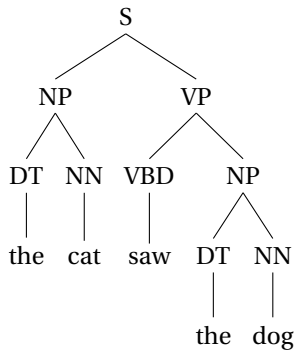
Chapter 14

(Partially) Unsupervised Parsing

The linguistically-motivated tree transformations we discussed previously are very effective, but when we move to a new language, we may have to come up with new ones. It would be nice if we could automatically discover these transformations. Suppose that we have a grammar defined over nonterminals of the form $X[q]$, where X is a nonterminal from the training data (e.g., NP) and q is a number between 1 and k (for simplicity, let's say $k = 2$). We only observe trees over nonterminals X, but need to learn weights for our grammar. We can do this using EM Matsuzaki, Miyao, and Tsujii, 2005; Petrov et al., 2006.

14.1 Hard EM on trees

Suppose our first training example is the following tree:



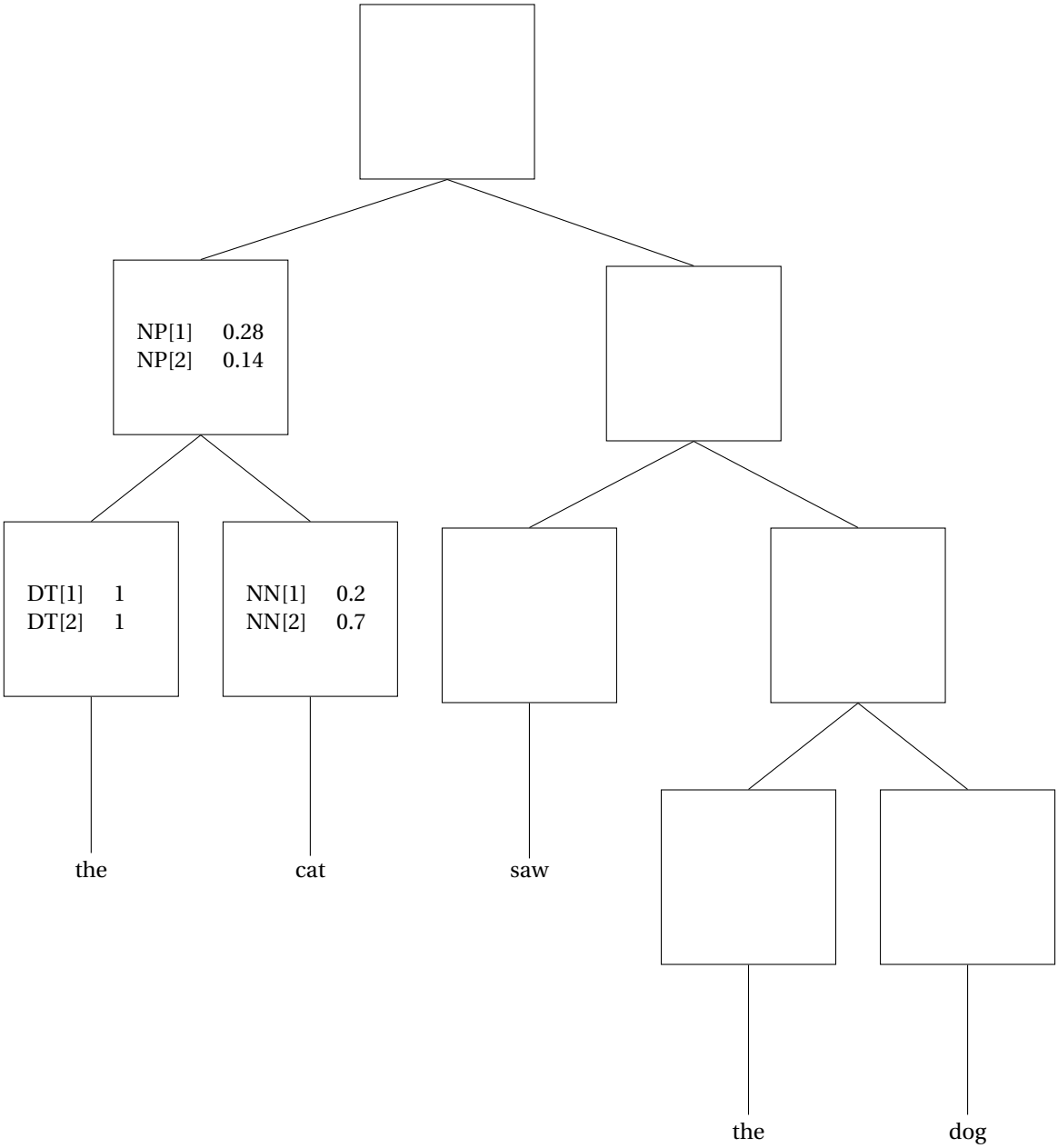
And suppose that our initial grammar is:

$\text{DT}[1] \xrightarrow{1} \text{the}$	$\text{S}[1] \xrightarrow{0.2} \text{NP}[1]\ \text{VP}[1]$	$\text{NP}[1] \xrightarrow{0.2} \text{DT}[1]\ \text{NN}[1]$	$\text{VP}[1] \xrightarrow{0.2} \text{VBD}[1]\ \text{NP}[1]$
$\text{DT}[2] \xrightarrow{1} \text{the}$	$\text{S}[1] \xrightarrow{0.4} \text{NP}[1]\ \text{VP}[2]$	$\text{NP}[1] \xrightarrow{0.4} \text{DT}[1]\ \text{NN}[2]$	$\text{VP}[1] \xrightarrow{0.4} \text{VBD}[1]\ \text{NP}[2]$
$\text{NN}[1] \xrightarrow{0.2} \text{cat}$	$\text{S}[1] \xrightarrow{0.1} \text{NP}[2]\ \text{VP}[1]$	$\text{NP}[1] \xrightarrow{0.1} \text{DT}[2]\ \text{NN}[1]$	$\text{VP}[1] \xrightarrow{0.1} \text{VBD}[2]\ \text{NP}[1]$
$\text{NN}[1] \xrightarrow{0.8} \text{dog}$	$\text{S}[1] \xrightarrow{0.3} \text{NP}[2]\ \text{VP}[2]$	$\text{NP}[1] \xrightarrow{0.3} \text{DT}[2]\ \text{NN}[2]$	$\text{VP}[1] \xrightarrow{0.3} \text{VBD}[2]\ \text{NP}[2]$
$\text{NN}[2] \xrightarrow{0.7} \text{cat}$	$\text{S}[2] \xrightarrow{0.5} \text{NP}[1]\ \text{VP}[1]$	$\text{NP}[2] \xrightarrow{0.5} \text{DT}[1]\ \text{NN}[1]$	$\text{VP}[2] \xrightarrow{0.5} \text{VBD}[1]\ \text{NP}[1]$
$\text{NN}[2] \xrightarrow{0.3} \text{dog}$	$\text{S}[2] \xrightarrow{0.1} \text{NP}[1]\ \text{VP}[2]$	$\text{NP}[2] \xrightarrow{0.1} \text{DT}[1]\ \text{NN}[2]$	$\text{VP}[2] \xrightarrow{0.1} \text{VBD}[1]\ \text{NP}[2]$
$\text{VBD}[1] \xrightarrow{1} \text{saw}$	$\text{S}[2] \xrightarrow{0.2} \text{NP}[2]\ \text{VP}[1]$	$\text{NP}[2] \xrightarrow{0.2} \text{DT}[2]\ \text{NN}[1]$	$\text{VP}[2] \xrightarrow{0.2} \text{VBD}[2]\ \text{NP}[1]$
$\text{VBD}[2] \xrightarrow{1} \text{saw}$	$\text{S}[2] \xrightarrow{0.2} \text{NP}[2]\ \text{VP}[2]$	$\text{NP}[2] \xrightarrow{0.2} \text{DT}[2]\ \text{NN}[2]$	$\text{VP}[2] \xrightarrow{0.2} \text{VBD}[2]\ \text{NP}[2]$

If we want to do *hard* EM, we need to do:

- E step: find the highest-weight derivation of the grammar that matches the observed tree (modulo the annotations $[q]$).
- M step: re-estimate the weights of the grammar by counting the rules used in the derivations found in the E step, and normalize.

The M step is easy. The E step is essentially Viterbi CKY, only easier because we're given a tree instead of just a string. The chart for this algorithm looks like the following, where each cell works exactly like the cells in CKY. Can you fill in the rest?



14.2 Hard EM on strings

Another scenario is that we're given only strings instead of trees, and we have some grammar that we want to learn weights for. For example, if we train a grammar on the Wall Street Journal portion of the Penn Treebank, but we want to learn a parser for Twitter. This kind of *domain adaptation* problem is

often solved using something similar to hard EM.

- Extract the grammar rules and initialize the weights by training on the labeled training data (in our example, the WSJ portion of the Treebank).
- E step: find the highest-weight tree of the grammar for each string in the unlabeled training data (in our example, the Twitter data).
- M step: re-estimate the weights of the grammar.

The E/M steps can be repeated, though in practice one might find that one iteration works the best.

The following sections, all optional, present the machinery that we need in order to do real EM on either strings or trees. First, we present a more abstract view of parsing as intersection between a CFG and a finite-state automaton (analogous to how we viewed decoding with FSTs as intersection). Then, we show how to compute expected counts of rules using the Inside/Outside algorithm, which is analogous to computing expected counts of transitions using the Forward/Backward algorithm.

14.3 Parsing as intersection (optional)

Previously we showed how to find the best tree for w. But suppose we don't just want the best tree, but all possible trees. This is, again, an easy modification. Instead of keeping the best back-pointer in each $back[i, j][X]$, we keep a set of all the back-pointers.

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Ensure: G' generates all parses of w

initialize $chart[i, j][X] \leftarrow \emptyset$ **for all** $0 \le i < j \le n, X \in N$
for all $i \leftarrow 1, \ldots, n$ **and** $(X \xrightarrow{p} w_i) \in R$ **do**
    $chart[i[1], i][X] \leftarrow chart[i-1, i][X] \cup \{X_{i-1,i} \xrightarrow{p} w_i\}$
end for
for $\ell \leftarrow 2, \ldots, n$ **do**
    for $i \leftarrow 0, \ldots, n - \ell$ **do**
        $j \leftarrow i + \ell$
        for $k \leftarrow i+1, \ldots, j-1$ **do**
            for all $(X \xrightarrow{p} Y Z) \in R$ **do**
                if $chart[i, k][Y] \neq \emptyset$ **and** $chart[k, j][Z] \neq \emptyset$ **then**
                    $chart[i, j][X] \leftarrow chart[i, j][X] \cup \{X_{i,j} \xrightarrow{p} Y_{i,k} Z_{k,j}\}$
                end if
            end for
        end for
    end for
end for
$G' \leftarrow \bigcup_{0 \le i < j \le n} \bigcup_{X \in N} chart[i, j][X]$

The grammar G' generates all and only the parse trees of w. It is usually called a *packed forest*. It is a forest because it represents a set of trees, and it is packed because it is only polynomial sized yet represents a possibly exponential set of trees. (It is more commonly represented as a data structure called a

hypergraph rather than as a CFG. But since the two representations are equivalent, we will stick with a CFG.)

The packed forest can be thought of as a CFG that generates the language $L(G) \cap \{w\}$. More generally, a CFG can be intersected with any finite-state automaton to produce another CFG. Why would we want to do this? For example, consider the following fragment G of our grammar (**??**) from above (NP is the start symbol).

$$\text{NP} \rightarrow \text{DT NN} \tag{14.1}$$
$$\text{DT} \rightarrow \text{a} \tag{14.2}$$
$$\text{DT} \rightarrow \text{an} \tag{14.3}$$
$$\text{NN} \rightarrow \text{arrow} \tag{14.4}$$
$$\text{NN} \rightarrow \text{banana} \tag{14.5}$$

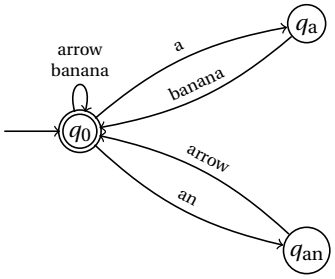
This generates the ungrammatical strings

(14.6) a arrow

(14.7) an banana

How do we tell a computer when to say *a* and when to say *an*? The most natural way is to use an FSA like the following FSA M:

(14.8)



So we need to combine G and M into a single thing (another CFG) that generates the intersection of $L(G)$ and $L(M)$.

To do this, we can use a construction due to Bar-Hillel (Bar-Hillel, Perles, and Shamir, 1961), we which illustrate by example. Assume that M has no ϵ-transitions. The basic idea is to modify the CFG so that it simulates the action of M. Every nonterminal symbol A gets annotated with two states, becoming $A_{q,r}$. State q says where the FSA could be before reading the yield of the symbol, and r says where the FSA would end up after reading the yield of the symbol.

We consider each rule of the grammar one by one. First, consider the production DT → a. Suppose that M is in state q_0. After reading the yield of this rule, it will end up in state q_{a}. So we make a new rule,

$$\text{DT}_{q_0,q_{\text{a}}} \rightarrow \text{a}$$

But what if M starts in state q_{a} or state q_{an}? In that case, M would reject the string, so we don't make any new rules for these cases. By similar reasoning, we make rules

$$\begin{aligned}
\text{DT}_{q_0,q_{\text{an}}} &\rightarrow \text{an} \\
\text{NN}_{q_0,q_0} &\rightarrow \text{arrow} \\
\text{NN}_{q_{\text{an}},q_0} &\rightarrow \text{arrow} \\
\text{NN}_{q_0,q_0} &\rightarrow \text{banana} \\
\text{NN}_{q_{\text{a}},q_0} &\rightarrow \text{banana}
\end{aligned}$$

Now consider the production NP → DT NN. Imagine that M is in state q_0 and reads the yield of DT. What state will it be in after? Since we don't know, we consider all possibilities. Suppose that it moves to state q_{a}, and suppose that after reading the yield of NN, it ends up in state q_{an}. To capture this case, we make a new rule,

$$\text{NP}_{q_0,q_{\text{an}}} \rightarrow \text{DT}_{q_0,q_{\text{a}}}\ \text{NN}_{q_{\text{a}},q_{\text{an}}}$$

We do this for every possible sequence of states. (We didn't promise that this construction would be the most efficient one!)

$$\begin{aligned}
\text{NP}_{q_0,q_0} &\rightarrow \text{DT}_{q_0,q_0}\ \text{NN}_{q_0,q_0} \\
\text{NP}_{q_0,q_{\text{a}}} &\rightarrow \text{DT}_{q_0,q_0}\ \text{NN}_{q_0,q_{\text{a}}} \\
\text{NP}_{q_0,q_{\text{an}}} &\rightarrow \text{DT}_{q_0,q_0}\ \text{NN}_{q_0,q_{\text{an}}} \\
\text{NP}_{q_0,q_0} &\rightarrow \text{DT}_{q_0,q_{\text{a}}}\ \text{NN}_{q_{\text{a}},q_0} \\
&\vdots
\end{aligned}$$

Finally, we create a new start symbol, S', and a rule

$$S' \rightarrow \text{NP}_{q_0,q_0}$$

because q_0 is both the initial state and a final state. If M had more than one final state, we would make a new rule for each.

Question 24. How big is the resulting grammar, as a function of the number of rules in G and states in M?

But not all the rules generated are actually used. For example, it's impossible to get from q_0 to q_0 while reading a DT, so any rule that involves DT_{q_0,q_0} can be deleted. We can build a more compact grammar as follows. Consider again the production NP → DT NN. Imagine that M is in state q_0 and reads the yield of DT. What state will it be in after? Instead of considering all possibilities, we only look at the annotated nonterminals that have been created already. We have created $DT_{q_0,q_{\text{a}}}$, so M could be in state q_{a}. Then, after reading NN, it could be in state q_0 (because we have created $\text{NN}_{q_{\text{a}},q_0}$). So we make a new rule,

$$\text{NP}_{q_0,q_0} \rightarrow \text{DT}_{q_0,q_{\text{a}}}\ \text{NN}_{q_{\text{a}},0}$$

and, by similar reasoning, we make

$$\text{NP}_{q_0,q_0} \rightarrow \text{DT}_{q_0,q_{\text{an}}}\ \text{NN}_{q_{\text{an}},0}$$

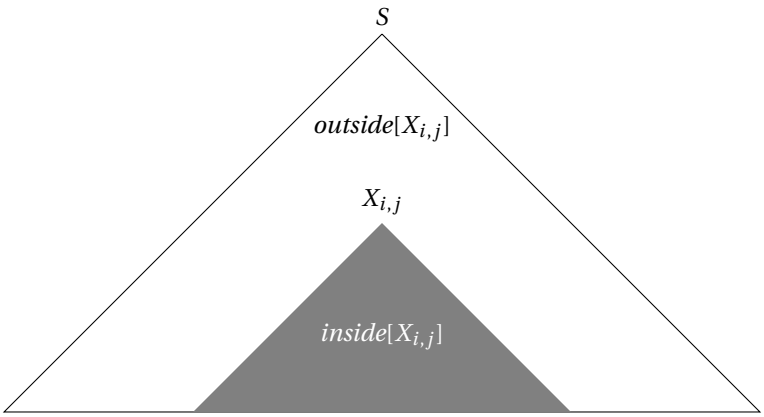


Figure 14.1: The inside probability of $X_{i,j}$ is the total weight of all subtrees rooted in $X_{i,j}$ (gray), and the outside probability is the total weight of all tree fragments with a "broken-off" node $X_{i,j}$ (white).

but no others. We do this for all the rules in the grammar. But after we are done, we must go back and process all the rules again, because the set of annotated nonterminals may have changed. Only when the set of annotated nonterminals converges can we stop. In this case, we are done.

The resulting grammar is:

$$\begin{aligned}
\mathrm{NP}_{q_0,q_0} &\rightarrow \mathrm{DT}_{q_0,q_\mathrm{a}}\ \mathrm{NN}_{q_\mathrm{a},q_0} \\
\mathrm{NP}_{q_0,q_0} &\rightarrow \mathrm{DT}_{q_0,q_\mathrm{an}}\ \mathrm{NN}_{q_\mathrm{an},q_0} \\
\mathrm{DT}_{q_0,q_\mathrm{a}} &\rightarrow \mathrm{a} \\
\mathrm{DT}_{q_0,q_\mathrm{an}} &\rightarrow \mathrm{an} \\
\mathrm{NN}_{q_0,q_0} &\rightarrow \mathrm{arrow} \\
\mathrm{NN}_{q_\mathrm{an},q_0} &\rightarrow \mathrm{arrow} \\
\mathrm{NN}_{q_0,q_0} &\rightarrow \mathrm{banana} \\
\mathrm{NN}_{q_\mathrm{a},q_0} &\rightarrow \mathrm{banana}
\end{aligned}$$

Question 25. Extend the FSA M to include all the vocabulary from grammar (**??**) and find their intersection.

14.4 Inside/outside probabilities (optional)

Recall that when training a weighted FSA using EM, the key algorithmic step was calculating forward and backward probabilities. The forward probability of a state q is the total weight of all paths from the start state to q, and the backward probability is the total weight of all paths from q to any final state. We can define a similar concept for a node (nonterminal symbol) in a forest.

The *inside probability* of a node $X_{i,j}$ is the total weight of all derivations $X_{i,j} \Rightarrow^* w$, where w is any string of terminal symbols. That is, it is the total weight of all subtrees derivable by $X_{i,j}$.

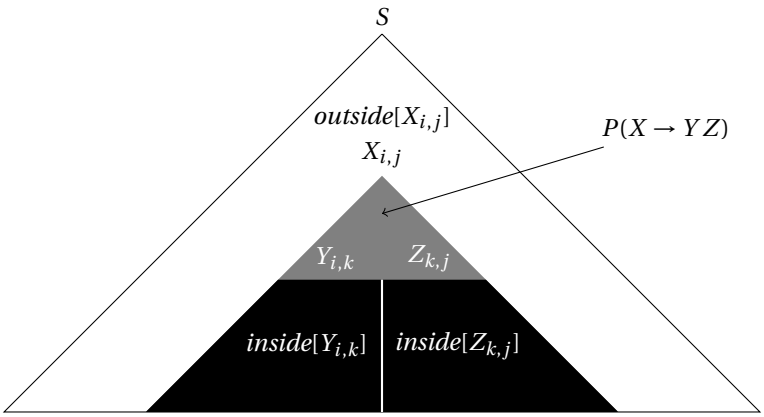


Figure 14.2: To compute the outside probability of $Y_{i,k}$, we use the outside probability of $X_{i,j}$ and the inside probability of $Z_{k,j}$.

The *outside probability* is the total weight of all derivations $S \Rightarrow^* v X_{i,j} w$, where v and w are strings of terminal symbols. That is, it is the total weight of all tree fragments with root S and a single "broken-off" node $X_{i,j}$ (see Figure **??**).

We already left it to you (Question **??**) to figure out how to calculate the total weight of all derivations. The intermediate values of this calculation are the inside probabilities. How about the outside probabilities? This computation proceeds top-down. To compute the outside probability of $Y_{i,k}$, we look at its possible parents $X_{i,j}$ and siblings $Z_{k,j}$. For each, we can compute the total weight of the outside part of all derivations going through these three nodes: it is $\mathit{outside}[X_{i,j}] \cdot P(X \to Y\,Z) \cdot \mathit{inside}[Z_{k,j}]$. If we sum over all possible parents and siblings (both left and right), we get the outside probability of $Y_{i,k}$.

More precisely, the algorithm looks like this:

Require: string $w = w_1 \cdots w_n$ and grammar $G = (N, \Sigma, R, S)$
Require: $\mathit{inside}[X_{i,j}]$ is the inside probability of $X_{i,j}$
Ensure: $\mathit{outside}[X_{i,j}]$ is the outside probability of $X_{i,j}$

```
for all X, i, j do
    initialize outside[X_{i,j}] ← 0
end for
outside[S, 0, n] ← 1
for ℓ ← n, n−1, ..., 2 do                                   ▷ top-down
    for i ← 0, ..., n − ℓ do
        j ← i + ℓ
        for k ← i + 1, ..., j − 1 do
            for all (X −p→ Y Z) ∈ R do
                outside[Y_{i,k}] ← outside[Y_{i,k}] + outside[X_{i,j}] · p · inside[Z_{k,j}]
                outside[Z_{k,j}] ← outside[Z_{k,j}] + outside[X_{i,j}] · p · inside[Y_{i,k}]
            end for
        end for
    end for
```

end for

Question 26. Fill in the rest of the details for EM training of a PCFG.

1. What is the total weight of derivations going through a forest edge $X_{i,j} \rightarrow Y_{i,k} Z_{k,j}$?

2. What is the fractional count of this edge (probability of using this edge given input string w)?

3. How do you compute the fractional count of a production $X \rightarrow Y Z$ (probability of using this production given input string w)?

4. How do you reestimate the probability of a production $X \rightarrow Y Z$?

Bibliography

Bar-Hillel, Y., M. Perles, and E. Shamir (1961). "On formal properties of simple phrase structure grammars". In: *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung* 14.2, pp. 143–172.

Matsuzaki, Takuya, Yusuke Miyao, and Jun'ichi Tsujii (2005). "Probabilistic CFG with Latent Annotations". In: *Proc. ACL*, pp. 75–82.

Petrov, Slav et al. (2006). "Learning Accurate, Compact, and Interpretable Tree Annotation". In: *Proc. COLING-ACL*, pp. 433–440.