6.864 Advanced Natural Language Processing[1]

Lecture 9: Parsing and Probabilistic Context Free Grammar

8 October 2015

[1] Instructors: Prof. Regina Barzilay, and Prof. Tommi Jaakkola.
TAs: Franck Dernoncourt, Karthik Rajagopal Narasimhan, Tianheng Wang.
Scribes: Clare Liu, Evan Pu, Kalki Seksaria

This lecture first discusses the evaluation of and the limitations of Hidden Markov Models for unsupervised part-of-speech (POS) tagging. The lecture then introduces the concepts of parse trees, context-free grammars (CFGs), and probabilistic context-free grammars (PCFGs). The CYK algorithm is then presented as an efficient method that uses dynamic programming to find the best parse tree for a sentence with a PCFG. Finally, the limitations of probabilistic context-free grammars are discussed and evaluated.

Evaluation of Unsupervised POS tagging

In the previous lecture we have shown how to solve the part-of-speech (POS) tagging problem by using a Hidden Markov Model (HMM). The hidden states of the HMM are supposed to correspond to parts of speech. However, the model is not given any linguistic information. Furthermore, we do not have the correspondence between numeric hidden states and actual POS tags, such as "noun" or "verb". To evaluate the accuracy of the HMM, we can use a matching algorithm, which attempts to construct a correspondence between hidden states and POS tags that maximizes the POS tagging accuracy. The goal of this matching is to see what POS each hidden state corresponds to.

Let's assume that the HMM has two possible hidden states, y_1 and y_2, and that there are three possible POS tags, N, V, and ADJ. Consider the two example sentences "Strong stocks are good" and "Big weak stocks are bad" (Figure 1). The HMM has assigned a hidden state (y_1 or y_2) to each word. In addition, we also know the correct POS tag for each word. Our goal is to use a matching algorithm to match each hidden state to the POS tag that would give us the highest score.

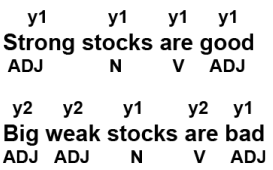

Figure 1: Here are two example sentences annotated with the hidden states generated by an HMM above and the gold-standard POS tags below. We want to find a matching between the hidden states and POS tags to maximize the number of words tagged correctly.

Matching

A matching is a bipartite graph between part-of-speech tags and hidden states. Two types of matchings can be performed: a 1-to-many matching or a 1-to-1 matching. The definitions of 1-to-many and 1-to-1 matchings are given below. Both types of matchings have different advantages and disadvantages when they are used to evaluate the performance of an unsupervised model.

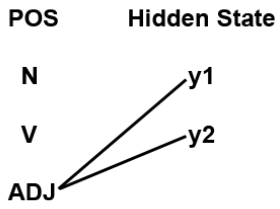

Figure 2: Example of a 1-to-many matching. Multiple hidden states are matched to the ADJ tag.

1-to-Many Matching In a 1-to-many matching, each hidden state is assigned to with the POS tag that gives the most correct matches. Each

POS tag can be matched to multiple hidden states, hence the name "1-to-many".

1-to-1 Matching In contrast, in a 1-to-1 matching, each POS tag can only be matched to a maximum of one hidden state. In this case, if the number of hidden states is greater than the number of tags, then some hidden states will not correspond to any tag. This will decrease the reported accuracy.

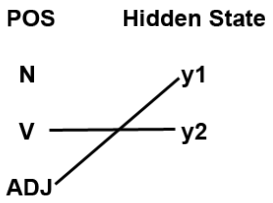

Figure 3: Example of a 1-to-1 matching. Each part-of-speech tag has a maximum of one hidden state that is matched with it.

Evaluation

Given a matching, the accuracy of the HMM tagging can be computed against a gold-standard corpus, where each word has been assigned the correct POS tag (based on the Penn WSJ Treebank). The correct tag for a word is compared with the tag that is matched with the word's hidden state. If these two tags match, this means that the tagging is correct for this word.

To find a matching, we can create a table that contains the counts of each correct POS tag for each hidden state. Here is an example evaluation table for the example sentences from Figure 1.

Hidden State	POS Tag	Count
y1	N	2
y1	V	1
y1	ADJ	3
y2	N	0
y2	V	1
y2	ADJ	2

Figure 4: An example evaluation table for the sentences from Figure 1. State y_1 has been matched with 2 nouns, 1 verb, and 3 adjectives. State y_2 has been matched with 0 nouns, 1 verb, and 2 adjectives.

The matching between hidden states and POS tags can then be found based on this table for both 1-to-many and 1-to-1 matchings.

Finding a good matching

As we can see, the accuracy of the HMM depends on the type of matching that is chosen (1-to-many or 1-to-1). To find a good matching, we can use a greedy algorithm that chooses individual state to tag matchings one at a time and picks the new matching that maximally increases the accuracy at each step. The algorithm terminates when no more matchings can be made. In a 1-to-1 matching, the algorithm terminates when either each POS tag has been assigned a hidden state or each hidden state has received a POS tag (whichever happens first). In a 1-to-many matching, the algorithm terminates when each hidden state has received a POS tag.

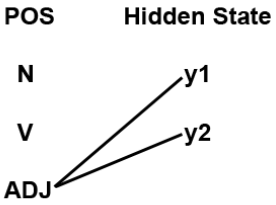

Figure 5: The matching that we get from maximizing the scores from the one-to-many evaluation table.

Example: 1-to-many Matching Consider the table from Figure 4. The highest count we see in the table is 3, which corresponds to y_1 being assigned the ADJ tag. Thus, we greedily assign state y_1 to ADJ. Since the highest count for y_2 is also ADJ, state y_2 is also assigned to the ADJ tag. y_1 tags 3 words correctly, y_2 tags 2 words correctly, and there are 9 words in total, so the best score from a 1-to-many matching is $\frac{5}{9}$. The bipartite matching is illustrated in Figure 5.

Hidden State	POS Tag	Count
y1	N	2
y1	V	1
y1	**ADJ**	**3**
y2	N	0
y2	V	1
y2	**ADJ**	**2**

Figure 6: This table corresponds to a 1-to-many matching. As we can see, ADJ is assigned to multiple hidden states because it leads to the highest score for both states

Example: 1-to-1 Matching We know that the highest count for y_1 is 3, which corresponds to an adjective, so we again greedily assign state y_1 to ADJ. Even though state y_2 also has the highest count for adjectives, ADJ has already been matched to state y_1 so we cannot use it again. Thus, y_2 must be assigned to V, which has the highest count that is not yet assigned to any state. y_1 tags 3 words correctly and y_2 tags 1 word correctly, so the total score for a 1-to-1 matching is $\frac{4}{9}$. See Figure 8 for the matching.

Hidden State	POS Tag	Count
y1	N	2
y1	V	1
y1	**ADJ**	**3**
y2	N	0
y2	**V**	**1**
y2	ADJ	2

Figure 7: This table corresponds to a 1-to-1 matching. Even though choosing ADJ would give the highest score for state y_2, we cannot repeat POS tags, so it must be assigned to V instead.

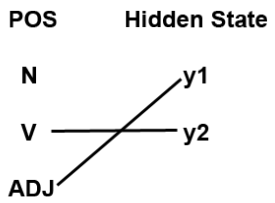

Figure 8: The matching that we get from maximizing the scores from the 1-to-one evaluation table.

Pitfalls of the HMM

Fully unsupervised tagging models generally perform very poorly. The HMM computed with the EM algorithm tends to give a relatively uniform distribution of hidden states, while the empirical distribution for POS tags is highly skewed since some tags are much more common than others. As a result, any 1-to-1 matching will give a POS tag distribution that is relatively uniform, which results in a low accuracy score.

A 1-to-many matching can create a non-uniform distribution by matching a single POS tag with many hidden states. However, as the

number of hidden states increases, the model could overfit by giving each word in the vocabulary its own hidden state. If each word has its own hidden state, we would achieve 100 percent accuracy, but the model would not give us any useful information. For more details on the evaluation of HMM for POS tagging, see "Why doesn't EM find good HMM POS-taggers?" (hyperlinked) by Mark Johnson.

Improving Unsupervised Models

There are several modifications we can make to improve the accuracy of unsupervised models:

1. Supply the model with a dictionary and limit possibilities for each word: For example, we know that the word "train" can only be a verb or a noun. This limits the set of possible tags for each word, which will push the model in the correct direction.

2. Prototypes: Provide each POS tag with several representative words for the tag. Even just a few words for each tag will help push the model in the right direction.

Parse Trees

We now move from POS tagging to sentence parsing. A parse tree is a rooted tree that represents the syntactic structure of a sentence according a grammar. In syntactic parsing, each word in a sentence is tagged with a POS tag, and groups of POS tags and/or phrases are labeled with a single phrase tag. For example, the root of a parse tree is the tag "S", describing the whole sentence. The sentence "S" is commonly broken into an "NP" (noun phrase) and "VP" (verb phrase). The lowest level of the parse tree consists of the words of the sentence. We will now informally describe several applications of parse trees in NLP. Figure 9 gives an example of a syntactic parse tree.

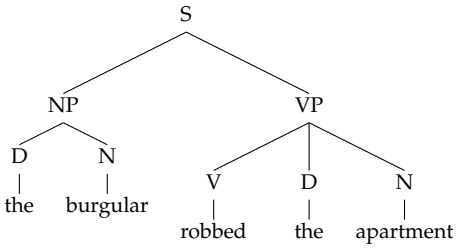

Figure 9: An example of a parse tree for the sentence "The burgular robbed the apartment"

Applications of Parse Trees

Grammar Checking Syntactic parse trees can be used for grammar checking/correction. If a sentence cannot be parsed against a given grammar, this could indicate a grammar error. Grammar suggestions can also be generated by comparing the sentence against the given grammar.

Machine Translation Parse trees can also be used in machine translation, where a source language is first parsed against the source lan-

guage's grammar, and the resulting tree is transformed to produce a sentence in the target language.

Semantic Role Labeling In semantic role labeling, a sentence is decomposed into a *predicate*, which represents a key concept of a sentence, and *arguments*, which take on different roles according to the predicate. For example, in the sentence "Mary sold the book to John", the verb "sold" is the predicate. The arguments are as follows: "Mary" is the seller, "the book" is the good, and "John" is the recipient. Note the sentence "The book was sold by Mary to John", while syntactically different, has the same semantic role labeling as before. Figure 10 shows another example tree for semantic role labeling.

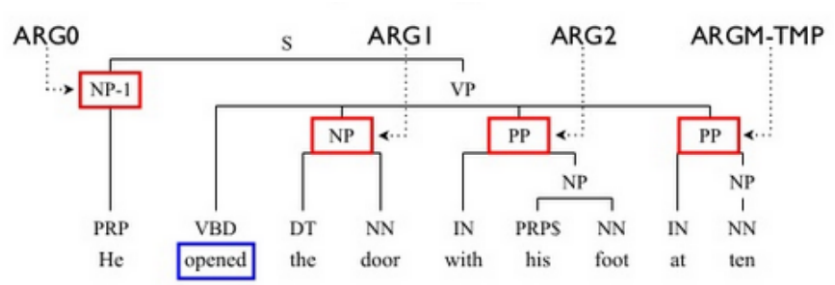

Figure 10: An example of Semantic Role Labeling, where the word "opened" is labeled as the predicate.

Discourse Parsing In discourse parsing, groups of related sentences, which are called discourses, are parsed as a group. This reveals the structure of a piece of text, and also shows us the relationships between the different sentences. The techniques used for discourse parsing are similar to those used for syntactic parsing. While words form the leaves of a syntactic tree, whole phrases or sentences form the leaves of a rhetorical structure tree. Figure 11 shows an example rhetorical structure tree for discourse parsing.

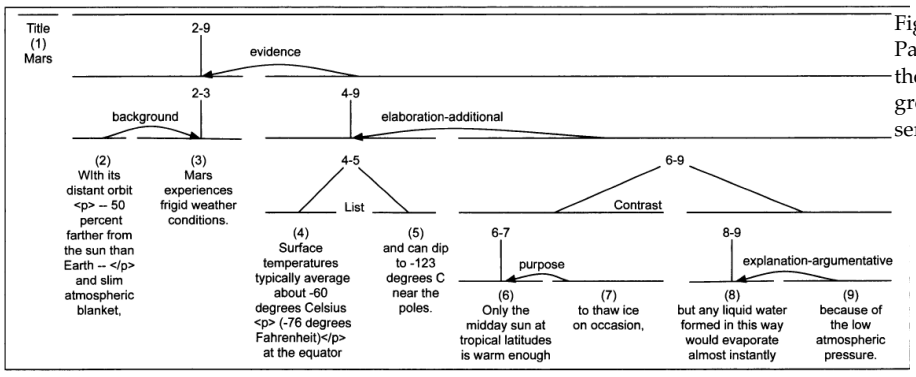

Figure 11: An example of Discourse Parsing that explains the structure of the given text. Sentence 2 acts as background for sentence 3, and sentences 4-9 serve as evidence for sentences 2-3.

Information Extraction In information extraction, structured information is extracted from a sentence. Information extraction can be used to populate a database directly from sentences instead of having to manually fill in the required fields. Information extraction is used to produce calendar notifications and contact cards directly from email text. Figure 12 shows an example tree for information extraction.

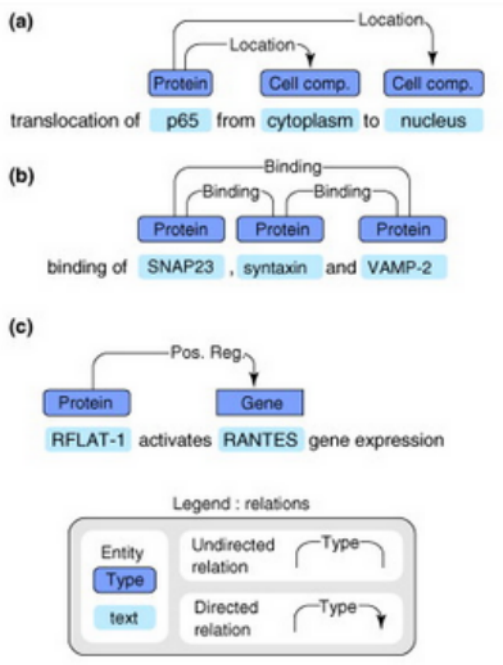

Figure 12: An example of information extraction where in a), the parse tree annotates that cytoplasm and nucleus are possible locations for the protein p65.

Datasets for Training Syntactic Parse Trees

In order to train a syntactic parser, annotated training data is required. Datasets for training parse trees include:

1. The Penn WSJ (Wall Street Journal) Treebank is a collection of 50,000 sentences with their associated trees. It is commonly used in training and test sets for parsing. However, the Penn WSJ Treebank only contains English sentences, so it cannot be used for other languages. Furthermore, the WSJ does not contain the full range of possible English sentences, as it is a newspaper focused on business. There is some overfitting in training and testing a parser on WSJ data only.

2. Universal Dependencies is a more recent data source for multilanguage parsing. It is available at `https://universaldependencies.github.io/docs/`.

Context-Free Grammar

In order to generate a parse tree, a grammar is required. In order to reduce the problem complexity and save computation time, We will make the assumption that the grammar is context-free. This assumption is very good for English, as no counter-examples have been found. In addition, across languages, only a few rare constructions violate this assumption. A context-free grammar (CFG) contains a set of symbols and rules. Each symbol can be either terminal or nonterminal. Terminal symbols correspond to words in the vocabulary and nonterminal symbols correspond either to parts of speech or to groupings of words, such as noun phrases and verb phrases. Each rule maps a nonterminal symbol to a set of nonterminal or terminal symbols. A CFG formalizes how a string (sentence) may be generated from its rules. The application of these rules forms a tree structure, which is the syntax tree (parse tree) for the string. In this section we will only discuss the generation of strings for a given CFG. The inverse operation, parsing, which takes in a string and produces a parse tree for the string, will be discussed later.

Formal Definition

A context-free grammar is defined by $G = (\mathcal{N}, \Sigma, R, S)$ where:

$\mathcal{N}$ = set of nonterminal symbols

Σ = set of terminal symbols

R = set of production rules of the form $X \rightarrow Y_1 ... Y_n$, where X is a nonterminal symbol and each Y_i can be a terminal or a nonterminal symbol.

S = starting symbol

Example Context-Free Grammar

$\mathcal{N} = \{S, NP, VP, PP, DT, VI, VT, NN, IN\}$

$S = S$

$\Sigma =$ {sleeps, saw, man, woman, telescope, the, with, in}

Generation of Strings with a CFG

We can generate strings with symbols in the terminal set Σ from a CFG by using a *left-most derivation* strategy. Here is a non-deterministic algorithm describing the left-most derivation strategy, where the input A is a symbol, and the output is a string:

$R =$

S ⇒ NP VP	VI ⇒ sleeps
VP ⇒ VI	VT ⇒ saw
VP ⇒ VT NP	NN ⇒ man
VP ⇒ VP PP	NN ⇒ woman
NP ⇒ DT NN	NN ⇒ telescope
NP ⇒ NP PP	DT ⇒ the
PP ⇒ IN NP	IN ⇒ with
	IN ⇒ in

Figure 13: The rules for the example context-free grammar

```
left_most_derivation(A):
     if terminal(A):
         return A
     rule = choose (rules(A))
     rhs = right_hand_side(rule)
     return concatenate([left_most_derivation(A') for A' in rhs])
```

If the symbol is a terminal, the algorithm terminates. If the symbol is non-terminal, the non-deterministic operator *chooses* one rule where the left-hand-side is A. The algorithm then recursively expands the list of symbols on the right-hand-side, starting with the left-most symbol and concatenates the result. //

To generate a string from a CFG, we call this algorithm on the starting symbol S, which represents an entire sentence. The non-deterministic derivations used in generating the string will form the parse tree of the string.

Language of a CFG

Given a CFG, one can define the language generated by the CFG as follows: $L = \{s \in \Sigma^* | s = left\_most\_derivation(S)\}$ where Σ^* is the set of strings formed by terminals in Σ. In other words, a string belongs to the language of a CFG if there exists a sequence of left-most derivations that can generate the string.

Example of left-most derivation

We can visualize how the grammar defined earlier generates the sentence "The woman saw the man with the telescope." in Figures 14-16, along with their parse trees.

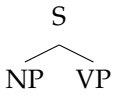

Figure 14: First, the starting symbol S is expanded into NP and VP.

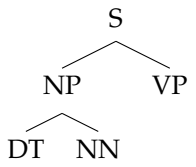

Figure 15: Next, *S*'s left child, NP, is expanded into DT and NN.

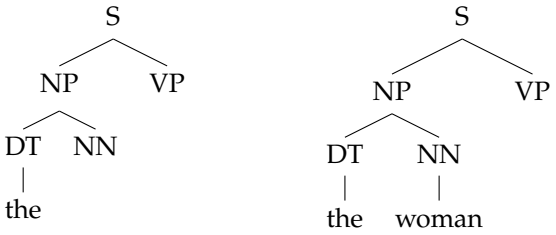

Figure 16: Next, the left child, DT, is expanded into "the", which is a terminal. Then we recursively go back to NP and expand its right child (NN), which gives us the terminal symbol "woman". Since the right side of NP has reached a terminal, we have now completed the left-most derivation for the symbol NP. The next step is to expand the right child of *S*.

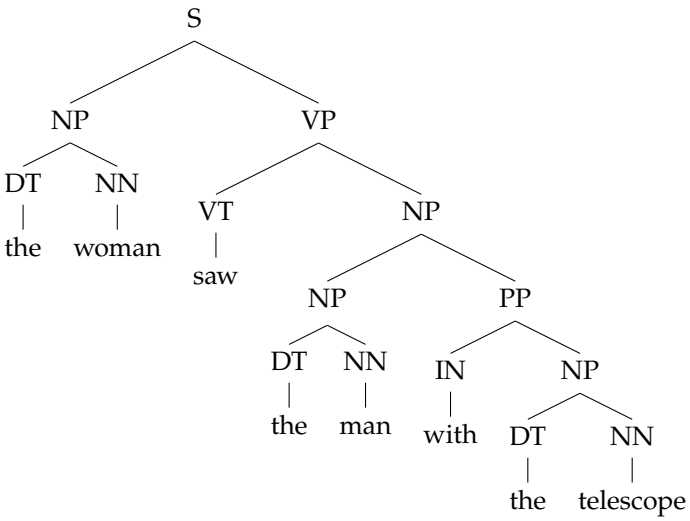

Figure 17: By completing this process of left-most derivation, we obtain a full parse tree. This tree shows one possible derivation and the resulting parse tree for the sentence "The woman saw the man with the telescope". Under this parsing, we interpret the sentence as stating that *the man who the woman saw had a telescope*.

Ambiguities

Given a CFG, there may be multiple possible derivations using the left-most-derivation strategy that generate the same string, but give different parse trees. These sentences have an ambiguous meaning, as each parse tree represents a different meaning. Figure 18 shows another way of parsing the example sentence "The woman saw the man with the telescope". Ambiguities are troublesome because

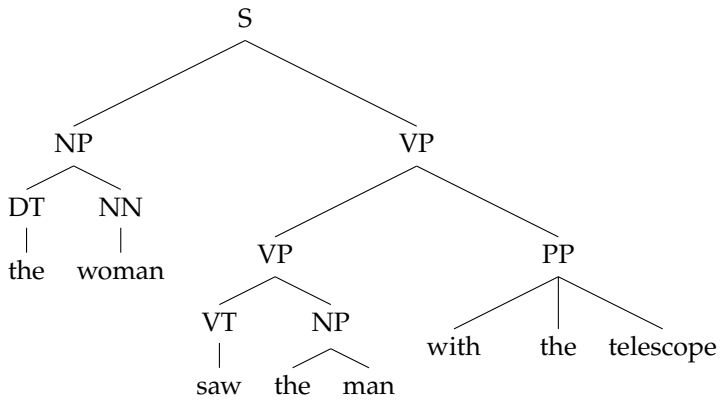

Figure 18: Another possible parse tree. Under this parsing, the woman is using a telescope to see the man.

one would like to have a way of preferring one parse tree over another. We can partially address this issue by augmenting the rules in R with probabilities, making certain derivations more likely than others.

Probabilistic Context-Free Grammar

A Probablistic Context Free Grammar (PCFG) is a CFG where each rule is augmented with a conditional probability of using that rule given the left-hand-side non-terminal. The conditional probabilities are written as $P(\alpha \rightarrow \beta|\alpha)$, where α is a non-terminal, and β is the right-hand-side derivation of the rule. Given a non-terminal, all the probabilities of the rules that apply to this non-terminal must add up to 1, i.e. $\sum_\beta P(\alpha \rightarrow \beta|\alpha) = 1$. This forms a probability distribution over the possible derivations.

Estimating Probabilities in PCFG

One way of learning these probabilities is to use a gold-standard corpus such as the Penn Treebank, and use the empirical distributions of the derivations. For example, if we have the rule $S \rightarrow NP\ VP$, the estimated conditional probability is equal to:

$$P(S \rightarrow NP\ VP|S) = \frac{count(S \rightarrow NP\ VP)}{count(S)}.$$

PCFG as a Language Model

The probability that a particular tree T (representing sentence s) is generated from the PCFG can be calculated by finding the product of the probabilities of all of the rules used in the tree:

$$P(T, s) = \Pi_i P(\alpha_i \rightarrow \beta_i | \alpha_i)$$

Furthermore, the probability of the sentence s being generated is equal to the sum of the probabilities of all of its possible parse trees:

$$P(s) = \sum_{T \in T(s)} P(T, s)$$

where $T(s)$ is the set of all parse trees that can generate s.

PCFG Example:

Here is a PCFG where we augmented the CFG of the previous example with conditional probabilities.

S ⇒ NP VP	1.0
VP ⇒ VI	0.4
VP ⇒ VT NP	0.4
VP ⇒ VP PP	0.2
NP ⇒ DT NN	0.3
NP ⇒ NP PP	0.7
PP ⇒ IN NP	1.0

VI ⇒ sleeps	1.0
VT ⇒ saw	1.0
NN ⇒ man	0.7
NN ⇒ woman	0.2
NN ⇒ telescope	0.1
DT ⇒ the	1.0
IN ⇒ with	0.5
IN ⇒ in	0.5

Figure 19: The rules for the context-free grammar from Figure 13, with added conditional probabilities

We can now determine the probabilities of the two ambiguous parse trees for the sentence "The woman saw the man with the telescope".

Tree 1:

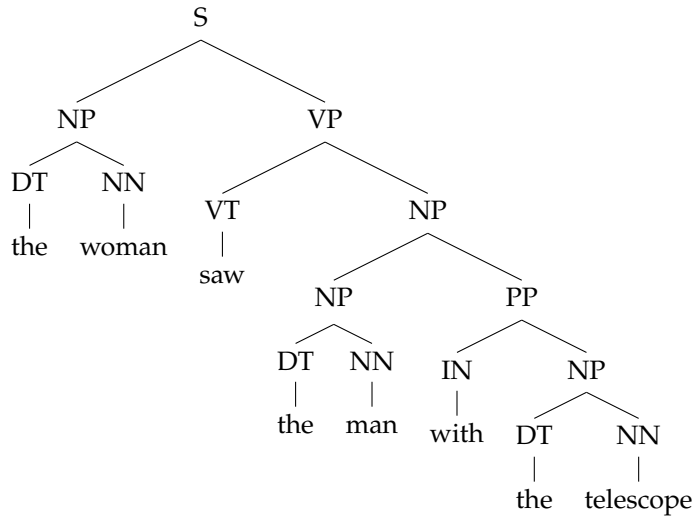

Figure 20: The parse tree from Figure 17 with probabilities for the derivations.

$$P(T_1, S) = 1 \cdot 0.3 \cdot 1 \cdot 0.2 \cdot 0.4 \cdot 1 \cdot 0.7 \cdot 0.3 \cdot 1 \cdot 0.7 \cdot 1 \cdot 1 \cdot 0.3 \cdot 1 \cdot 0.1 \approx 1 \times 10^{-4}$$

Tree 2:

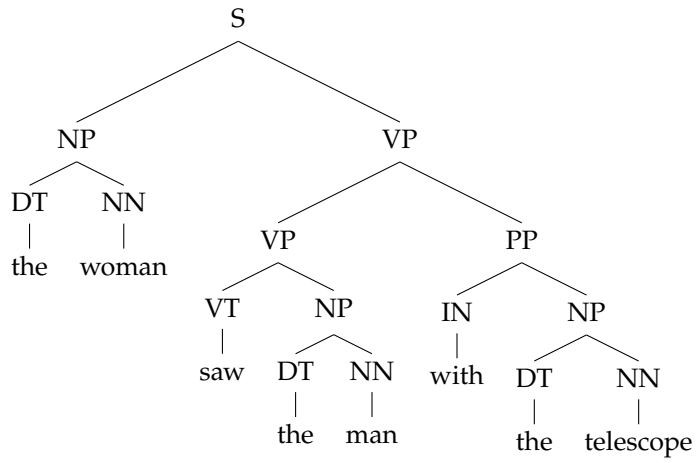

Figure 21: The parse tree from Figure 18 with probabilities for the derivations.

$$P(T_2, S) = 1 \cdot 0.3 \cdot 1 \cdot 0.2 \cdot 0.2 \cdot 0.4 \cdot 1 \cdot 0.3 \cdot 1 \cdot 0.7 \cdot 1 \cdot 1 \cdot 0.3 \cdot 1 \cdot 0.1 \approx 3 \times 10^{-5}$$

We see that the first parse tree is more likely under this PCFG. However, to compute the probability of the sentence "The woman saw the man with the telescope", we need to sum over all the possible parses of it, which can quickly become intractable as there can be an exponential number of parses. In the next section we resolve this issue with the CYK algorithm.

The CYK Algorithm

In this section, we explain the CYK parsing algorithm, that can achieve the following tasks:

1. Language modeling: Given a sentence s, find the probability of the PCFG generating the sentence. Formally:

$$P(s) = \sum_{T \in T(s)} P(T, s)$$

2. Syntactic parsing: Given a sentence s, find the parse tree T that derives the sentence with the highest probability. Formally:

$$T^* = argmax_{T \in T(s)} P(T, s)$$

We will describe the CYK algorithm that solves the first task in detail, and show a simple modification that allows it to solve the second task.

To find the probability of generating a particular sentence s given the PCFG, we calculate the probability of every possible derivation $T \in T(s)$, and sum them:

$$P(s) = \sum_{T \in T(s)} P(T, s)$$

The CYK algorithm solves this summation over a set of exponential size using Dynamic Programming. However, it requires the CFG to be in Chomsky Normal Form (CNF).

Chomsky Normal Form

Chomsky Normal Form restricts the set of allowable rules. A grammar is in Chomsky Normal Form if each rule either converts a nonterminal symbol into two nonterminal symbols or a single terminal symbol:

$$X \to Y_1 Y_2$$

$$X \to y$$

where $X, Y_1, Y_2 \in N$ and $y \in \Sigma$.

It can be shown that every CFG can be converted to CNF. See `http://www.cs.nyu.edu/courses/fall07/V22.0453-001/cnf.pdf` for the details. For example, the rule $NP \rightarrow DT\ ADJ\ NN$ can be converted into the following two rules that satisfy CNF:

$$NP \rightarrow DT\ ADJP$$

$$ADJP \rightarrow ADJ\ NN$$

The Algorithm

The CYK algorithm is a dynamic programming algorithm. Like any other dynamic programming algorithm, it has 2 components: The sub-problems, and the recurrence relation between the sub-problems.

Given a sentence s, let $s[i, j]$ denote the sub-sentence starting at the ith word and ending at the jth word, inclusive.

Sub-problems

The sub-problems of the CYK algorithm are of the form:

$$\pi(i, j, N)$$

Where i and j are the indexes of the words in the sentence, given $i \leq j$, and N is a non-terminal. The value of $\pi(i, j, N)$ corresponds to the probability that $s[i, j]$ can be generated by the non-terminal N.

Recurrence Relations

The recurrence relations of the CYK algorithm are defined by the base case and the inductive case.

Base Case: $\pi(i, i, N) = P(N \rightarrow w_i | N)$

Inductive Case:

$$\pi(i, j, N) = \sum_{k,P,Q} P(N \rightarrow P\ Q | N) \cdot \pi(i, k, P) \cdot \pi(k+1, j, Q)$$

where $k \in \{i, \ldots, j-1\}$, $P \in \mathcal{N}$, and $Q \in \mathcal{N}$.

The base case gives the probability where a non-terminal N would generate the word w_i, which forms the sub-sentence $s[i, i]$.

To understand the recurrence, notice we're trying to find the value for $\pi(i, j, N)$, i.e. the probability which $s[i, j]$ can be generated by N.

We can find this probability by sub-dividing $s[i,j]$ at index k into two pieces: $s[i,k]$ and $s[k+1,j]$, and pick two non-terminals, P and Q to generate each piece. Notice the importance of Chomsky Normal Form, because each rule has to produce exactly 2 non-terminals, this is the only form of decomposition we have to consider. The probability of P generating $s[i,k]$ is $\pi(i,k,P)$, and the probability of Q to generate $s[k+1,j]$ is $\pi(k+1,j,Q)$. Therefore, given a particular index k to split, and a particular P and Q to generate each split sub-sentence, the probability of this particular derivation for $s[i,j]$ is $P(N \rightarrow P\ Q|N) \cdot \pi(i,k,P) \cdot \pi(k+1,j,Q)$. This decomposition can be visualized in Figure 22.

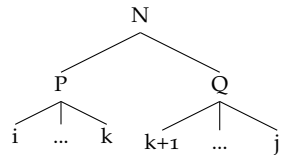

Figure 22: N generates two other non-terminals, P and Q. P spans words i through k and Q spans words $k+1$ through j. At this point, the CYK algorithm has already calculated the total probability of the trees generated by P and Q covering their respective sub-sentences. We can use these values to find the total probability of the tree generated by N.

By summing k, P, and Q over their respective domains, we obtain the total probability of generating $s[i,j]$ by non-terminal N.

The result of the CYK parser is $\pi[1,n,S]$, which is the probability that the start symbol S generates the whole sentence s. This value will be 0 if the sentence cannot be generated by our PCFG.

All that's left is to define an ordering to compute the sub-problems, for the CYK algorithm, we can order it by sub-sentence length, r.

```
# base case
for i in 1..n:
  for N in 𝒩:
    pi[i,i,N] = Pr(N->w_i|N)

# induction
for r in 2..n:
  for i in 1..(n-r):
    for N in 𝒩:
      j = i + r - 1
      pi[i,j,N] = 0

      # over all possible decompositions
      for k in i..j-1:
        for P in 𝒩:
          for Q in 𝒩:
            pi[i,j,N] += Pr(N->PQ|N)*pi[i,k,P]*pi[k+1,j,Q]

return pi[1,n,S]
```

We can also use the CYK algorithm to find the most probable parse tree for a given string. This is the syntactic parsing problem and can be solved by replacing the sum operator in the inductive case of the CYK algorithm with the max operator.

$$\pi(i,j,N) = \max_{k,P,Q} Pr(N \rightarrow P\ Q|N) \cdot \pi(i,k,P) \cdot \pi(k+1,j,Q)$$

where $k \in \{i, \ldots, j-1\}$, $P \in \mathcal{N}$, and $Q \in \mathcal{N}$. The reasoning for correctness is analogous.

Example: Running CYK with a simple PCFG

To illustrate the computation steps for CYK, let's consider a simple artificial PCFG. Consider the following example PCFG:

$\mathcal{N} = \{A, B\}$

$\Sigma = \{a, b, c\}$

$S = \{A\}$

$R =$

$A \rightarrow AB$	0.8
$A \rightarrow a$	0.2
$B \rightarrow BB$	0.7
$B \rightarrow b$	0.1
$B \rightarrow c$	0.2

Figure 23: The grammar rules of the example PCFG.

Now we will compute the probability that the string *abc* is generated by the given grammar:

Base Case:

$$\pi[1,1,A] = P(A \rightarrow a|A) = 0.2$$
$$\pi[1,1,B] = P(B \rightarrow a|B) = 0$$
$$\pi[2,2,A] = P(A \rightarrow b|A) = 0$$
$$\pi[2,2,B] = P(B \rightarrow b|A) = 0.1$$
$$\pi[3,3,A] = P(A \rightarrow c|A) = 0$$
$$\pi[3,3,B] = P(B \rightarrow c|A) = 0.2$$

Recursive Case:

$$\begin{aligned}\pi[1,2,A] =& P(A \rightarrow AA) \cdot P(A \rightarrow a|A) \cdot P(A \rightarrow b|A) \\ &+ P(A \rightarrow AB) \cdot P(A \rightarrow a|A) \cdot P(B \rightarrow b|B) \\ &+ P(A \rightarrow BA) \cdot P(B \rightarrow a|B) \cdot P(A \rightarrow b|A) \\ &+ P(A \rightarrow BB) \cdot P(B \rightarrow a|B) \cdot P(B \rightarrow b|B) \\ =& 0 + P(A \rightarrow AB) \cdot \pi[1,1,A] \cdot \pi[2,2,B] + 0 + 0 \\ =& 0.8 \cdot 0.2 \cdot 0.1 = 0.016\end{aligned}$$

Using similar calculations, we obtain:

$$\pi[1,2,B] = 0$$

$$\pi[2,3,A] = 0$$
$$\pi[2,3,B] = 0.014$$

Solution:

$$\begin{aligned}\pi[1,3,A] =& P(A \to AB) \cdot P(A \to a|A) \cdot P(B \to bc|B) \\ &+ P(A \to AB) \cdot P(A \to ab|A) \cdot P(B \to c|B) \\ =& P(A \to AB) \cdot \pi[1,1,A] \cdot \pi[2,3,B] + P(A \to AB) \cdot \pi[1,2,A] \cdot \pi[3,3,B] \\ =& (0.8 \cdot 0.2 \cdot 0.014) + (0.8 \cdot 0.016 \cdot 0.2) \\ =& \mathbf{0.0048}\end{aligned}$$

Some components of the sum are omitted as they must be 0 since A can only go to AB or a.

Weaknesses of PCFGs

PCFGs do not always provide accurate probability estimates for sentences. Two major weaknesses are:

1. Lack of sensitivity to lexical information
2. Lack of sensitivity to structural frequency

Lack of Sensitivity to Lexical Information

Consider the verb phrase: "drove down the street in the car". Below are two possible parse trees for this phrase:

a)

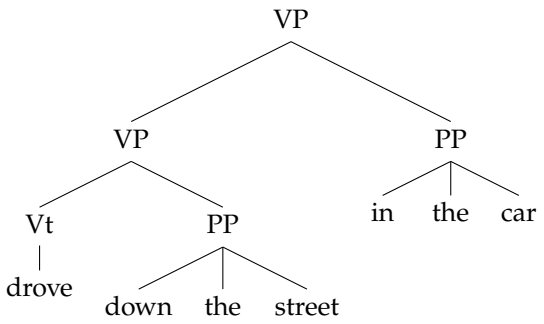

Figure 24: Parse tree a) for the verb phrase "drove down the street in the car". Here, the car is (logically) driving down the street.

A human can easily see that the phrase groupings in parse tree a) make much more sense than those in parse tree b). However, a PCFG considers each nonterminal expansion independently without taking into account the positions and semantics of other words and phrases in the sentence. Therefore, if $P(NP \to NP\ PP|NP) > P(VP \to VP\ PP|VP)$, then the second tree will have a higher probability according to the PCFG. The PCFG does not know that cars move on streets and streets should not be inside cars. This type of information is termed lexical information.

b)

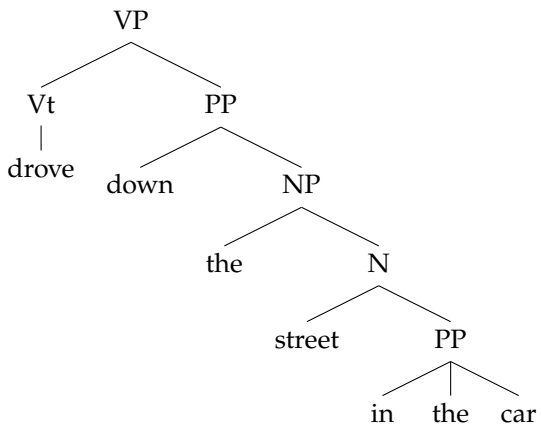

Figure 25: Parse tree b) for the verb phrase "drove down the street in the car". Here, the street is somehow inside the car.

Lack of Sensitivity to Structural Frequency

Another problem of PCFGs is the lack of sensitivity to structural frequency. Consider the following parse trees for the noun phrase "president of a company in Africa":

a)

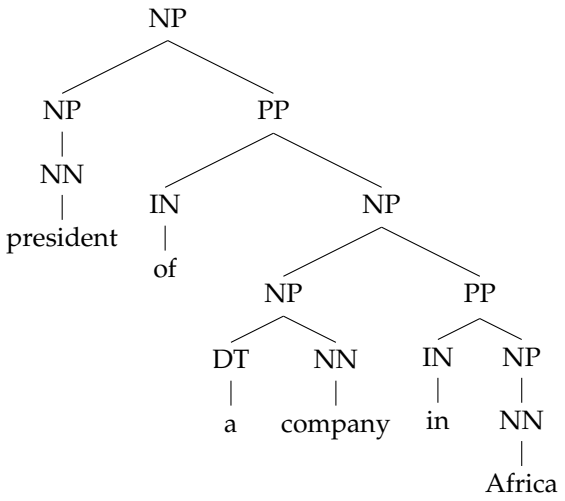

Figure 26: This parse tree has close attachment because "in Africa" describes the company and not the president.

b)

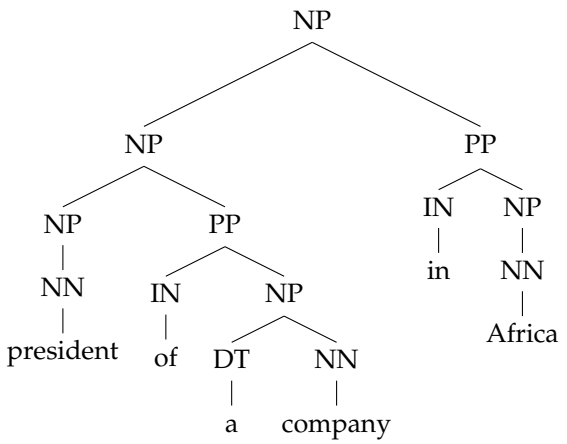

Figure 27: This parse tree does not have close attachment because "in Africa" describes the president and not the company.

As both parse trees use the same rules, they have the same probability under a PCFG. However, it has been shown that the "close attachment" interpretation occurs more often in Wall Street Journal (WSJ) texts. Therefore, the PCFG incorrectly assigns the same probabilities to both trees, even though tree a) should be assigned a higher probability. This error is also due to the PCFG's decisions to optimize at the edge-level rather than at the tree-level. In the next lecture, we will see how to construct better parsers that correct the errors made by PCFGs.

Rules
NP → NP PP
NP → NN
NP → DT NN
PP → IN NP
NN → president
NN → company
NN → Africa
IN → of
IN → in
DT → a

Figure 28: The rules used in trees a) and b). Both trees use the same rules. Therefore, they have the same probability under a PCFG.