Chapter 6

Statistical Parsing

Given a corpus of trees, it is easy to extract a CFG and estimate its parameters. Every tree can be thought of as a CFG derivation, and we just perform relative frequency estimation (count and divide) on them. That is, let $c(A \rightarrow \beta)$ be the number of times that the rule $A \rightarrow \beta$ was observed, and then

$$c(A) = \sum_{\beta} c(A \rightarrow \beta) \tag{6.1}$$

$$\hat{P}(A \rightarrow \beta \mid A) = \frac{c(A \rightarrow \beta)}{c(A)} \tag{6.2}$$

6.1 Parser evaluation

Evaluation of parsers almost always uses *labeled precision and recall* or the *labelled F1 score* Black et al., 1991. To define this metric, we make use of the notion of a *multiset*, which is a set where items can occur more than once. If A and B are multisets, define $A(x)$ to be the number of times that x occurs in A, and define

$$|A| = \sum_{x} A(x) \tag{6.3}$$

$$(A \cap B)(x) = \min\{A(x), B(x)\} \tag{6.4}$$

We view a tree as a multiset of brackets $[X, i, j]$ for each node of the tree, where X is the label of the node and $w_{i+1} \cdots w_j$ is its span. Note that in Penn Treebank style trees, every word is an only child and its parent is a part-of-speech tag. The part-of-speech tag nodes (also called *preterminal* nodes) are *not* included in the multiset.

Let t (for *test*) be the parser output and g (for *gold*) be the gold-standard tree that we are evaluating against. Then define the precision $p(t, g)$ and recall $g(t, g)$ to be:

$$p(t, g) = \frac{|t \cap g|}{|t|} \tag{6.5}$$

$$r(t, g) = \frac{|t \cap g|}{|g|} \tag{6.6}$$

and the F1 score to be their harmonic mean:

$$F_1(t,g) = \frac{1}{\frac{1}{2}\left(\frac{1}{p(t,g)} + \frac{1}{r(t,g)}\right)} \tag{6.7}$$

$$= \frac{2|t \cap g|}{|t| + |g|} \tag{6.8}$$

The typical setup for English parsing is to train the parser on the Penn Treebank, Wall Street Journal sections 02–21, to do development on section 00 or 22, and to test on section 23. If we train a PCFG without any modifications, we will get an F1 score of only 73%. State-of-the-art scores are above 90%.

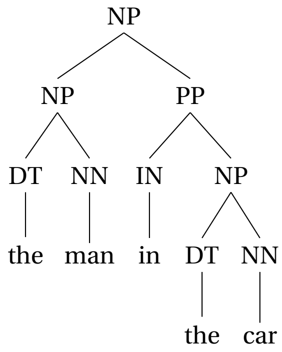
6.2 Markovization

A PCFG captures the dependency between a parent node and all of its children. On the Penn Treebank, this leads to over 10,000 rules, each with its own probability. In practice, it turns out that this tends to be both too little and too much.

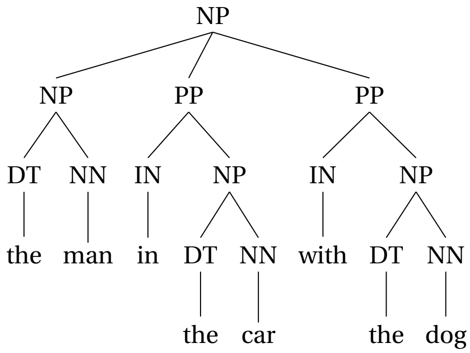
6.2.1 Vertical markovization

Too see why it can be too little, suppose our Treebank looked like this (Johnson, 1998; Klein and Manning, 2003):

90 times



10 times



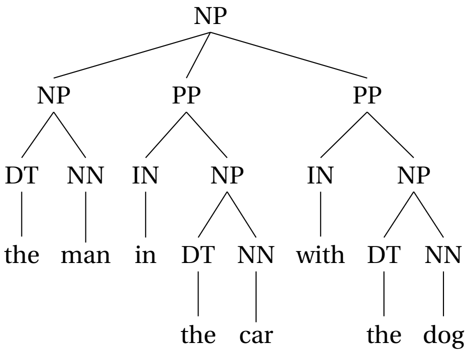
From this we would learn

$$\hat{P}(\text{NP} \rightarrow \text{NP PP}) = 90/310 \tag{6.9}$$

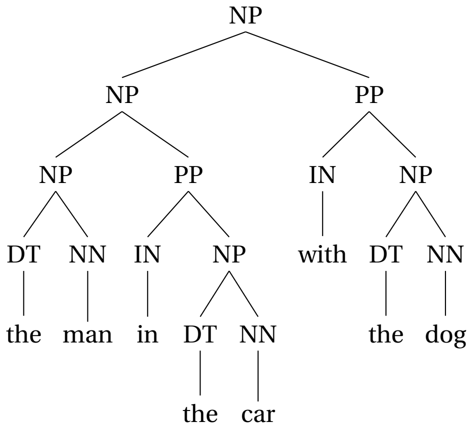
$$\hat{P}(\text{NP} \rightarrow \text{NP PP PP}) = 10/310 \tag{6.10}$$

and whenever the parser is asked to choose between these two trees:

(6.11)



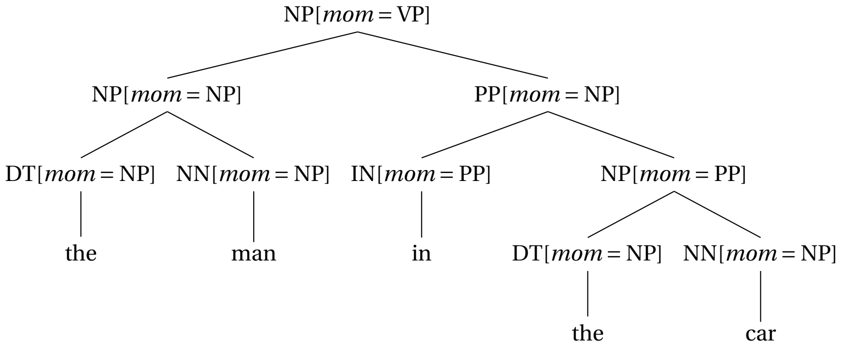
(6.12)



it will prefer the second one, which was never observed in the training data!

This can be corrected by modifying the node labels to increase their sensitivity to their vertical context, much in the same way that we can increase the context-sensitivity of an n-gram language model by increasing n. We simply annotate each node with its parent's label. For example (assuming that the parent of the upper NP is VP):

(6.13)

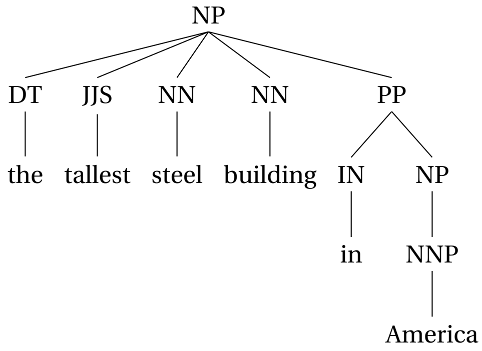


Now, the parser will not be tempted to build a three-level NP (because it would require an NP[*mom* = NP] with an NP[*mom* = NP] child, which is rare). We train the PCFG on these annotated trees, and then after we parse the test data, we have to remove the annotations before evaluation. This helps the accuracy of the parser considerably (to about 77% F1).

6.3 Binarization and horizontal markovization

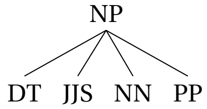
On the other hand, our PCFG also captures too much dependency. Suppose the Treebank contains the tree fragment

(6.14)



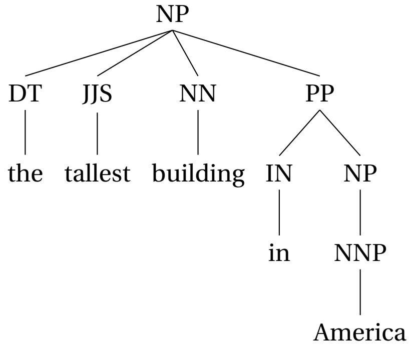
but never contains

(6.15)



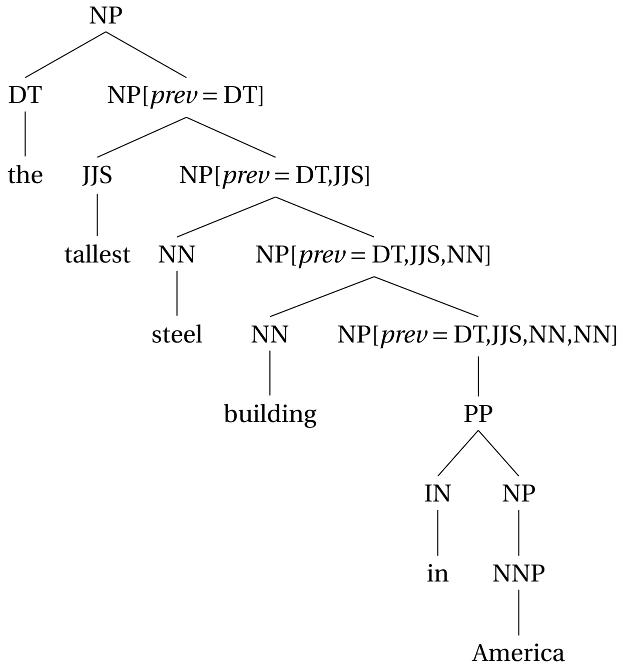
Then the parser will fail trying to parse:

(6.16)



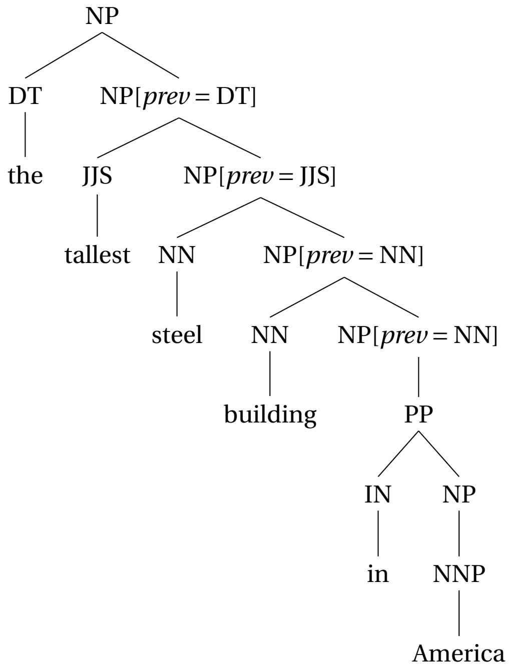
The problem is that if we allow long rules, then there are many possible long rules, which our models says are all independent. But we believe that there is some relationship between them. The solution is to break down the long rules into smaller rules, just as we did to reduce parsing complexity. Here, it's easier to binarize the trees instead of binarizing the grammar. For example, to binarize (6.14), we introduce new NP nodes, and annotate each one with the children that have been generated so far:

(6.17)



Note that there is enough information in the annotations to reverse the binarization. So much information, in fact, that we still can't parse (6.16). We can again apply an idea from language modeling, this time in the horizontal direction: make the generation of each child depend only on the previous $(n-1)$ children Miller et al., 1996; Collins, 1999; Klein and Manning, 2003. For example, if $n = 2$:

(6.18)



Now we can parse (6.16), and the parser accuracy should be a little bit better.

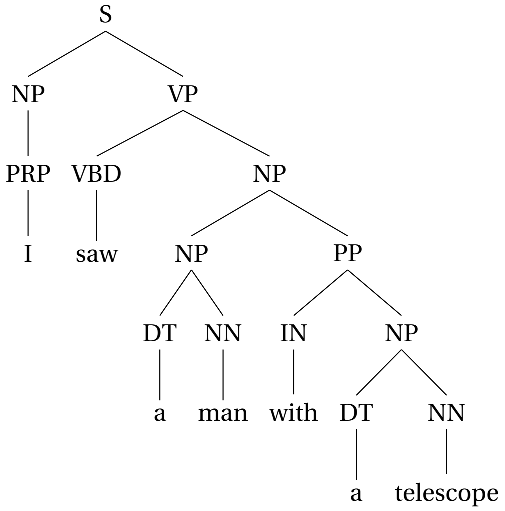
6.4 Using linguistic knowledge

Previously we saw how to increase the amount of vertical context dependency in a PCFG by changing it, effectively, from a bigram model to a trigram model, and how to decrease the amount of horizontal context dependency by changing it, effectively, from a ∞-gram model to a bigram model. We can try to use linguistic knowledge to make these context dependencies more intelligent.

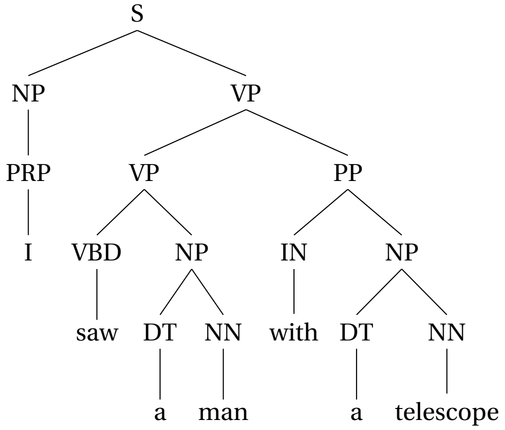
6.4.1 Lexicalization

In the vertical direction, a common technique is *lexicalization* (sometimes called *head-lexicalization* to distinguish it from another concept with the same name). In English parsing, *PP attachment* is one of the most difficult ambiguities to resolve, as illustrated by the well-known sentence:

(6.19)

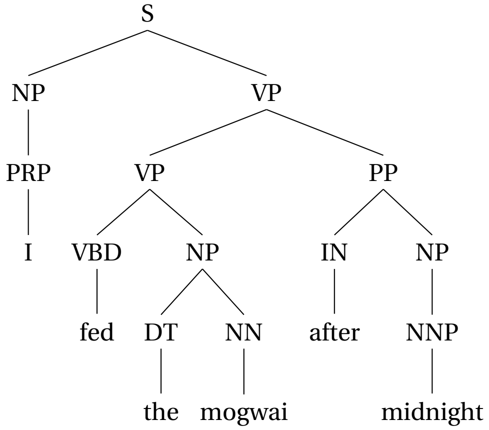


(6.20)



Although there is a strong general preference for low attachment (6.19), the words involved may change this preference. For example, *after* would have a definite preference for attaching to VP.

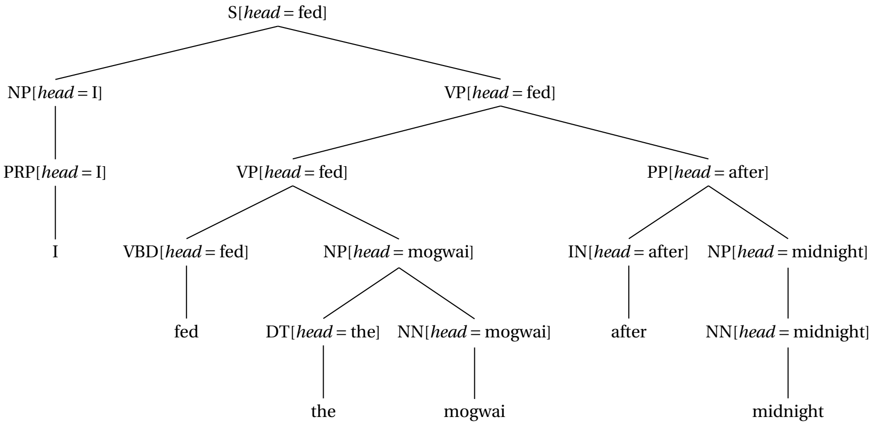
(6.21)



Last time, we annotated each node with the label of its parent; now, we go in the opposite direction, annotating each node with the label of one of its leaves. Which one? We choose the linguistically "most important" one, known as its *head* word, using some heuristics (e.g., the head of a VP is the verb; the head of an NP is the final noun).

For example, tree (6.21) would become:

(6.22)



What did this buy us? We are going to learn a high probability for rules like

$$\text{VP}[\mathit{head} = w] \rightarrow \text{VP}[\mathit{head} = w]\ \text{PP}[\mathit{head} = \text{after}] \tag{6.23}$$

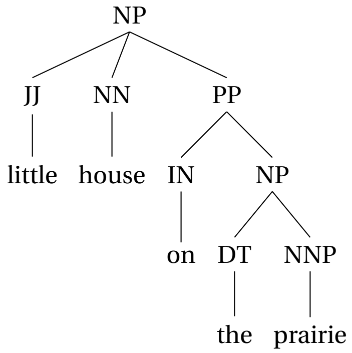
and low probability for rules like

$$\text{NP}[\mathit{head} = w] \rightarrow \text{NP}[\mathit{head} = w]\ \text{PP}[\mathit{head} = \text{after}] \tag{6.24}$$

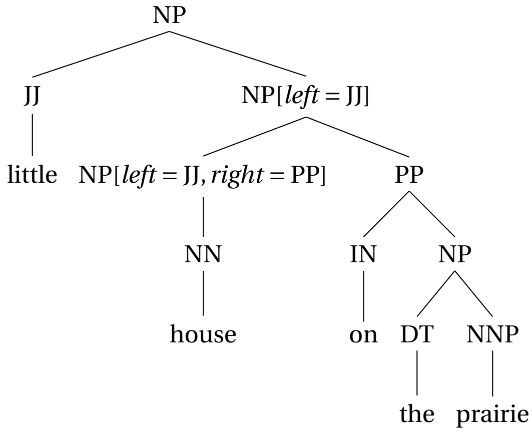
so that we can learn that PPs headed by *after* prefer to attach to VPs instead of NPs.

If we binarize, it is convenient to binarize so that the head is generated last (lowest). Thus:

(6.25)



(6.26)



6.4.2 Subcategorization

In the horizontal direction, a common technique is to use *subcategorization.* The basic idea is that some phrases (called *arguments*) are required and others (called *adjuncts*) are optional:

(6.27) Godzilla obliterated the city

(6.28) ? Godzilla obliterated

The verb *obliterated* normally takes a direct object, making the second sentence odd. On the other hand, in the sentences

(6.29) Godzilla exists

(6.30) * Godzilla exists the monster

the verb *exists* never takes a direct object. By contrast, adjuncts can occur much more freely:

(6.31) Godzilla exists today

(6.32) Godzilla obliterated the city today

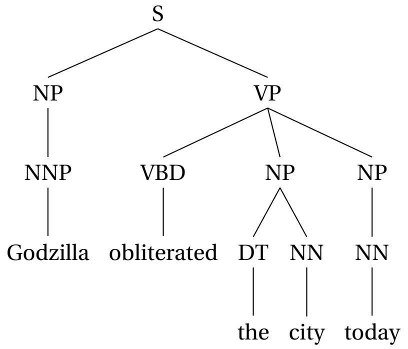
This can affect parsing decisions. For example, (6.33) I saw her duck

(6.34) I obliterated her duck

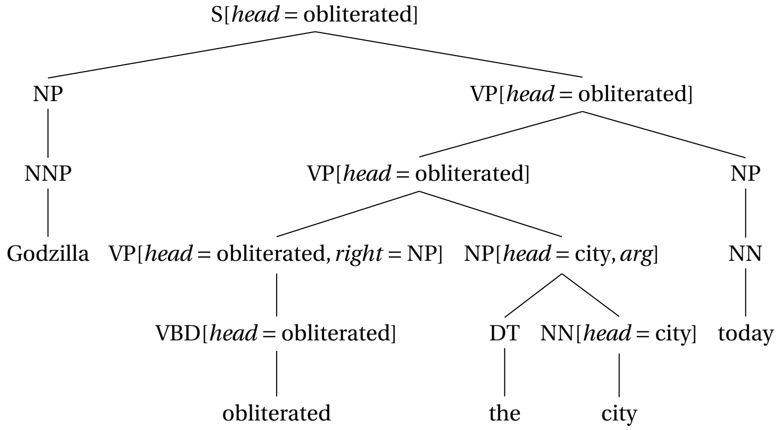
The first sentence is ambiguous for humans because *saw* can take either an NP or an S as an argument. The second sentence is unambiguous for humans, but ambiguous for computers unless they learn that *obliterated* must take an NP argument, not an S argument.

Last time, we made the generation of a child node depend on one previous child. Now, we would like to use this same mechanism to control the number of arguments, depending on the verb. We can do this by making the generation of a child node depend on all of the previous arguments, and none of the previous adjuncts. I've left off some annotations to save space:

(6.35)



(6.36)



We marked [$_{\text{NP}}$ the city] with an *arg* feature to indicate that it is an argument, not an adjunct. Moreover, the *right* feature, and the *left* feature if there were one, only keeps track of the previous arguments, not adjuncts.

6.5 Practical Details

6.5.1 Smoothing

With the complex nonterminals we have been creating, it may become hard to reliably estimate rule probabilities from data. The solution is to apply smoothing, as in language modeling. Witten-Bell smoothing is a fairly common choice in parsing. For example, to estimate the probability of

$$\mathrm{VP}[\mathit{head} = \text{obliterated}] \rightarrow \mathrm{VP}[\mathit{head} = \text{obliterated}, \mathit{right} = \mathrm{NP}] \quad \mathrm{NP}[\mathit{head} = \text{city}, \mathit{arg}]$$

we might interpolate its relative-frequency estimate with that of

$$\mathrm{VP}[\mathit{head} = w] \rightarrow \mathrm{VP}[\mathit{head} = w, \mathit{right} = \mathrm{NP}] \quad \mathrm{NP}[\mathit{head} = \text{city}, \mathit{arg}]$$

where we have replaced the word *obliterated* with a placeholder w to make the rule probability easier to estimate.

If we test our parser on unseen data, it is inevitable that it will encounter unseen words. If we don't do anything about it, the parser will simply reject any string that has an unknown word, which is obviously bad.

The simplest thing to do is to simulate unknown words in the training data. That is, in the training data, replace every word that occurs only once (or $\leq k$ times) with a special symbol `<unk>`. Then train the PCFG as usual. Then, in the test data, replace all unknown words with `<unk>`. It's also fine to use multiple unknown symbols. For example, we can replace words ending in *-ing* with `<unk-ing>`.

A more sophisticated approach would be to apply some of the ideas that we saw in language modeling.

6.5.2 Beam search

The Viterbi CKY algorithm can be slow, especially if modifications to the grammar increase the nonterminal alphabet a lot. We can use *beam search* to speed up the search if we are willing to allow potential search errors.

After the completion of each chart cell $\mathit{best}[i, j]$, do the following:

```
 1: for all $X \in N$ do
 2:     $\mathit{score}[X] \leftarrow \mathit{best}[i,j][X] \times h(X)$
 3: end for
 4: choose $\mathit{minscore}$
 5: for all $X \in N$ do
 6:     if $\mathit{score}[X] < \mathit{minscore}$ then
 7:     end if
 8:     delete $\mathit{best}[i,j][X]$
 9:     delete $\mathit{back}[i,j][X]$
10: end for
```

The function $h(X)$ is called a *heuristic* function and is meant to estimate the relative probability of getting from S at the root down to X. The typical thing to do is to let $h(X)$ be the frequency of X in the training data.

There are two common ways of choosing *minscore* (line 4):

- $\mathit{minscore} = \left(\max_X \mathit{score}[X]\right) \times \beta$, where $0 < \beta < 1$ (typical values: 10^{-3} to 10^{-5})
- *minscore* is the score of the b'th best member of *score* (typical values of b: 10–100)

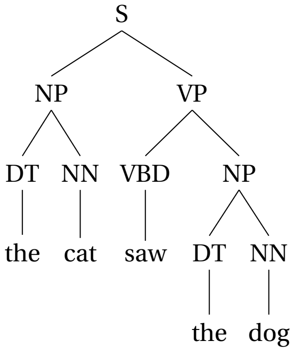
It is also fine to set *minscore* to the larger of these two values.

Question The time complexity of CKY is normally $\mathcal{O}(n^3|N|^3)$, because we have to loop over i, j, k, X, Y, and Z. If we add beam search, what will the time complexity be in terms of n and b? Assume $b < |N|$.

6.6 Partially Unsupervised Training

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. This is possible, and quite effective (Matsuzaki, Miyao, and Tsujii, 2005; Petrov et al., 2006).

Suppose our first training example is the following tree, T:



And suppose that our initial grammar is:

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

Notice the bracketed numbers that we have added to the nonterminals. This grammar has many possible derivations, all of which generate the same tree *modulo* the bracketed numbers. We've initialized the rule probabilities randomly, and our goal is to optimize the rule probabilities to maximize the (log-)likelihood of the trees in the training data. The hope is that we can automatically learn ways of augmenting the nonterminals that perform as well or better than the linguistically-motivated augmentations we saw earlier.

To compute $P(T)$, the total probability of all derivations of tree T, we use CKY, just as in parsing, but with two differences. First, because we're given T, the search is constrained to nodes of T instead of all possible spans. Second, whenever we have to merge two duplicate entries in a cell, instead of choosing the maximum probability, we add probabilities.

```
**Require:** tree $T$ and grammar $G = (N, \Sigma, R, S)$
**Ensure:** $\mathit{best}[\mathrm{root}(T)][S]$ is the total weight of all derivations of $T$
initialize $\mathit{best}[\eta][X] \leftarrow 0$ **for all** nodes $\eta \in T$ and $X \in N$
**for all** leaf nodes $\eta$ labeled $X$ and rules $(X[q] \xrightarrow{p} w_i) \in R$ **do**
    $\mathit{best}[\eta][X] \leftarrow \mathit{best}[\eta][X]\{\mathit{best}[\eta][X], p\}$
**end for**
**for** non-leaf nodes $\eta$, bottom-up **do**
    let $\eta_1$ and $\eta_2$ be the children of $\eta$
    let $X$ be label of $\eta$ and $X_1, X_2$ labels of its children
    **for all** $(X[q] \xrightarrow{p} X_1[r] X_2[s]) \in R$ **do**
        $p' \leftarrow p \times \mathit{best}[\eta_1][X_1] \times \mathit{best}[\eta_2][X_2]$
        $\mathit{best}[\eta][X] \leftarrow \mathit{best}[\eta][X] + p'$
    **end for**
**end for**
```

To train, we loop through all the trees in the training data, and for each one, we use SGD to increase the log-likelihood of $P(T)$, as computed above.

With some additional tricks Petrov et al., 2006, this method can be made to learn a different number of q's for each nonterminals, and the result is a very good parser. Other parsers have surpassed it in parsing accuracy, but this method remains the best way to train a PCFG.

Bibliography

Black, E. et al. (1991). “A procedure for quantitatively comparing the syntactic coverage of English grammars”. In: *Proc. DARPA Speech and Natural Language Workshop*, pp. 306–311.

Collins, Michael (1999). “Head-Driven Statistical Models for Natural Language Parsing”. PhD thesis. University of Pennsylvania.

Johnson, Mark (1998). “PCFG models of linguistic tree representations”. In: *Computational Linguistics* 24, pp. 613–632.

Klein, Dan and Christopher D. Manning (2003). “Accurate Unlexicalized Parsing”. In: *Proc. ACL*, pp. 423–430.

Matsuzaki, Takuya, Yusuke Miyao, and Jun’ichi Tsujii (2005). “Probabilistic CFG with Latent Annotations”. In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL’05)*, pp. 75–82. URL: `http://www.aclweb.org/anthology/P05-1010`.

Miller, Scott et al. (1996). “A Fully Statistical Approach to Natural Language Interfaces”. In: *Proc. ACL*, pp. 55–61.

Petrov, Slav et al. (2006). “Learning Accurate, Compact, and Interpretable Tree Annotation”. In: *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pp. 433–440. URL: `http://www.aclweb.org/anthology/P06-1055`.