}_{\mathcal{O}}(i, \vec{i}, i+1)(1 - \mathrm{P}_{\texttt{ORDER}}(s_i)), \ \forall i \in [0, l], \text{ with } \mathrm{P}_{\texttt{ORDER}}(\diamondsuit) \equiv 1.$$

5.2.3 Remarks

Both charts lend themselves to straight-forward algorithms with $O(n^5)$ running times, based on the recurrences given above. And while $O(n^3)$ solutions exist, making use of various triangular and trapezoidal sub-structures, we found no urgent need to chase such efficiency prematurely...

However, as we already hinted multiple times, the intricate details of these algorithms leave much room for errors. Thus, we sprinkled into our code numerous `asserts`, which verify structural invariants, including the known probabilities associated with `ROOT` $\diamondsuit$, as well via multiple calculations and comparisons of P_s.

5.2.4 Re-Estimation

Klein claims that given the inside and outside scores, the fraction of trees over a given sentence which contain any of the structural configurations which are necessary to re-estimate the model multi-nomials can be calculated easily. For the most part, we did not find this to be the case...

- $\mathrm{P}_{\mathtt{STOP}}$: as an example, Klein explains that $\mathrm{P}_{\mathtt{STOP}}(s_h, \mathtt{L}, 0)$, according to a current model Θ, could be computed as the ratio of two quantities: 1) the (expected) number of trees headed by $\overset{\leftrightarrows}{s}_h$ whose start position i is strictly left of h; to 2) the (expected) number of trees headed by $\overline{s}_h$ with start position i strictly left of h. Their ratio is the maximum likelihood estimator of the local probability factor:

$$\mathrm{P}_{\mathtt{STOP}}(s_h, \mathtt{L}, 0) = \frac{\sum_{s \in \Omega} \sum_{i=0}^{h-1} \sum_{j=i+1}^{l} c_s(i, \overline{h}, j)}{\sum_{s \in \Omega} \sum_{i=0}^{h-1} \sum_{j=i+1}^{l} c_s(i, \overset{\leftrightarrows}{h}, j).}$$

  Klein points out that this can be intuitively thought of as the relative number of times a tree headed by s_h had already taken at least one argument to the left, had an opportunity to take another, but didn't. Sadly, this formulation tripped some of our assertions. Upon closer examination, we were left confused about two issues: 1) shouldn't only a fraction of the numerator's probability mass associated with $\overline{s}_h$ stem from $\overset{\leftrightarrows}{s}_h$, with the remainder having been derived from $\overset{\rightleftarrows}{s}_h$ (indeed, as is, the formula sometimes yields "probabilities" greater than one)? and 2) doesn't this miss an entire spectrum of stopping observations generated in the other direction, when $\overset{\leftarrow}{s}_h$ morphs into $\overset{\rightleftarrows}{s}_h$?

  We attempted to reason our way towards a more appealing ratio, but we found this business quite tricky, since there is a lot of very subtle aggregation going on. Instead, we implemented a more direct brute force approach, which is more obviously a sound probability distribution. For each position h in every sentence $s \in \Omega$, various sealed events $(i, \overline{h}, j)$ must be disjoint and occurring with likelihood $c_s(i, \overline{h}, j)$. Furthermore, each event corresponds to $\overline{s}_h$ having exactly $h - i$ left-attachments and $j - h - 1$ right-attachments. Focusing on a particular direction $dir \in \{\mathtt{L}, \mathtt{R}\}$, we could compute p_i — the weighted-average fraction of time that position h had i children in this direction. Then estimating the adjacent probability of stopping becomes straight-forward:

$$\mathrm{P}_{\mathtt{STOP}}(s_h, dir, 1) = p_0,$$

  the probability of having no children in direction dir. Now, this contribution must itself be weighted by $\sum c_s(i, \overline{h}, j)$, since there may be other instances of s_h in the sentence, but that's a detail...

  The non-adjacent case is only slightly trickier. If $p_0 = 1$, then there isn't any data to make an informed contribution. Otherwise, let $p'_i = p_i/(1 - p_0)$ for $i > 0$ and note that we are faced with an instance of the Geometric distribution: if the probability of stopping is p, then the probability of having i children is $p(1-p)^{i-1}$, given that the first child was already generated by an independent process.

The expected number of trials up to and including the final stop, for the Geometric distribution, is $1/p$. Thus, we compute the sample mean of the number of attempted stops — letting the first child represent the actual stop symbol, to make accounting simpler — and find the method of moments estimate:

$$\sum_{i=1}^{\infty} ip'_i = \hat{\mu} = \frac{1}{\hat{p}} \;\rightarrow\; \hat{p} = \frac{1}{\sum_{i=1}^{\infty} ip'_i} \text{ and let } \mathrm{P}_{\texttt{STOP}}(s_h, dir, 0) = \hat{p},$$

also to be weighted down, only even further, to exclude the likelihood of adjacent events...

- $\mathrm{P}_{\texttt{ORDER}}$: Klein does not explain how this probability is to be inferred from Θ, but in this case we did see an entirely trivial calculation:

$$\mathrm{P}_{\texttt{ORDER}}(w) = \frac{\sum_{h:s_h=w} \mathrm{P}_{\mathcal{O}}(w, \overleftarrow{w}, w+1)}{\sum_{h:s_h=w} \mathrm{P}_{\mathcal{O}}(w, \overleftarrow{w}, w+1) + \mathrm{P}_{\mathcal{O}}(w, \overrightarrow{w}, w+1)}.$$

- $\mathrm{P}_{\texttt{ATTACH}}$: Klein neglects to explain how to infer this distribution as well, which was a problem for us, since it wasn't entirely clear what it even means — at first glance, it could be interpreted as the probability that arg attaches to w versus the alternative that w attaches to arg, which may seem reasonable as we decide how to merge two trees. But given the context, we chose to interpret it as the probability that a particular arg (of all word classes) attaches to w, given that something does: the long-run frequency of arg among all of w's attachments.

  We hope that this is the intended interpretation, but we found it to be quite tricky, since again, the tree has aggregated and co-mingled various information, making it difficult (if not impossible) to recover what we want. After experimenting with multiple strategies, we once again resorted to a "crazy hack" (see the `crh` array in our code), since we don't believe there is enough information accessible in the tree itself. Roughly speaking, we hacked together $\texttt{crh}(i, \overset{t}{h}, j)[\overline{a}]$ into the inside phase of the algorithm to track the fraction of the total inside probability $\mathrm{P}_{\mathcal{I}}(i, \overset{t}{h}, j)$ that arrived via attaching the argument $\overline{a}$. We then went on with Klein's earlier line of reasoning, by taking the ratio of actual frequency of attachment, to the total number of opportunities to attach:

$$\mathrm{P}_{\texttt{ATTACH}}(w, \texttt{L}, v) = \frac{\sum_{h:s_h=w} \sum_{i=0}^{h} \sum_{j=h+1}^{l} \sum_{t\in\{\leftarrow,\leftrightarrows\}} \mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j) \sum_{\substack{a:s_a=v \\ i\leq a<h}} \texttt{crh}(i, \overset{t}{h}, j)[\overline{a}]}{\sum_{h:s_h=w} \sum_{i=0}^{h} \sum_{j=h+1}^{l} \sum_{t\in\{\leftarrow,\leftrightarrows\}} \mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j)},$$

$$\mathrm{P}_{\texttt{ATTACH}}(w, \texttt{R}, v) = \frac{\sum_{h:s_h=w} \sum_{i=0}^{h} \sum_{j=h+1}^{l} \sum_{t\in\{\rightarrow,\rightleftarrows\}} \mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j) \sum_{\substack{a:s_a=v \\ h+1\leq a<j}} \texttt{crh}(i, \overset{t}{h}, j)[\overline{a}]}{\sum_{h:s_h=w} \sum_{i=0}^{h} \sum_{j=h+1}^{l} \sum_{t\in\{\rightarrow,\rightleftarrows\}} \mathrm{P}_{\mathcal{O}}(i, \overset{t}{h}, j)}.$$

With $\mathrm{P}_{\texttt{STOP}}$, $\mathrm{P}_{\texttt{ORDER}}$ and $\mathrm{P}_{\texttt{ATTACH}}$ re-estimated for each sentence, we simply point-wise averaged them across sentences, without any kind of smoothing, to generate the next generation model.

5.2.5 Initialization

Klein stresses that initialization is important to the success of any local search procedure and emphasizes that a specific approach was crucial for getting reasonable patterns out of this model. In it, the `ROOT` always had a single argument and took each word with equal probability; all other words took the same number of arguments, each taking other words as arguments in inverse proportion to a constant plus the distance between them. Klein does not specify which constant was used in this somewhat ad-hoc "harmonic" completion. We took all this to imply uniform priors

$$\mathrm{P}_{\texttt{ORDER}} = \mathrm{P}_{\texttt{STOP}} = \frac{1}{2},\ \mathrm{P}_{\texttt{ATTACH}}(\diamondsuit, \texttt{L}, w) = \frac{1}{|W|},$$

with the exceptions

$$\mathrm{P}_{\texttt{ORDER}}(\diamondsuit) = 1,\ \mathrm{P}_{\texttt{STOP}}(\diamondsuit, \texttt{L}, 1) = 0,\ \mathrm{P}_{\texttt{STOP}}(\diamondsuit, \texttt{L}, 0) = \mathrm{P}_{\texttt{STOP}}(\diamondsuit, \texttt{R}, 1) = 1,$$

as well as

$$\mathrm{P}_{\texttt{ATTACH}}(\diamondsuit, \texttt{R}, w) = 0,\ \mathrm{P}_{\texttt{ATTACH}}(s_i, dir, s_j) = \frac{1}{c + |j - i|},\ \text{for } s_i, s_j \neq \diamondsuit.$$

Klein points to two advantages of this approach: 1) it offers a common way for starting off and testing multiple models; and 2) it allows the model to be pointed in the vague general direction of what linguistic dependency structures should look like.

5.3 Constituency Conversions

Since Klein tested his model using the Penn Treebank data, we implemented several more or less trivial algorithms for parsing the relevant tree structures, extracting the flattened terminals, along with their part-of-speech word classes, and converting the constituency parses into dependency parses, for evaluation.

5.3.1 Head Percolation

One of the few interesting stops along this detour is the accepted conversion process, used by Klein, which we first encountered in our course reader. We implemented Collins' rules as follows:

1. Refer to all word class names by their prefixes, up to the first hyphen or equals sign (if any), so that `NP-SBJ` would be an instance of `NP`, `PP-CLR` would be handled as if it were `PP`, `ADVP=3` as `ADVP`, etc.

2. Truncate coordinating phrases by focusing only on the children up to the first `CC` term.

3. If fewer than two children remain, return the first child.

4. Special-case the `NP` rules as follows:

   - If the last word is `POS`, return it;
   - Otherwise, search `R`→`L` for the first child in {`NN`, `NNP`, `NNPS`, `NNS`, `NX`, `POS` or `JJR`};
   - Otherwise, search `L`→`R` for the first child which is an `NP`;
   - Otherwise, search `R`→`L` for the first child in {`$`, `ADJP` or `PRN`};
   - Otherwise, search `R`→`L` for the first child which is a `CD`;
   - Otherwise, search `R`→`L` for the first child in {`JJ`, `JJS`, `RB` or `QP`};
   - Otherwise, return the last word.

5. For all other rules, use the following prioritized heads (from left to right down the list), searching in the specified direction (we interpreted this to mean that, in the absence of a match, `L`→`R` non-terminals would return their first child, while `R`→`L` non-terminals would return their left child):

Parent	*Direction*	*Priority List*
ADJP	R→L	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT FW RBR RBS SBAR RB
ADVP	L→R	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
CONJP	L→R	CC RB IN
FRAG	L→R	
INTJ	R→L	
LST	L→R	LS :
NAC	R→L	NN NNS NNP NNPS NP NAC EX $ CD QP PRP VBG JJ JJS JJR ADJP FW
PP	L→R	IN TO VBG VBN RP FW
PRN	R→L	
PRT	L→R	RP
QP	R→L	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
RRC	L→R	VP NP ADVP ADJP PP
S	R→L	TO IN VP S SBAR ADJP UCP NP
SBAR	R→L	WHNP WHPP WHADVP WHADJP IN DT S SQ SINV SBAR FRAG
SBARQ	R→L	SQ S SINV SBARQ FRAG
SINV	R→L	VBZ VBD VBP VB MD VP S SINV ADJP NP
SQ	R→L	VBZ VBD VBP VB MD VP SQ
UCP	L→R	
VP	R→L	TO VBD VBN MD VBZ VB VBG VBP VP ADJP NN NNS NP
WHADJP	R→L	CC WRB JJ ADJP
WHADVP	L→R	CC WRB
WHNP	R→L	WDT WP WP$ WHADJP WHPP WHNP
WHPP	L→R	IN TO FW

Of the 49,208 sentences in the *The Wall Street Journal* section, we dropped 10 which exceeded length 100 (maximum length was 271). Training on the remaining 49,198 sentences, an additional 590 had to be excluded because they contained non-terminals which our implementation could not percolate all the way up. All of these had to do with productions relating to NX and X; here are those which occurred more than once, sorted by frequency:

NX → NN NNS	109
NX → JJ NN	102
NX → NN NN	86
NX → JJ NNS	82
NX → NNP NNP	66
NX → NX , NX CC NX	49
NX → NX PP	47
NX → JJ JJ NN	26
NX → NNP NNP NNP	23
X → DT JJR	23
NX → JJ NN NN	22
NX → NX , NX , NX CC NX	16
NX → JJ NN NNS	14
NX → NX PRN	13
NX → NNP NN	12
X → DT RBR	11
NX → NP PP	10
NX → NX , CC NX	9
NX → NN NN NN	8
X → X NP	8
NX → NNP NNPS	7
NX → NNS NN	7
NX → ADJP NN	6
NX → CD NN	6
NX → NX , CC NX ,	6
NX → VBN NNS	6
X → X : PP .	6
NX → DT NN	5
NX → NN VBG NN	5
NX → JJ JJ NNS	4
NX → NN NN NNS	4
NX → NNP NNP NNP NNP	4
NX → NX ADJP	4
X → DT ADJP	4
NX → CD JJ NX	3
NX → JJ CD	3
NX → JJ NNP NN	3
NX → NNP NNP NN	3
NX → NNP POS	3
NX → NX , NX , NX , NX CC NX	3
NX → QP	3
NX → RB NNS	3
NX → VBG NN	3
NX → $ CD	2
NX → ADJP NNS	2
NX → CD NNP NX	2
NX → CD NNS	2
NX → DT JJ NN	2
NX → FW FW	2
NX → JJ NN SBAR	2
NX → JJ VBG NN	2
NX → JJS NN	2
NX → NN JJ NN	2
NX → NN PP	2
NX → NN S	2
NX → NNP NNP PP	2
NX → NNP NNS	2
NX → NNS NNS	2
NX → NP NNP	2
NX → NP NNS	2
NX → NP SBAR	2
NX → NP VP	2
NX → NX , NX , CC NX	2
NX → NX , NX , NX	2
NX → NX NP	2
NX → NX PP PP	2
NX → PRP$ NNS	2
NX → VBG NNS	2
NX → VBN JJ NN	2
NX → VBN NN	2
NX → ‘‘ NX ’’ CC ‘‘ NX	2
NX → ‘‘ NX , ’’ SBAR	2
X → ADVP , NP .	2
X → DT JJR NN	2
X → SBAR , X NP	2
X → X : ADJP .	2
X → X : NP .	2
X → X PP	2

5.3.2 (P)CFG for the DMV

In order to score using the same metric as used by Klein, we had to find the most probable parse, given the probabilistic model Θ. Although in itself an interesting exercise, the dynamic programming involved in our standard CYK implementation was quite similar to (just simpler than) the inside chart computation of the inside-outside algorithm and did not offer any intellectual insights beyond what was covered in programming assignment three. Thus, we omit those details here.

6 Evaluation / Testing / Performance / Results

Since a dependency parse always consists of exactly as many dependencies as there are words in the sentence, the quality of a hypothesized dependency structure can be evaluated by accuracy as compared to a gold-standard, by reporting the percentage of shared links between the two analyses.

Klein advocates reporting an accuracy figure for both directed and undirected dependencies, as undirected numbers offer two advantages: 1) they facilitate comparison with earlier work; and, more importantly, 2) they allow one to partially obscure the effects of alternate analyses, such as the systematic choice between a modal and a main verb for the head of a sentence, as in either case the two verbs would be linked, but the direction would vary.

6.1 DMV

We now attempt to decouple the unsupervised learning aspects of this project from the intrinsic value of a potential model, if there were an efficient way to learn it. Luckily, the DMV lends itself to such analyses, given the head-percolated Penn Treebank dependency parses.

6.1.1 A Base-Line

To keep our experimentation cycle reasonably snappy, we lowered the threshold for maximum sentence length down to 25 — $O(n^5)$ is no joking matter, since a single very long sentence can take a disproportionate fraction of the time (see `#define MAX_LEN` in our code). This takes us down to 26,251 reference parses, which is slightly more than half of the total available number of sentences.

Using a "zero knowledge" model, without even the ad-hoc harmonic smoothing — instead, using the uniform prior for everything except for the hard-wired `ROOT` probabilities — already gives a score of 14.4% for the directed edges (29.9% for undirected). Not surprisingly, this is not quite as high as the 33.6% scored by the adjacent-word heuristic, mentioned by Klein.

6.1.2 An Acid-Line?

Since we had already spent quite some time thinking about how to estimate the DVM from data, it occurred to us that we could create an oracle-like model, simply by inferring probabilities from the surface statistics presented by our reference parses. Unfortunately, there is no obvious way to estimate $P_{\texttt{ORDER}}$, so it remains uniform at one half. But $P_{\texttt{ATTACH}}$ could be extracted simply by tracking the fraction of attachments' classes for each head's word class. And $P_{\texttt{STOP}}$ could be gleaned using the simple estimators associated with the Bernoulli and the Geometric probability distributions, as before. This gave us an upper bound of 75.5% for the directed edges (77.5% for undirected). Once again, we are not surprised that these numbers are higher than Klein's scores of 43.2% (63.7% undirected), but it's impressive how close his unsupervised training came to the theoretical limit, modulo a dumbed down $P_{\texttt{ORDER}}$. We find it note-worthy that 1) this theoretical ceiling is so far away from 100%; and 2) that the directed and undirected scores came out so close to one another.

6.1.3 Further Dissection

In order to investigate which parts of the model are contributing to the theoretical ceiling, we tried training it piece-meal: 1) with $P_{\texttt{ATTACH}}$ alone, leaving both $P_{\texttt{ORDER}}$ and $P_{\texttt{STOP}}$ uniform, it scored 60.0% (63.6% undirected); 2) with $P_{\texttt{STOP}}$ alone, without $P_{\texttt{ORDER}}$ or $P_{\texttt{ATTACH}}$, it scored 53.9% (57.7% undirected); 3) using just the adjacent portion of $P_{\texttt{STOP}}$ gave 50.0% (54.8% undirected); and finally, 4) using only the non-adjacent portion of $P_{\texttt{STOP}}$ gave 12.5% (30.8% undirected), which isn't much better than the chance baseline, strictly speaking. This confirms the usefulness of the attachment probabilities and the probabilities of whether or not there are attachments to speak of, but questions the contribution of non-adjacent stopping, at least in its present bucketed form.

6.1.4 Error Analysis

We now present several pseudo-randomly selected sentences, which indicate the kinds of mistakes which are likely to be made even by the oracle-trained DMV:

```
Reference: 1
    0    1    1  D Pierre[NNP]
    1    7    7  D Vinken[NNP]
    2    1    7    ,[,]
    3    4    4  D 61[CD]
    4    5    7    years[NNS]
    5    1    4  U old[JJ]
    6    1    7    ,[,]
    7   18   18  D will[MD]
    8    7    7  D join[VB]
    9   10   10  D the[DT]
   10    8    8  D board[NN]
   11    8   10    as[IN]
   12   14   14  D a[DT]
   13   14   14  D nonexecutive[JJ]
   14   11   11  D director[NN]
   15    8    8  D Nov.[NNP]
   16   15   15  D 29[CD]
   17    7    7  D .[.]
  pos head guess
```

Above, we notice that the model struggles with punctuations, such as commas.

```
Reference: 115
    0    1    1  D All[DT]
    1    6    6  D came[VBD]
    2    1    1  D from[IN]
    3    4    4  D Cray[NNP]
    4    2    2  D Research[NNP]
    5    1    1  D .[.]
  pos head guess
```

And here is a rare (small) case when it gets entirely right!

Rather than eye-ball the sentences and spot-check sporadically on, we implemented a verbose option in our program (see `#define VERBOSE`, which, among other things, produces a sorted dump of all of the mistakes. Specifically, we track the correct attachments when the model is entirely wrong, as well as the incorrect attachments that were suggested, in addition to the swapped attachments when the model is almost right. Many of the top offenders can also be found at the top of Klein's list of top over- and under-proposed dependency types.

Wrongly Claimed (1011 Unique)			
NN	→	IN	5925
DT	←	NN	4642
NNP	←	NNP	3568
JJ	←	NN	3396
IN	→	NN	2971
NNS	→	IN	2260
VBN	→	IN	2175
VBD	→	.	1891
NN	←	NN	1855
IN	→	NNS	1663
NNP	→	,	1561
VBD	→	NN	1297
VBZ	→	.	1285
NN	→	,	1269
NN	←	VBD	1179
NN	←	NNS	1096
VBD	←	DIAMOND	1084
NN	←	VBZ	1082
NN	→	CC	1037
,	←	VBD	1027
NNS	→	CC	1000
IN	→	NNP	971
VBD	→	-NONE-	967
VBD	→	TO	954
VBD	→	IN	946
NNP	←	NN	943
VBZ	←	DIAMOND	935
NNS	←	VBD	932
NNS	→	,	888
NNS	←	VBP	873
JJ	←	NNS	859
VBZ	→	NN	844
,	←	VBZ	678
VB	→	NN	661
VBD	→	VBD	651
CD	←	NN	647
VBP	→	RB	623
RB	←	VBN	617
VBD	→	RB	597
VBP	→	.	596
RB	←	JJ	582
NNP	→	CC	579
-NONE-	←	TO	571
NNP	←	VBD	556
VBN	→	-NONE-	551
VBZ	→	RB	545
CD	←	NNS	523
DT	←	NNS	506
POS	←	NN	502
PRP$	←	NN	456
IN	←	VBD	451
VB	→	NNS	446
VBD	→	,	435
``	←	VBZ	420
VBD	→	NNS	393
NNP	→	NNP	386
VBZ	→	TO	385
-NONE-	←	VBZ	381
-NONE-	←	-NONE-	361
VBZ	→	VBG	358
MD	→	RB	357
VBZ	→	IN	355
VBP	←	DIAMOND	353
VBG	→	NN	351
RB	←	VBD	346
NNP	←	VBZ	341
VBD	→	VBN	337

Totally Missed (1549 Unique)			
DT	←	NN	2998
NN	→	IN	2525
VBD	→	IN	1920
NN	→	NN	1696
VB	→	IN	1668
IN	→	NN	1663
NN	←	NN	1606
NNP	←	NN	1537
NNP	←	NNP	1500
JJ	←	NN	1486
NNS	→	IN	1383
NNP	→	NNP	1375
VBD	→	.	1354
JJ	←	NNS	1228
DT	←	POS	1074
VBZ	→	.	1067
NN	→	,	1034
NNP	→	,	991
NN	←	NNS	958
IN	→	NNS	866
VBP	→	.	810
IN	→	NNP	802
NNS	→	NNS	774
VBD	→	NN	771
NNP	→	NN	737
DT	←	NNS	729
NNP	←	NNS	729
-NONE-	←	VBD	725
VBD	→	,	714
VBG	→	IN	709
,	←	VBD	699
VBZ	→	IN	694
IN	→	VBD	676
VBN	→	IN	667
NN	→	NNS	639
JJ	→	IN	608
NN	→	CC	596
VBZ	→	RB	587
VBD	→	CC	586
NN	←	VBD	575
NNS	→	NN	570
IN	→	CD	561
NN	→	TO	561
MD	→	.	521
VBD	→	RB	512
RB	←	NN	497
RB	→	IN	478
NNP	→	IN	477
RB	←	IN	474
IN	→	VBZ	467
NNS	→	,	466
VBZ	→	,	466
DT	←	NNP	451
NN	→	-NONE-	449
VBZ	→	NN	444
NNS	←	NN	438
NN	←	VBZ	435
CC	→	NN	431
VBP	→	IN	424
CC	→	NNS	401
VBZ	→	CC	399
NNS	←	VBD	390
VBD	←	DIAMOND	382
VBZ	←	DIAMOND	376
,	←	VBZ	372
JJ	→	JJ	361
JJ	→	TO	358

Simply Swapped (422 Unique)			
NNP	←	NNP	760
NN	←	NN	494
IN	←	VBD	275
IN	→	NNS	234
IN	→	NN	226
VBG	→	NN	205
VBD	→	VBD	182
VBG	→	NNS	174
NNP	←	NN	166
IN	→	$	159
NN	←	NNS	159
VBZ	→	VBZ	156
IN	←	VBZ	155
-NONE-	→	VBP	128
CD	→	CD	120
-NONE-	→	VBD	118
-NONE-	→	VBZ	113
NNP	→	NNP	109
JJR	→	IN	107
RB	←	IN	100
VBN	→	NNS	96
NNP	→	NN	94
NN	←	.	88
IN	←	MD	83
,	←	VBD	82
VBN	→	NN	75
IN	←	VBP	74
RB	→	IN	74
-NONE-	→	MD	73
RB	←	JJ	73
IN	→	IN	71
,	←	VBZ	69
-NONE-	←	VBZ	64
VBZ	←	VBZ	64
VBZ	←	VBD	63
JJ	→	IN	62
VBD	→	VBZ	59
JJR	←	IN	58
NN	→	IN	56
JJ	←	NN	52
-NONE-	←	VBD	50
-NONE-	←	MD	45
NNS	→	VBN	45
-NONE-	←	VBP	44
NNS	→	JJ	44
NNS	←	TO	44
VBP	←	VBZ	43
NNPS	→	NNP	42
NNS	←	.	42
CD	←	NN	40
NN	←	VBD	39
NNS	→	IN	39
JJ	←	NNS	37
NN	→	VBN	37
NNS	→	NNS	37
VBD	←	VBD	36
NNP	←	.	34
NNP	←	NNS	34
,	→	VBZ	32
VBG	←	NN	32
VBG	←	NNS	32
VBP	←	VBD	32
IN	→	RB	31
IN	→	VBZ	31
NN	←	TO	30
IN	→	VBD	28
IN	←	NN	27

We note that the `ROOT` $\diamondsuit$ isn't to be found at the top of any lists (not that it could appear in the swapped list), which creates hope that the model is fairly good at identifying the overall head. At the same time, we observe an abundance of improper subordination to punctuation and lots of confusing `-NONE-` nodes.

We hesitate to draw any serious linguistical claims from this, since our model is inbred directly from the Treebank parses themselves. Klein, however, stresses that in his experience the model's top mis-matches come from systematically choosing determiners to be the heads of noun phrases, where the test trees have the right-most noun as the head, clarifying that the model's choice is supported by a good deal of linguistic research (in fact, Klein modified the head-percolation rules to favor determiners when present, in case of `NP`, and quoted improved scores of 55.7% directed, 67.9% undirected, with this modification). Other common sources of errors included modals dominating main verbs, the choice of the wrong noun as the head of a noun cluster, and having some sentences headed by conjunctions.

6.2 EM

In this sub-section, we focus on how well (our implementation) of expectation maximization is able to arrive at a decent set of probability distributions.

6.2.1 A Base-Line

As advertised, without special initialization, EM doesn't take us much farther from the zero-knowledge base-line: 18.4% directed, 38.4% undirected. Strangely, EM gets these scores on the very first iteration, then makes very little progress going forward, usually slipping back...

We programmed EM to bail out when the maximum point-wise change in probability has gone down sufficiently low, or when the number of iterations of the algorithm has reached a threshold (see `#define MIN_EM_ERR` and `#define MAX_EM_ITER`). But in addition to printing the maximum point-wise change found so far, we also make use of the oracle's probability distributions and take advantage of our ability to track the average discrepancy in point-wise probabilities between our current best and the goal.

In this case, the l_1 losses are 1.72% for $\mathrm{P}_{\mathtt{ATTACH}}$, 60.60% for $\mathrm{P}_{\mathtt{STOP}}^{adj}$ and 38.29% for $\mathrm{P}_{\mathtt{STOP}}^{\overline{adj}}$.

6.2.2 An Ad-Hoc Harmonic

As advised, we tried initializing our model using the ad-hoc harmonic function for valence-sensitive $\mathrm{P}_{\mathtt{ATTACH}}$, which immediately brought the scores up to 21.5% directed, 47.1% undirected, before even the first iteration of EM. Here, we used a constant of 5, though we found that many small numbers perform almost equally as well.

The very first iteration of EM makes things slightly worse: 20.9% directed, 45.7% undirected. Sadly, things don't improve much from there, with the losses lingering around 1.73% for $\mathrm{P}_{\mathtt{ATTACH}}$, 59.66% for $\mathrm{P}_{\mathtt{STOP}}^{adj}$ and 38.29% for $\mathrm{P}_{\mathtt{STOP}}^{\overline{adj}}$.

6.2.3 An Acid-Line?!?

Curious about how EM would behave if we started it at the optimum, we were disappointed to find it drop after the very first step: 72.5% directed, 74.5% undirected, and drifting down, with losses of 1.65% for $\mathrm{P}_{\mathtt{ATTACH}}$, 55.01% for $\mathrm{P}_{\mathtt{STOP}}^{adj}$ and 38.09% for $\mathrm{P}_{\mathtt{STOP}}^{\overline{adj}}$.

6.2.4 Further Dissection

In order to investigate which parts of the model are contributing to this degradation, we short-circuited the updating rules for both $\mathrm{P}_{\mathtt{ATTACH}}$ an $\mathrm{P}_{\mathtt{STOP}}$, leaving only $\mathrm{P}_{\mathtt{ORDER}}$ open to mutation. EM still managed to lose a little bit of the edge, but nevertheless maintained the high level set by the oracle almost intact, around 75.5% directed, 77.4% undirected after some iterations.

With $P_{\texttt{ATTACH}}$ alone updating, the scores drop immediately to 72.4% directed, 74.5% undirected, then continue to slide, reaching 68.1% directed, 70.5% undirected after just a few iterations, thus $P_{\texttt{ATTACH}}$ is suspect; with $P_{\texttt{STOP}}$ alone updating, the scores drop even faster, reaching 66.8% directed, 69.9% undirected after just a few iterations, rendering $P_{\texttt{STOP}}$ suspect as well and casting doubts over (our implementation of) EM.

6.2.5 Error Analysis

We are not entirely sure what is going wrong with our implementation of EM. There are many potential reasons: it's possible that we misunderstood the model, although the oracle's solid performance makes this unlikely. It could be that the surface estimate is better than what EM could do, since it gathers statistics across all sentences before averaging, whereas EM is forced to aggregate at the sentence level. There are the more likely possibilities that either our re-estimation is incorrect or that we have bugs in the innards of the inside-outside algorithm (or both). Still, the fact that the oracle performs well suggests that our (P)CFG is good, and that code is quite similar to what's in IO...

7 Forward-Looking Statements

Despite the botched experience with EM, we feel that the DMV is a simple, powerful and extensible model, based on our experiments with the oracle and Klein's unsupervised successes. We are very interested in exploring the issue of scaling to much larger corpora — perhaps even using the oracle itself as a seed for an unsupervised algorithm, once we figure out why our implementation of EM can't hang on to a good solution even when one is staring at it... It's tempting to think of the ways in which the DMV could be extended (e.g. lexicalization, better bucketing, perhaps a Poisson distribution over the number of children, which worked well for us in the second programming assignment on EM for word alignment models). Nevertheless, we hear Klein's advice and plan to be careful:

In contrast to what's generally done in supervised parsers, the DMV's lexical identity is a word class, not the identity of the head word itself. However, in Klein's unsupervised experiments, having lexical items in the model led to distant topical associations being preferentially modeled over class-level syntactic patterns. Supervised parsers' decisions to stop or continue generating arguments are also typically conditioned on finer notions of distance than simple adjacency (e.g. using buckets or punctuation-defined distance). Moreover, decisions about argument identity are conditioned on the identities of any previous arguments, not just a binary indicator of whether there were any... But, in the unsupervised case, this much freedom can be dangerous. Such features' success in supervised systems suggests that they could be exploited in this setting too, perhaps in a system which originally ignored richer context, then gradually began to model it. Part of the beauty of this model is in its simplicity: Klein reports that it worked quite well (at above-base-line levels) for both Chinese and German, without any modifications.

8 Conclusions

Even though the EM experiment did not work out, we still learned a great deal about NLP over the course of the last few weeks, exploring the fields of unsupervised and dependency parsing.

Thank you for a great class!! :)

9 Bibliography

Collins, M. Head-Driven Statistical Models for Natural Language Parsing. 1999.
Ph.D. Thesis, MIT.

Jurafsky, D., and J.H.Martin. Speech and Language Processing:
An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.
2008. 2nd ed.

Klein, D. The Unsupervised Learning of Natural Language Structure. 2005.
Ph.D. Thesis, Stanford University.

Klein, D., and C.D.Manning. Corpus-Based Induction of Syntactic Structure:
Models of Dependency and Constituency. In *ACL 42*, 479-486.

McDonald, R. and J.Nivre. Introduction to Data Driven Dependency Parsing.
`http://dp.esslli07.googlepages.com/`

Xia, F. Inner loop of the Inside-outside algorithm.
`http://courses.washington.edu/ling570/fei_fall07/12_5_Summary.pdf`

Xia, F. Inside-outside algorithm.
`http://faculty.washington.edu/fxia/courses/LING572/inside-outside.ppt`

A Makefile

```
#!gmake

fp: fp.cc
	g++ -o fp -O6 fp.cc
	strip fp
```

B Data

```
cat /usr/class/cs224n/pa3/data/parser/ptb/wsj/??/wsj_????.mrg > /tmp/ptb.wsj.full
```

C Usage

```
./fp < /tmp/ptb.wsj.full > /tmp/ptb.wsj.full.out
```

D The Code

Although what follows has neither been polished for readability nor optimized for efficiency, we hope that — at least in the context of the preceding report — our code strikes you as moderately clean and intelligible enough to be useful.

```
// Valentin I. Spitkovsky
//
// CS224N: Spring 2008
// Prof. Christopher D. Manning


// Final Project: Dependency Model with Valence
//                   (Unsupervised Learning)
// 2008-06-04


#define VERBOSE (false)
#define MAX_EM_ITER (20)
#define MIN_EM_ERR (1.0/(1024))
#define MAX_TOL (1.0/(1024*1024))
#define MAX_LEN ((25)+1)  // a cap on sentence lengths, for efficiency...

static bool harmonic = false;
static double harmonic_c = 5;

#include <string>
#include <vector>

#include <ext/hash_map>

#include <iomanip>
#include <iostream>

#include <assert.h>
#include <math.h>

using namespace std;
using namespace __gnu_cxx;

namespace __gnu_cxx {
  template<> struct hash<std::string> {
    size_t operator()(const std::string &x) const {
      return hash<const char *>()(x.c_str());
    }
  };
}

const string TERMINATOR = "DIAMOND";

enum {
  IN = 0,  // inside
  OUT,     // outside
  N_DIRS
};

enum {
  L = 0,   // left
  R,       // right
  LR,      // right, left
  RL,      // left, right
  S,       // sealed
  N_TYPES
};

typedef struct {
  double x[2][2];
} stop_type;
typedef struct {
  double *x[2];
} attach_type;

static vector<string> words;
typedef hash_map<string,unsigned> hm_su_type;
static hm_su_type word_ids, unheaded;

static string word(const unsigned id) {
  assert(id < words.size());
  return words[id];
}
static unsigned word_id(const string &w) {
  assert(w.size() > 0);
  const hm_su_type::const_iterator q = word_ids.find(w);
  if (q == word_ids.end()) {
    const unsigned int len = words.size();
    words.push_back(w);
    word_ids[w] = len;
    return len;
  }
  return q->second;
}


static inline double Porder(  // left-first
    const unsigned wc,
    const double *const p) {
  return p[wc];
```

```
}

static inline bool adj(
    const unsigned dir,
    const unsigned pos,
    const unsigned w) {
  switch(dir) {
    case L:
      assert(pos <= w);
      return (pos == w);
    case R:
      assert(pos > w);
      return (pos == w + 1);
    default:
      assert(false);
  }
}

static inline double Pstop(  // stop
    const unsigned wc,
    const unsigned dir,
    const unsigned pos,
    const unsigned w,
    const stop_type *const p) {
  return p[wc].x[dir][adj(dir,pos,w)];
}

static inline double Pattach(const unsigned ac,
                             const unsigned wc,
                             const unsigned a,
                             const unsigned w,
                             const attach_type *const p) {
  if (harmonic) {  // ad-hoc harmonic...
    return 1 / (harmonic_c + fabs(a - w));
  }
  return p[ac].x[a < w ? L : R][wc];
}

static inline void process(
    const vector<unsigned> &sentence,
    const double *const p_order,
    double *const l_order,
    double *const r_order,
    const stop_type *const p_stop,
    stop_type *const c_stop,
    stop_type *const t_stop,
    const attach_type *const p_attach,
    attach_type *const c_attach,
    double t_attach[][2],
    const unsigned nwords) {
  const unsigned len = sentence.size();
  assert(len <= MAX_LEN);
  assert(len > 0);

  // The generic charts for computing Inside and Outside probabilites:
  static double (*const mem)[MAX_LEN][N_TYPES][MAX_LEN][MAX_LEN+1]
      = new double[N_DIRS][MAX_LEN][N_TYPES][MAX_LEN][MAX_LEN+1];
  // For each of two directions (IN / OUT), for every head word h, and
  // for its every possible annotation (L,R,LR,RL,S), associates a
  // computation for the range i...j.

  // A crazy hack for learning Pattach -- for every head word h, tracks
  // the contributions of attachments a, when spanning i...j (over IN):
  static double (*const crh_attach)[MAX_LEN][MAX_LEN][MAX_LEN+1]
      = new double[MAX_LEN][MAX_LEN][MAX_LEN][MAX_LEN+1];

  // Another crazy hack for learning Pstop -- for some (transient) head
  // word h, track enough information to compute the fraction of
  // childless sub-trees, as well as the average number of children,
  // in both directions:
  static double crh_zero[N_DIRS], crh_weighted[N_DIRS], crh_sum[N_DIRS];

  for (unsigned h = 0; h < len; ++h) {
    for (unsigned i = 0; i <= h; ++i) {
      for (unsigned j = h + 1; j <= len; ++j) {
        for (unsigned d = 0; d < N_DIRS; ++d) {
          for (unsigned t = 0; t < N_TYPES; ++t) {
            mem[d][h][t][i][j] = 0;
          }
        }
        for (unsigned a = i; a < h; ++a) {
          crh_attach[h][a][i][j] = 0;
        }
        for (unsigned a = h + 1; a < j; ++a) {
          crh_attach[h][a][i][j] = 0;
        }
      }
    }
  }

  for (unsigned i = 0; i < nwords; ++i) {
```

```
  l_order[i] = r_order[i] =
      c_stop[i].x[L][true] = c_stop[i].x[L][false] =
      c_stop[i].x[R][true] = c_stop[i].x[R][false] =
      t_stop[i].x[L][true] = t_stop[i].x[L][false] =
      t_stop[i].x[R][true] = t_stop[i].x[R][false] =
      t_attach[i][L] = t_attach[i][R] = 0;
  for (unsigned j = 0; j < nwords; ++j) {
    c_attach[i].x[L][j] = c_attach[i].x[R][j] = 0;
  }
}

// Inside (Bottom-Up)
for (unsigned i = 0; i < len; ++i) {
  const double p = Porder(sentence[i], p_order);
  mem[IN][i][L][i][i+1] = p;
  mem[IN][i][R][i][i+1] = 1 - p;
  mem[IN][i][LR][i][i+1] = mem[IN][i][RL][i][i+1]
      = mem[IN][i][S][i][i+1] = 0;
}

for (unsigned d = 1; d <= len; ++d) {
  for (unsigned i = 0; ; ++i) {
    const unsigned j = i + d;
    if (j > len)
      break;

    for (unsigned w = i; w < j; ++w) {
      const unsigned wc = sentence[w];

      unsigned k = i;
      while (++k <= w) {
        const double
            go = 1 - Pstop(wc, L, k, w, p_stop),
            go_l = go * mem[IN][w][L][k][j],
            go_lr = go * mem[IN][w][LR][k][j],
            go_sum = go_l + go_lr;

        if (go_sum > 0) {
          double p = 0;
          for (unsigned a = i; a < k; ++a) {
            const unsigned ac = sentence[a];
            const double q = Pattach(ac, wc, a, w, p_attach)
                * mem[IN][a][S][i][k];
            p += q;
            crh_attach[w][a][i][j] += q * go_sum;
          }
          mem[IN][w][L][i][j] += p * go_l;
          mem[IN][w][LR][i][j] += p * go_lr;
        }
      }

      for (; k < j; ++k) {
        const double
            go = 1 - Pstop(wc, R, k, w, p_stop),
            go_r = go * mem[IN][w][R][i][k],
            go_rl = go * mem[IN][w][RL][i][k],
            go_sum = go_r + go_rl;

        if (go_sum > 0) {
          double p = 0;
          for (unsigned a = k; a < j; ++a) {
            const unsigned ac = sentence[a];
            const double q = Pattach(ac, wc, a, w, p_attach)
                * mem[IN][a][S][k][j];
            p += q;
            crh_attach[w][a][i][j] += q * go_sum;
          }
          mem[IN][w][R][i][j] += p * go_r;
          mem[IN][w][RL][i][j] += p * go_rl;
        }
      }

      const double
          sl = Pstop(wc, L, i, w, p_stop),
          sr = Pstop(wc, R, j, w, p_stop);

      mem[IN][w][LR][i][j] += mem[IN][w][R][i][j] * sr;
      mem[IN][w][RL][i][j] += mem[IN][w][L][i][j] * sl;

      mem[IN][w][S][i][j]
          += mem[IN][w][RL][i][j] * sr
          +  mem[IN][w][LR][i][j] * sl;
    }
  }
}

// Normalize the Crazy Hack (Attachment Probabilities)
for (unsigned w = 0; w < len; ++w) {
  for (unsigned i = 0; i <= w; ++i) {
    for (unsigned j = w + 1; j <= len; ++j) {
      {  // left
```

```
        const double lnorm = mem[IN][w][L][i][j] + mem[IN][w][LR][i][j];
        if (lnorm > 0) {
          for (unsigned a = i; a < w; ++a) {
            crh_attach[w][a][i][j] /= lnorm;
          }
        }
      }
      {  // right
        const double rnorm = mem[IN][w][R][i][j] + mem[IN][w][RL][i][j];
        if (rnorm > 0) {
          for (unsigned a = w + 1; a < j; ++a) {
            crh_attach[w][a][i][j] /= rnorm;
          }
        }
      }
    }
  }
}

// Outside (Top-Down)
mem[OUT][len-1][S][0][len] = mem[OUT][len-1][LR][0][len] = 1;
mem[OUT][len-1][R][0][len] = (len == 1 ? 1 : 0);

for (unsigned d = len - 1; d > 0; --d) {
  for (unsigned i = 0; ; ++i) {
    const unsigned j = i + d;
    if (j > len)
      break;

    for (unsigned w = i; w < j; ++w) {
      const unsigned wc = sentence[w];

      for (unsigned k = 0; k < i; ++k) {
        double p = 0;
        for (unsigned a = k; a < i; ++a) {
          const unsigned ac = sentence[a];
          p += Pattach(ac, wc, a, w, p_attach) * mem[IN][a][S][k][i];
          mem[OUT][w][S][i][j]
              += Pattach(wc, ac, w, a, p_attach)
              * (1 - Pstop(ac, R, i, a, p_stop))
              * ((mem[IN][a][R ][k][i] * mem[OUT][a][R ][k][j])
                +(mem[IN][a][RL][k][i] * mem[OUT][a][RL][k][j]));
        }
        p *= 1 - Pstop(wc, L, i, w, p_stop);
        mem[OUT][w][L][i][j] += p * mem[OUT][w][L][k][j];
        mem[OUT][w][LR][i][j] += p * mem[OUT][w][LR][k][j];
      }

      for (unsigned k = j + 1; k <= len; ++k) {
        double p = 0;
        for (unsigned a = j; a < k; ++a) {
          const unsigned ac = sentence[a];
          p += Pattach(ac, wc, a, w, p_attach) * mem[IN][a][S][j][k];
          mem[OUT][w][S][i][j]
              += Pattach(wc, ac, w, a, p_attach)
              * (1 - Pstop(ac, L, j, a, p_stop))
              * ((mem[IN][a][L ][j][k] * mem[OUT][a][L ][i][k])
                +(mem[IN][a][LR][j][k] * mem[OUT][a][LR][i][k]));
        }
        p *= 1 - Pstop(wc, R, j, w, p_stop);
        mem[OUT][w][R][i][j] += p * mem[OUT][w][R][i][k];
        mem[OUT][w][RL][i][j] += p * mem[OUT][w][RL][i][k];
      }

      const double
          sl = Pstop(wc, L, i, w, p_stop),
          sr = Pstop(wc, R, j, w, p_stop);

      mem[OUT][w][RL][i][j] += mem[OUT][w][S][i][j] * sr;
      mem[OUT][w][LR][i][j] += mem[OUT][w][S][i][j] * sl;

      mem[OUT][w][R][i][j] += mem[OUT][w][LR][i][j] * sr;
      mem[OUT][w][L][i][j] += mem[OUT][w][RL][i][j] * sl;
    }
  }
}

// Sanity Checks; destroy charts to make counts...
const double p_sentence = mem[IN][len-1][S][0][len];
for (unsigned w = 0; w < len; ++w) {
  const unsigned wc = sentence[w];

  const double
      p = Porder(wc, p_order),
      alternative = mem[OUT][w][L][w][w+1] * p
                  + mem[OUT][w][R][w][w+1] * (1-p);
  assert(fabs(p_sentence - alternative) < MAX_TOL);

  for (unsigned i = 0; i <= w; ++i) {
    for (unsigned j = w + 1; j <= len; ++j) {
      for (unsigned t = 0; t < N_TYPES; ++t) {
```

```
        mem[0][w][t][i][j] *= mem[1][w][t][i][j];
      }
    }
  }

  // re-estimate counts for Porder
  l_order[wc] += mem[0][w][L][w][w+1];
  r_order[wc] += mem[0][w][R][w][w+1];

  // prepare to Aggregate the other Crazy Hack (Stopping Probabilities)
  for (unsigned d = 0; d < N_DIRS; ++d) {
    crh_zero[d] = crh_weighted[d] = crh_sum[d] = 0;
  }

  // re-estimate counts for Pattach
  for (unsigned i = 0; i <= w; ++i) {
    for (unsigned j = w + 1; j <= len; ++j) {
      {  // but first, aggregate the other crazy hack (see above)
        const double p = mem[0][w][S][i][j];
        for (unsigned d = 0; d < N_DIRS; ++d) {
          const unsigned nkids = (d == L ? w - i : j - w - 1);
          if (nkids == 0) {
            crh_zero[d] += p;
          } else {
            crh_weighted[d] += p * nkids;
          }
          crh_sum[d] += p;
        }
      }

      {  // left
        const double pl = mem[0][w][L][i][j] + mem[0][w][LR][i][j];
        for (unsigned a = i; a < w; ++a) {
          const unsigned ac = sentence[a];
          c_attach[ac].x[L][wc] += pl * crh_attach[w][a][i][j];
        }
        t_attach[wc][L] += pl;
      }
      {  // right
        const double pr = mem[0][w][R][i][j] + mem[0][w][RL][i][j];
        for (unsigned a = w + 1; a < j; ++a) {
          const unsigned ac = sentence[a];
          c_attach[ac].x[R][wc] += pr * crh_attach[w][a][i][j];
        }
        t_attach[wc][R] += pr;
      }
    }
  }

  // re-estimate counts for Pstop
  for (unsigned d = 0; d < N_DIRS; ++d) {
    const double z = crh_zero[d], w = crh_weighted[d], s = crh_sum[d];
    c_stop[wc].x[d][true] += z;
    t_stop[wc].x[d][true] += s;
    const double q = s - z;
    if (q > 0) {
      c_stop[wc].x[d][false] += q / (w + 1);
      t_stop[wc].x[d][false] += q;
    }
  }
}

assert(l_order[0] == 0);
assert(c_stop[0].x[L][true] == 0);
//assert(fabs(c_stop[0].x[R][true] - t_stop[0].x[R][true]) < MAX_TOL);
//assert(fabs(c_stop[0].x[L][false] - t_stop[0].x[L][false]) < MAX_TOL);
assert(t_stop[0].x[R][false] == 0);
}

typedef struct dep_type {
  string::size_type overlord;
  string word, word_class;
  unsigned class_id;
};
typedef struct parse {
  string name, prefix;
  string::size_type head;
  vector<parse> children;
  vector<dep_type> deps;
  bool dep_ok;
};
static inline string::size_type left_head(
    const string pq[],
    const unsigned pqlen,
    const vector<parse> &children) {
  const unsigned kids = children.size();
  for (unsigned j = 0; j < pqlen; ++j) {
    for (unsigned i = 0; i < kids; ++i) {
      if (children[i].prefix == pq[j]) {
        return i;
      }
```

```
    }
  }
  if (pqlen > 0) {
    ;  // sign of trouble?
  }
  return 0;
}
static inline string::size_type right_head(
    const string pq[],
    const unsigned pqlen,
    const vector<parse> &children) {
  const unsigned kids = children.size();
  for (unsigned j = 0; j < pqlen; ++j) {
    for (unsigned i = kids; i-- > 0; ) {
      if (children[i].prefix == pq[j]) {
        return i;
      }
    }
  }
  if (pqlen > 0) {
    ;  // sign of trouble?
  }
  return kids - 1;
}
static inline unsigned handle_cc(const vector<parse> &children) {
  const unsigned kids = children.size();
  unsigned effective = 0;
  while (effective < kids) {
    if (children[effective].prefix == "CC") {
      break;
    } else {
      ++effective;
    }
  }
  return effective;
}
static inline string::size_type find_head(
    const string &prefix,
    const vector<parse> &children) {
  const unsigned kids = handle_cc(children);
  if (kids < 2) {
    return 0;
  }

  const unsigned last = kids - 1;
  if (prefix == "NP") {
    if (children[last].prefix == "POS") {
      return last;
    }
    for (unsigned i = kids; i-- > 0; ) { // right
      const string &s = children[i].prefix;
      if (s == "NN" || s == "NNP" || s == "NNPS" || s == "NNS" ||
          s == "NX" || s == "POS" || s == "JJR") {
        return i;
      }
    }
    for (unsigned i = 0; i < kids; ++i) {  // left
      if (children[i].prefix == "NP") {
        return i;
      }
    }
    for (unsigned i = kids; i-- > 0; ) {  // right
      const string &s = children[i].prefix;
      if (s == "$" || s == "ADJP" || s == "PRN") {
        return i;
      }
    }
    for (unsigned i = kids; i-- > 0; ) {  // right
      if (children[i].prefix == "CD") {
        return i;
      }
    }
    for (unsigned i = kids; i-- > 0; ) {  // right
      const string &s = children[i].prefix;
      if (s == "JJ" || s == "JJS" || s == "RB" || s == "QP") {
        return i;
      }
    }
    return last;
  } else if (prefix == "ADJP") {
    static const string pq[] = {
      "NNS", "QP", "NN", "$", "ADVP", "JJ", "VBN", "VBG", "ADJP",
      "JJR", "NP", "JJS", "DT", "FW", "RBR", "RBS", "SBAR", "RB"
    };
    return left_head(pq, sizeof(pq) / sizeof(string), children);
  } else if (prefix == "ADVP") {
    static const string pq[] = {
      "RB", "RBR", "RBS", "FW", "ADVP", "TO", "CD", "JJR",
      "JJ", "IN", "NP", "JJS", "NN"
    };
    return right_head(pq, sizeof(pq) / sizeof(string), children);
```

```
} else if (prefix == "CONJP") {
  static const string pq[] = {
    "CC", "RB", "IN"
  };
  return right_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "FRAG") {
  return right_head(NULL, 0, children);
} else if (prefix == "INTJ") {
  return left_head(NULL, 0, children);
} else if (prefix == "LST") {
  static const string pq[] = {
    "LS", ":"
  };
  return right_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "NAC") {
  static const string pq[] = {
    "NN", "NNS", "NNP", "NNPS", "NP", "NAC", "EX", "$", "CD",
    "QP", "PRP", "VBG", "JJ", "JJS", "JJR", "ADJP", "FW"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "PP") {
  static const string pq[] = {
    "IN", "TO", "VBG", "VBN", "RP", "FW"
  };
  return right_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "PRN") {
  return left_head(NULL, 0, children);
} else if (prefix == "PRT") {
  static const string pq[] = {
    "RP"
  };
  return right_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "QP") {
  static const string pq[] = {
    "$", "IN", "NNS", "NN", "JJ", "RB", "DT", "CD", "NCD",
    "QP", "JJR", "JJS"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "RRC") {
  static const string pq[] = {
    "VP","NP", "ADVP", "ADJP", "PP"
  };
  return right_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "S") {
  static const string pq[] = {
    "TO", "IN", "VP", "S", "SBAR", "ADJP", "UCP", "NP"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "SBAR") {
  static const string pq[] = {
    "WHNP", "WHPP", "WHADVP", "WHADJP", "IN", "DT",
    "S", "SQ", "SINV", "SBAR", "FRAG"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "SBARQ") {
  static const string pq[] = {
    "SQ", "S", "SINV", "SBARQ", "FRAG",
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "SINV") {
  static const string pq[] = {
    "VBZ", "VBD", "VBP", "VB", "MD", "VP", "S", "SINV",
    "ADJP", "NP"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "SQ") {
  static const string pq[] = {
    "VBZ", "VBD", "VBP", "VB", "MD", "VP", "SQ"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "UCP") {
  return right_head(NULL, 0, children);
} else if (prefix == "VP") {
  static const string pq[] = {
    "TO", "VBD", "VBN", "MD", "VBZ", "VB", "VBG", "VBP", "VP",
    "ADJP", "NN", "NNS", "NP"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "WHADJP") {
  static const string pq[] = {
    "CC", "WRB", "JJ", "ADJP"
  };
  return left_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "WHADVP") {
  static const string pq[] = {
    "CC", "WRB"
  };
  return right_head(pq, sizeof(pq) / sizeof(string), children);
} else if (prefix == "WHNP") {
  static const string pq[] = {
    "WDT","WP", "WP$","WHADJP", "WHPP", "WHNP"
```

```
    };
    return left_head(pq, sizeof(pq) / sizeof(string), children);
  } else if (prefix == "WHPP") {
    static const string pq[] = {
      "IN", "TO", "FW"
    };
    return right_head(pq, sizeof(pq) / sizeof(string), children);
  }

  return string::npos;
}
static inline parse make_parse(const char * &s) {
  parse p;
  p.dep_ok = true;

  while (*s == ' ') ++s;
  const bool terminal = (*s != '(');
  const char target = (terminal ? ')' : ' ');

  const char *q = s;
  while (*(++q) != target);

  if (terminal) {
    p.name.assign(s, q - s);
    s = q;
  } else {
    p.name.assign(s + 1, q - s - 1);
    if (p.name.size() == 0) {
      p.name = p.prefix = "BLANK";
    } else {
      const string::size_type
          hyphen = p.name.find('-'),
          equals = p.name.find('=');
      string::size_type split = hyphen;
      if (equals != string::npos &&
          (hyphen == string::npos || hyphen > equals)) {
        split = equals;
      }
      if (split != string::npos) {
        p.prefix = p.name.substr(0, split);
      } else {
        p.prefix = p.name;
      }
    }
    while (true) {
      while (*q == ' ') ++q;
      if (*q == ')') {
        s = q + 1;
        break;
      } else {
        const parse pq = make_parse(q);
        p.children.push_back(pq);
        if (!pq.dep_ok) {
          p.dep_ok = false;
        }
      }
    }
  }
  p.head = find_head(p.prefix, p.children);
  const unsigned kids = p.children.size();
  if (p.head == string::npos) {
    string prod = p.prefix + " -->";
    for (unsigned i = 0; i < kids; ++i) {
      prod += " " + p.children[i].prefix;
    }
    const hm_su_type::iterator q = unheaded.find(prod);
    if (q == unheaded.end()) {
      unheaded.insert(hm_su_type::value_type(prod, 1));
    } else {
      ++q->second;
    }
    p.dep_ok = false;
  }

  if (!terminal && p.dep_ok) {
    const unsigned nkids = p.children.size();
    bool done = false;

    if (nkids == 1) {
      const parse &q = p.children[0];
      if (q.children.empty()) {
        dep_type d;
        d.word = q.name;
        d.word_class = p.name;
        d.overlord = string::npos;
        p.deps.push_back(d);
        done = true;
      }
    }

    if (!done) {
```

```
      string::size_type overlord = 0;
      const vector<dep_type> &h = p.children[p.head].deps;
      while (h[overlord].overlord != string::npos) {
        ++overlord;
      }
      for (unsigned i = 0; i < p.head; ++i) {
        overlord += p.children[i].deps.size();
      }
      unsigned shift = 0;
      for (unsigned i = 0; i < nkids; ++i) {
        const vector<dep_type> &q = p.children[i].deps;
        const unsigned qlen = q.size();
        for (unsigned j = 0; j < qlen; ++j) {
          dep_type d = q[j];
          if (d.overlord != string::npos) {
            d.overlord += shift;
          } else if (i != p.head) {
            d.overlord = overlord;
          }
          p.deps.push_back(d);
        }
        shift += qlen;
      }
    }
  }

  return p;
}
static inline void make_sentence(const parse &p, vector<string> *const v) {
  const unsigned kids = p.children.size();
  if (kids > 0) {
    for (unsigned i = 0; i < kids; ++i) {
      const parse &kid = p.children[i];
      if (kid.children.size() == 0) {
        v->push_back(p.name);
      } else {
        make_sentence(kid, v);
      }
    }
  }
}
static inline vector<string> make_sentence(const parse &p) {
  vector<string> v;
  make_sentence(p, &v);
  return v;
}

typedef struct {
  string::size_type a, k;
  double p;
} score_type;
static inline void maxim(const score_type &src,
                         score_type *const dst) {
  if (src.p > dst->p) {
    *dst = src;
  }
}
typedef struct {
  unsigned root, i, j, t;
} task_type;
static inline void pcfg(
    vector<dep_type> &deps,
    const double *const p_order,
    const stop_type *const p_stop,
    const attach_type *const p_attach) {
  const unsigned len = deps.size();
  assert(len <= MAX_LEN);
  assert(len > 0);

  // The chart for computing the best (P)CFG parses:
  static score_type (*const mem)[N_TYPES][MAX_LEN][MAX_LEN+1]
      = new score_type[MAX_LEN][N_TYPES][MAX_LEN][MAX_LEN+1];
  // For every head word h, and for its every possible
  // annotation (L,R,LR,RL,S), associates a crumb
  // leading to the best parse of i...j.

  score_type s;
  memset(&s, 0, sizeof(score_type));
  for (unsigned i = 0; i < len; ++i) {
    for (unsigned j = i + 1; j <= len; ++j) {
      for (unsigned w = i; w < j; ++w) {
        for (unsigned t = 0; t < N_TYPES; ++t) {
          mem[w][t][i][j] = s;
        }
      }
    }
  }

  // Base Case: initialize the terminal directions...
  for (unsigned w = 0; w < len; ++w) {
    const unsigned wc = deps[w].class_id;
```

```
  const double p = Porder(wc, p_order);
  mem[w][L][w][w+1].p = p;
  mem[w][R][w][w+1].p = 1 - p;
}

// Inductive Step: build up bigger parses...
for (unsigned d = 1; d <= len; ++d) {
  for (unsigned i = 0; i < len; ++i) {
    const unsigned j = i + d;
    if (j > len) {
      break;
    }

    for (unsigned h = i; h < j; ++h) {
      const unsigned hc = deps[h].class_id;
      if (d > 1) {  // propagate the binaries...
        for (unsigned t = 0; t < N_TYPES; ++t) {
          if (t == L || t == LR) {  // left
            for (s.k = i + 1; s.k <= h; ++s.k) {
              const double ns = 1 - Pstop(hc, L, s.k, h, p_stop);
              for (s.a = i; s.a < s.k; ++s.a) {
                const unsigned ac = deps[s.a].class_id;
                s.p = ns * Pattach(ac, hc, s.a, h, p_attach)
                    * mem[s.a][S][i][s.k].p * mem[h][t][s.k][j].p;
                maxim(s, &mem[h][t][i][j]);
              }
            }
          } else if (t == R || t == RL) {  // right
            for (s.k = h + 1; s.k <= j; ++s.k) {
              const double ns = 1 - Pstop(hc, R, s.k, h, p_stop);
              for (s.a = s.k; s.a < j; ++s.a) {
                const unsigned ac = deps[s.a].class_id;
                s.p = ns * Pattach(ac, hc, s.a, h, p_attach)
                    * mem[s.a][S][s.k][j].p * mem[h][t][i][s.k].p;
                maxim(s, &mem[h][t][i][j]);
              }
            }
          } else {  // sealed
          }
        }
      }

      {  // ... and don't forget the unaries!
        s.a = h;

        const double
            sl = Pstop(hc, L, i, h, p_stop),
            sr = Pstop(hc, R, j, h, p_stop);

        s.p = mem[h][L][i][j].p * sl;
        maxim(s, &mem[h][RL][i][j]);

        s.p = mem[h][R][i][j].p * sr;
        maxim(s, &mem[h][LR][i][j]);

        s.k = RL;  // slight abuse of notation...
        s.p = sr * mem[h][RL][i][j].p;
        maxim(s, &mem[h][S][i][j]);

        s.k = LR;  // ... and again
        s.p = sl * mem[h][LR][i][j].p;
        maxim(s, &mem[h][S][i][j]);
      }

      for (unsigned t = 0; t < N_TYPES; ++t) {
        const score_type &s = mem[h][t][i][j];
      }
    }
  }
}

vector<task_type> tasks; {
  task_type task;
  task.i = 0;
  task.j = len;
  task.t = S;
  double best_p = 0;
  for (unsigned h = 0; h < len; ++h) {
    const unsigned hc = deps[h].class_id;
    const double p = mem[h][S][0][len].p * Pattach(hc, 0, h, len, p_attach);
    if (p > best_p) {
      best_p = p;
      task.root = h;
    }
  }
  assert(best_p > 0);
  tasks.push_back(task);
  deps[task.root].overlord = len;
}
for (unsigned q = 0; q < tasks.size(); ++q) {
  task_type &task = tasks[q];
```

```
    const score_type &s = mem[task.root][task.t][task.i][task.j];
    if (task.j > task.i + 1) {
      if (task.root == s.a) {
        switch(task.t) {
          case S:
            task.t = s.k;
            break;
          case LR:
            task.t = R;
            break;
          case RL:
            task.t = L;
            break;
          default:
            assert(false);
        }
      } else {
        task_type a;
        deps[a.root = s.a].overlord = task.root;
        a.t = S;
        switch(task.t) {
          case L:
          case LR:
            a.i = task.i;
            a.j = task.i = s.k;
            break;
          case R:
          case RL:
            a.j = task.j;
            a.i = task.j = s.k;
            break;
          default:
            assert(false);
        }
        tasks.push_back(a);
      }
      --q;
    }
  }
}
static inline void example(const string &attach,
                           const unsigned apos,
                           const string &head,
                           const unsigned hpos,
                           hm_su_type *const hm) {
  const string s = (apos < hpos
                    ? attach + " <-- " + head
                    : head + " --> " + attach);
  const hm_su_type::iterator q = hm->find(s);
  if (q == hm->end()) {
    hm->insert(hm_su_type::value_type(s, 1));
  } else {
    ++q->second;
  }
}
static inline void appraise(
    const vector<vector<dep_type> > references,
    const double *const p_order,
    const stop_type *const p_stop,
    const attach_type *const p_attach,
    const bool verbose) {
  const unsigned n = references.size();
  double tdir = 0, tundir = 0;

  hm_su_type claimed, actual, swapped;
  for (unsigned i = 0; i < n; ++i) {
    if (verbose) {
      cout << endl << "Reference: " << (i+1) << endl;
    }

    const vector<dep_type> &deps = references[i];
    const unsigned int len = deps.size();
    vector<dep_type> copy;

    for (unsigned j = 0; j < len; ++j) {
      dep_type g = deps[j];
      g.overlord = string::npos;  // just to be sure...
      copy.push_back(g);
    }
    pcfg(copy, p_order, p_stop, p_attach);

    unsigned dir = 0, undir = 0;
    for (unsigned j = 0; j < len; ++j) {
      const dep_type &d = deps[j], &g = copy[j];
      char hit = ' ';
      if (d.overlord == g.overlord) {
        hit = 'D';
        ++undir;
        ++dir;
      } else if (g.overlord != len && (deps[g.overlord].overlord == j)) {
        hit = 'U';
```

```
        ++undir;
      }
      if (verbose) {
        if (hit == ' ') {
          example(d.word_class, j,
                  (d.overlord == len ? TERMINATOR : deps[d.overlord].word_class),
                  d.overlord, &actual);
          example(d.word_class, j,
                  (g.overlord == len ? TERMINATOR : deps[g.overlord].word_class),
                  g.overlord, &claimed);
        } else if (hit == 'U') {
          example(d.word_class, j,
                  (g.overlord == len ? TERMINATOR : deps[g.overlord].word_class),
                  g.overlord, &swapped);
        }
        cout << setw(5) << j << setw(5) << d.overlord
             << setw(5) << g.overlord << "  " << hit
             << "\t" << d.word << "[" << d.word_class << "]" << endl;
      }
    }
    tdir += static_cast<double>(dir) / len;
    tundir += static_cast<double>(undir) / len;
  }
  if (!verbose) {
    cout << "Undirected: " << setw(8) << setprecision(5) << (100 * tundir / n) << '%' << endl
         << "  Directed: " << setw(8) << setprecision(5) << (100 * tdir / n) << '%' << endl;
  } else {
    const hm_su_type *const anal[] = { &claimed, &actual, &swapped };
    for (unsigned i = 0; i < 3; ++i) {
      const hm_su_type &hm = *(anal[i]);
      cout << endl << (i == 0 ? "Claimed" :
                       (i == 1 ? "Actual" : "Swapped"))
           << " [" << hm.size() << "]" << endl;

      typedef vector<pair<unsigned, string> > o_type;
      o_type offending; {
        const hm_su_type::const_iterator end = hm.end();
        for (hm_su_type::const_iterator j = hm.begin(); j != end; ++j) {
          offending.push_back(pair<unsigned, string>(-j->second, j->first));
        }
      }
      sort(offending.begin(), offending.end());
      const o_type::const_iterator end = offending.end();
      for (o_type::const_iterator j = offending.begin(); j != end; ++j) {
        cout << "\t" << j->second << " [" << -j->first << "]" << endl;
      }
    }
  }
}

int main() {
  words.push_back(TERMINATOR);
  word_ids[TERMINATOR] = 0;

  unsigned max_len = 0, skipped = 0, kept = 0;
  vector<vector<dep_type> > references;
  vector<vector<unsigned> > sentences; {
    string item, tree;
    unsigned level = 0;
    while (cin >> item) {
      string::size_type pair;
      while ((pair = item.find("((")) != string::npos) {
        item.replace(pair, 1, "( ", 2);
      }
      tree += " " + item;

      const string::size_type len = item.size();
      for (string::size_type pos = 0; pos < len; ++pos) {
        switch(item[pos]) {
          case '(':
            ++level;
            break;
          case ')':
            --level;
          default:
            ;
        }
      }

      if (level == 0) {
        const char *str = tree.c_str();
        const parse p = make_parse(str);
        const vector<string> sentence = make_sentence(p);
        const unsigned len = sentence.size();
        if (len > max_len) {
          max_len = len;
        }
        if (len >= MAX_LEN) {
          ++skipped;
        } else {
          ++kept;
```

```
        if (p.dep_ok) {
          vector<unsigned> v;
          for (unsigned i = 0; i < len; ++i) {
            v.push_back(word_id(sentence[i]));
          }
          v.push_back(0);
          sentences.push_back(v);
          references.push_back(p.deps);
        }
      }
      tree.clear();
    }
  }
}
const unsigned
    nwords = words.size(),
    nsentences = sentences.size(),
    nreferences = references.size();

cout << endl << "Parsed " << (kept + skipped)
     << " sentences (maximum length " << max_len << ");" << endl << "Skipped "
     << skipped << " sentences (those exceeding length " << (MAX_LEN-1) << ")." << endl
     << endl << "Training on the remaining " << kept << " sentences..."
     << endl << "Identified " << nwords
     << " unique word classes (including the diamond)." << endl;

cout << endl << "Percolated various head rules to make "
     << nreferences << " reference" << endl << "dependency parses";
if (!unheaded.empty()) {
  cout << "; offending productions follow (sorted by frequency):" << endl << endl;
  typedef vector<pair<unsigned, string> > o_type;
  o_type offending; {
    const hm_su_type::const_iterator end = unheaded.end();
    for (hm_su_type::const_iterator i = unheaded.begin(); i != end; ++i) {
      offending.push_back(pair<unsigned, string>(-i->second, i->first));
    }
  }
  sort(offending.begin(), offending.end());
  const o_type::const_iterator end = offending.end();
  for (o_type::const_iterator i = offending.begin(); i != end; ++i) {
    cout << "\t" << i->second << " [" << -i->first << "]" << endl;
  }
} else {
  cout << "." << endl;
}

if (nsentences == 0 || nreferences == 0) {
  cout << endl << "Not much can be done here!!" << endl;
  exit(1);
}
for (unsigned i = 0; i < nreferences; ++i) {
  vector<dep_type> &deps = references[i];
  const unsigned len = deps.size();
  for (unsigned j = 0; j < len; ++j) {
    dep_type &d = deps[j];
    if (d.overlord == string::npos) {
      d.overlord = len;
    }
    d.class_id = word_id(d.word_class);
  }
}


double p_order[nwords], l_order[nwords], r_order[nwords], n_order[nwords];
p_order[0] = 0;
for (unsigned i = 1; i < nwords; ++i) {
  p_order[i] = 0.5;
}

stop_type p_stop[nwords], c_stop[nwords], t_stop[nwords], n_stop[nwords], z_stop[nwords];
for (unsigned i = 1; i < nwords; ++i) {
  p_stop[i].x[L][false] = p_stop[i].x[L][true] =
      p_stop[i].x[R][false] = p_stop[i].x[R][true] = 0.5;
}
p_stop[0].x[R][false] = p_stop[0].x[R][true] =
    p_stop[0].x[L][false] = 1;
p_stop[0].x[L][true] = 0;

attach_type p_attach[nwords], c_attach[nwords], n_attach[nwords], z_attach[nwords];
double t_attach[nwords][N_DIRS];
for (unsigned i = 0; i < nwords; ++i) {
  for (unsigned s = 0; s < N_DIRS; ++s) {
    p_attach[i].x[s] = (double *)calloc(nwords, sizeof(double));
    c_attach[i].x[s] = (double *)calloc(nwords, sizeof(double));
    n_attach[i].x[s] = (double *)calloc(nwords, sizeof(double));
    z_attach[i].x[s] = (double *)calloc(nwords, sizeof(double));
  }
  t_attach[i][L] = t_attach[i][R] = 0;
  z_stop[i].x[L][false] = z_stop[i].x[L][true] =
      z_stop[i].x[R][false] = z_stop[i].x[R][true] = 0;
  t_stop[i].x[L][false] = t_stop[i].x[L][true] =
```

```
      t_stop[i].x[R][false] = t_stop[i].x[R][true] = 0;
}
for (unsigned i = 1; i < nwords; ++i) {
  p_attach[i].x[L][0] = 1.0 / (nwords - 1);
}
for (unsigned i = 1; i < nwords; ++i) {
  for (unsigned j = 1; j < nwords; ++j) {
    p_attach[i].x[L][j] = 1.0 / (nwords - 1);
    p_attach[i].x[R][j] = 1.0 / (nwords - 1);
  }
}

cout << endl << "Zero knowledge:" << endl;
appraise(references, p_order, p_stop, p_attach, VERBOSE);

cout << endl << "Perfect knowledge:" << endl;
for (unsigned i = 0; i < nreferences; ++i) {
  const vector<dep_type> &r = references[i];
  const unsigned len = r.size();
  for (unsigned j = 0; j < len; ++j) {
    const dep_type &d = r[j];
    const unsigned
        ac = d.class_id,
        hc = (d.overlord == len ? 0 : r[d.overlord].class_id);
    const unsigned dir = (j < d.overlord ? L : R);
    ++z_attach[ac].x[dir][hc];
    ++t_attach[hc][dir];
    {
      unsigned ls = 0;
      for (unsigned k = 0; k < j; ++k) {
        if (r[k].overlord == j) {
          ++ls;
        }
      }
      if (ls == 0) {
        ++z_stop[ac].x[L][true];
      } else {
        z_stop[ac].x[L][false] += ls;
        ++t_stop[ac].x[L][false];
      }
      ++t_stop[ac].x[L][true];
    }
    {
      unsigned rs = 0;
      for (unsigned k = j + 1; k < len; ++k) {
        if (r[k].overlord == j) {
          ++rs;
        }
      }
      if (rs == 0) {
        ++z_stop[ac].x[R][true];
      } else {
        z_stop[ac].x[R][false] += rs;
        ++t_stop[ac].x[R][false];
      }
      ++t_stop[ac].x[R][true];
    }
  }
}
for (unsigned hc = 0; hc < nwords; ++hc) {
  for (unsigned ac = 1; ac < nwords; ++ac) {
    if (t_attach[hc][L] > 0) {
      z_attach[ac].x[L][hc] /= t_attach[hc][L];
      p_attach[ac].x[L][hc] = z_attach[ac].x[L][hc];
    }
    if (t_attach[hc][R] > 0) {
      z_attach[ac].x[R][hc] /= t_attach[hc][R];
      p_attach[ac].x[R][hc] = z_attach[ac].x[R][hc];
    }
  }
  for (unsigned d = 0; d < N_DIRS; ++d) {
    if (hc == 0) {
      z_stop[hc].x[d][true] = p_stop[hc].x[d][true];
      z_stop[hc].x[d][false] = p_stop[hc].x[d][false];
    } else {
      if (t_stop[hc].x[d][true] > 0) {
        z_stop[hc].x[d][true] /= t_stop[hc].x[d][true];
      }
      if (t_stop[hc].x[d][false] > 0) {  // Geometric distribution...
        z_stop[hc].x[d][false] = t_stop[hc].x[d][false]
                               / z_stop[hc].x[d][false];
      }
    }
  }
}
appraise(references, p_order, z_stop, z_attach, VERBOSE);

harmonic = true;
cout << endl << "Ad-hoc harmonic: [" << harmonic_c << "]" << endl;
appraise(references, p_order, p_stop, p_attach, VERBOSE);
```

```
double delta = 1;
unsigned iter = 0;
while (delta > MIN_EM_ERR && iter < MAX_EM_ITER) {
  cout << endl << endl << "EM Iteration: " << ++iter << endl;

  delta = 0;
  for (unsigned w = 0; w < nwords; ++w) {
    n_order[w] = 0;  // Porder

    for (unsigned d = 0; d < N_DIRS; ++d) {  // Pstop
      for (unsigned a = 0; a < 2; ++a) {
        n_stop[w].x[d][a] = 0;
      }
    }

    for (unsigned a = 0; a < nwords; ++a) {
      n_attach[a].x[L][w] = n_attach[a].x[R][w] = 0;
    }
  }

  for (unsigned j = 0; j < nsentences; ++j) {
    process(sentences[j],
            p_order, l_order, r_order,
            p_stop, c_stop, t_stop,
            p_attach, c_attach, t_attach,
            nwords);

    for (unsigned w = 0; w < nwords; ++w) {
      {  // Porder
        const double l = l_order[w], r = r_order[w], sum = l + r;
        if (sum > 0) {
          n_order[w] += l / sum;
        }
      }

      {  // Pstop
        for (unsigned d = 0; d < N_DIRS; ++d) {
          for (unsigned a = 0; a < 2; ++a) {
            const double t = t_stop[w].x[d][a];
            if (t > 0) {
              const double c = c_stop[w].x[d][a];
              n_stop[w].x[d][a] += c / t;
            }
          }
        }
      }

      {  // Pattach
        const double tl = t_attach[w][L], tr = t_attach[w][R];
        if (tl > 0) {
          for (unsigned a = 0; a < nwords; ++a) {
            n_attach[a].x[L][w] += c_attach[a].x[L][w] / tl;
          }
        }
        if (tr > 0) {
          for (unsigned a = 0; a < nwords; ++a) {
            n_attach[a].x[R][w] += c_attach[a].x[R][w] / tr;
          }
        }
      }
    }
  }

  double lossStopAdj = 0, lossStopNon = 0, lossAttach = 0;
  unsigned lossStopAdjN = 0, lossStopNonN = 0, lossAttachN = 0;
  for (unsigned w = 0; w < nwords; ++w) {
    {  // Porder
      const double before = p_order[w];
      if (w > 0) {
        const double
            after = n_order[w] / nsentences,
            diff = fabs(before - after);
        if (diff > delta) {
          delta = diff;
          cout << "\tPorder[" << words[w] << "][L]: "
               << diff << " (" << before << " --> " << after << ")" << endl;
        }
        p_order[w] = after;
      } else {
        p_order[w] = before;
      }
    }

    {  // Pstop
      for (unsigned d = 0; d < N_DIRS; ++d) {
        for (unsigned a = 0; a < 2; ++a) {
          const double before = p_stop[w].x[d][a];
          if (w > 0) {
            const double
                after = n_stop[w].x[d][a] / nsentences,
                diff = fabs(before - after);
```

```
            if (diff > delta) {
              delta = diff;
              cout << "\tPstop[" << words[w] << "]["
                   << (d == L ? 'L' : 'R') << "]["
                   << (a ? "adj" : "not") << "]: "
                   << diff << " (" << before << " --> " << after << ") vs. "
                   << z_stop[w].x[d][a] << endl;
            }
            p_stop[w].x[d][a] = after;
          } else {
            p_stop[w].x[d][a] = before;
          }
          if (a == 0) {
            lossStopNon += fabs(p_stop[w].x[d][a] - z_stop[w].x[d][a]);
            ++lossStopNonN;
          } else {
            lossStopAdj += fabs(p_stop[w].x[d][a] - z_stop[w].x[d][a]);
            ++lossStopAdjN;
          }
        }
      }
    }

    {  // Pattach
      for (unsigned a = 0; a < nwords; ++a) {
        for (unsigned d = 0; d < N_DIRS; ++d) {
          const double
              before = p_attach[a].x[d][w],
              after = n_attach[a].x[d][w] / nsentences,
              diff = fabs(before - after);
          if (diff > delta) {
            delta = diff;
            cout << "\tPattach[" << words[a] << "]["
                 << words[w] << "][" << (d == L ? 'L' : 'R') << "]: "
                 << diff << " (" << before << " --> " << after << ") vs. "
                 << z_attach[a].x[d][w] << endl;
          }
          p_attach[a].x[d][w] = after;
          lossAttach += fabs(after - z_attach[a].x[d][w]);
          ++lossAttachN;
        }
      }
    }
  }
  cout << "OracleLoss: " << setw(8) << setprecision(5)
       << (-100 * lossStopAdj / lossStopAdjN) << "% (PstopAdj)" << endl
       << "OracleLoss: " << setw(8) << setprecision(5)
       << (-100 * lossStopNon / lossStopNonN) << "% (PstopNon)" << endl
       << "OracleLoss: " << setw(8) << setprecision(5)
       << (-100 * lossAttach / lossAttachN) << "% (Pattach)" << endl;

  harmonic = false;
  appraise(references, p_order, p_stop, p_attach, false);
 }

 if (VERBOSE) {
  appraise(references, p_order, p_stop, p_attach, true);
 }
}
```