Assignment 4
CSE 517: Natural Language Processing

University of Washington

Winter 2016
Due: March 2, 2016, 1:30 pm

1 HMMs and PCFGs

Here's the definition of a PCFG given in class on 2/17:

- A finite set of nonterminal symbols $\mathcal{N}$
  - A start symbol $S \in \mathcal{N}$
- A finite alphabet Σ, called "terminal" symbols, distinct from $\mathcal{N}$
- Production rule set $\mathcal{R}$, each of the form "$N \rightarrow \boldsymbol{\alpha}$" where
  - The lefthand side N is a nonterminal from $\mathcal{N}$
  - The righthand side $\boldsymbol{\alpha}$ is a sequence of zero or more terminals and/or nonterminals: $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$
    - Special case: **Chomsky normal form** constrains $\boldsymbol{\alpha}$ to be either a single terminal symbol or two nonterminals
- For each $N \in \mathcal{N}$, a probability distribution over the rules where N is the lefthand side, $p(* \mid N)$. We will use p_r to denote the conditional probability corresponding to rule $r \in \mathcal{R}$.

A PCFG defines the probability of a tree $\boldsymbol{t}$ as:

$$p(\boldsymbol{t}) = \prod_{r \in \mathcal{R}} p_r^{c_{\boldsymbol{t}}(r)} \tag{1}$$

where $c_{\boldsymbol{t}}(r)$ is the number of times rule r is used in the tree $\boldsymbol{t}$.

The definition of an HMM was given on 2/1; we spell it out more carefully here.

- A finite set of states $\mathcal{L}$ (often referred to as "labels")
  - A probability distribution over initial states, $\boldsymbol{\pi}$
  - One state denoted ⯃ is the special stopping state
- A finite set of observable symbols $\mathcal{V}$
- For each state $y \in \mathcal{L}$:

  - An "emission" probability distribution over $\mathcal{V}$, $\boldsymbol{\theta}_{*|y}$
  - A "transition" probability distribution over next states, $\boldsymbol{\gamma}_{*|y}$, including a final state

An HMM defines the probability distribution of a state sequence $\boldsymbol{y}$ and an emission sequence $\boldsymbol{x}$ (length n, excluding the stop symbol) as:

$$p(\boldsymbol{x}, \boldsymbol{y}) = \pi_{y_0} \prod_{i=1}^{n+1} \theta_{x_i|y_i} \cdot \gamma_{y_i|y_{i-1}} \tag{2}$$

Question 1: Demonstrate how to implement any HMM using a PCFG. Do this by showing formally how to define each element of the PCFG: $\mathcal{N}$, S, Σ, $\mathcal{R}$, and p. Hint: think of a state sequence as a right-branching tree; the nonterminals will correspond to HMM states/labels, and you will probably want to add a new nonterminal to serve as the start state S.

Question 2: Transform the grammar you defined above into Chomsky normal form. Every tree derived by the original grammar must have a corresponding tree in the new grammar, with the same probability. Explain how to transform the new PCFG's derivations back into derivations under the old PCFG. Hint: this might be easier if you add a special "start" word to the beginning of every sequence recognized by the new grammar, and you will likely need to add some new nonterminals, as well.

Question 3: Does it make sense to use the CKY algorithm to find the most probable state sequence for a sentence $\boldsymbol{x}$, under an HMM?

2 Probabilistic Finite-State Automata

On 2/17, we defined a finite-state automaton as:

- A finite set of states $\mathcal{S}$
  - Initial state $s_0 \in \mathcal{S}$
  - Final states $\mathcal{F} \subseteq \mathcal{S}$
- A finite alphabet Σ
- Transitions $\delta : \mathcal{S} \times \Sigma \rightarrow 2^{\mathcal{S}}$
  - Special case: **deterministic** FSA defines $\delta : \mathcal{S} \times \Sigma \rightarrow \mathcal{S}$

A string $\boldsymbol{x} \in \Sigma^n$ is recognizable by the FSA iff there is a sequence $\langle s_0, \ldots, s_n \rangle$ such that $s_n \in \mathcal{F}$ and

$$\bigwedge_{i=1}^{n} [[s_i \in \delta(s_{i-1}, x_i)]] \tag{3}$$

This is sometimes called a **path**.

As you might imagine, there are different ways to augment a finite-state automaton with probabilities; we'll consider two. The first one defines a joint distribution over paths (random variable $\boldsymbol{S}$ ranging over $\mathcal{S}^*$) and strings (random variable $\boldsymbol{X}$ ranging over Σ^*):

$$p(\boldsymbol{S} = \boldsymbol{s}, \boldsymbol{X} = \boldsymbol{x}) = \prod_{i=1}^{n} p(S_i = s_i, X_i = x_i \mid S_{i-1} = s_{i-1}) \tag{4}$$

The second one only defines a conditional distribution over paths given strings:

$$p(\boldsymbol{S} = \boldsymbol{s} \mid \boldsymbol{X} = \boldsymbol{x}) = \prod_{i=1}^{n} p(S_i = s_i \mid X_i = x_i, S_{i-1} = s_{i-1}) \tag{5}$$

Question 4: Describe an algorithm for efficiently solving

$$\operatorname*{argmax}_{\boldsymbol{s} \in \mathcal{S}^{n+1}} p(\boldsymbol{s} \mid \boldsymbol{x}) \tag{6}$$

where $\boldsymbol{x} \in \Sigma^n$.

Question 5: What are the asymptotic runtime and space requirements of your algorithm, expressed as a function of n and $|\mathcal{S}|$?

A more general sequence model might define:

$$p(\boldsymbol{S} = \boldsymbol{s} \mid \boldsymbol{X} = \boldsymbol{x}) = \prod_{i=1}^{n} p(S_i = s_i \mid \boldsymbol{X} = \boldsymbol{x}, S_{i-1} = s_{i-1}) \tag{7}$$

This is no longer a finite-state automaton, but rather a powerful sequence model for the label/state sequence that can condition on any part of the input ($\boldsymbol{x}$) at any position.

Question 6: What are some of the advantages and disadvantages of such an approach?

Question 7: List some ways to model the conditional distribution $p(S_i \mid \boldsymbol{X}, S_{i-1})$.

3 Labeling

Your friend, a linguist, has developed a new theory of syntactic categories. She wants to build a new treebank based on this theory, but she doesn't want to start from scratch. Instead, she proposes to use the sentences and unlabeled tree structures—sometimes called bracketings—from the Penn Treebank, and relabel each constituent based on her new theory.

After labeling a few hundred sentences' worth of trees, exhaustion sets in, and she asks whether you can build a classifier to help. You ask her about the criteria she uses to label each constituent, and you find that most of the decisions seem to depend on neighboring constituents, specifically, the parent, siblings, and children of a node in the tree. In essence, you suspect that a PCFG would be a good model of the labeling process.

Question 8: Describe a dynamic programming algorithm that takes as input a PCFG, a sentence, $\boldsymbol{x}$, and an unlabeled phrase-structure tree (denoted by a collection of spans $\langle\langle i_1, j_1\rangle, \langle i_2, j_2\rangle, \ldots, \rangle$), and produces the PCFG's most probable *labeled* tree consistent with the input. To simplify the problem a bit, assume that the PCFG is in Chomsky normal form and the every word's parent in the tree has a single child (i.e., that word), and that every other node of the tree has two children. The runtime of your algorithm should be linear in the size of the tree, and hence in the size of the sentence.

4 Parsing and Factor Graphs

The parsing algorithms we discussed in class are built on the idea of finding the optimal parse tree consistent with an input sentence under a statistical scoring function. Efficient optimization is possible when the score factors into local parts. This idea is closely related to factor graphs, which score assignments to (possibly many) random variables.

Question 9: Show how to represent dependency parsing using a factor graph. Assume that the scoring function is arc-factored (this idea was/will be explained in the 2/22 or 2/24 lecture). The random variables you need to account for will have the form $Y_{i \to j}$, taking the value $\emptyset$ if the tree does *not* make i the parent of j, and taking the value $\ell \in \mathcal{L}$ (the set of dependency labels) if i is the parent of j with label ℓ. For this question, define a factor graph (i.e., explain all the factors you need) that, given an assignment to $\boldsymbol{Y}$ that corresponds to a valid tree, returns the score of the tree.

Question 10: It is possible to use a factor graph to also assign a score of zero to assignments of $\boldsymbol{Y}$ that do not form a valid tree. To do this, you want to ensure that every node has one incoming arc, except the root node, which has zero, and further that the graph is connected. To encode these constraints in the factor graph, you need additional variables and factors; explain what they are. **This question is for extra credit.**

5 Transition-Based Parsing

An alternative to the optimization view of parsing is to think of the problem as a sequence of incremental decisions. This is often modeled as a sequence of *transitions* taken in an abstract state machine.[1] The most common of these transitions build the tree "bottom up." Here, we consider a way to build the tree "top down."

The algorithm maintains two data structures, a buffer B and a stack S. The buffer is initialized to contain the words in $\boldsymbol{x}$, with the left-most word at the top. The stack is initialized to be empty. At each iteration one of the following actions is taken:

- $\textsc{nt}(N)$ (for some $N \in \mathcal{N}$): introduces an "open" nonterminal and places it on the stack. This is represented by "(N" to show that the child nodes of this nonterminal token have not yet been fully constructed.

[1] A finite-state automaton is one kind of abstract state machine, but for parsing we keep track of more information.

- SHIFT removes the word at the front of the buffer and pushes it onto the stack.
- REDUCE repeatedly pops completed subtrees from the stack until an open nonterminal is enountered; this open nonterminal is popped and used as the label of a new constituent whose children are the popped subtrees. The new completed constituent is placed on the stack.

The goal of the algorithm is to reach a state where the buffer is empty and the stack contains a single tree, which will be a parse of the sentence. Figure 1 shows an example of the sequence of actions that might be taken in (fortunately, correctly!) parsing "*The hungry cat meows* ."

Input: *The hungry cat meows* .

	Stack	**Buffer**	**Action**
0		*The* | *hungry* | *cat* | *meows* | .	NT(S)
1	(S	*The* | *hungry* | *cat* | *meows* | .	NT(NP)
2	(S | (NP	*The* | *hungry* | *cat* | *meows* | .	SHIFT
3	(S | (NP | *The*	*hungry* | *cat* | *meows* | .	SHIFT
4	(S | (NP | *The* | *hungry*	*cat* | *meows* | .	SHIFT
5	(S | (NP | *The* | *hungry* | *cat*	*meows* | .	REDUCE
6	(S | (NP *The hungry cat*)	*meows* | .	NT(VP)
7	(S | (NP *The hungry cat*) | (VP	*meows* | .	SHIFT
8	(S | (NP *The hungry cat*) | (VP *meows*	.	REDUCE
9	(S | (NP *The hungry cat*) | (VP *meows*)	.	SHIFT
10	(S | (NP *The hungry cat*) | (VP *meows*) | .		REDUCE
11	(S (NP *The hungry cat*) (VP *meows*) .)		

Figure 1: Top-down transition-based parsing example. | is used to delimit elements of the stack and buffer. This example ignores part of speech symbols.

One way to use this abstract state machine is within a **search** framework. The states of the search space are exactly those of the abstract state machine—here, the buffer and stack. It should be clear that there is an infinite set of states that can be reached by taking different sequences of the actions above; only a very small set of them will lead to a desired final state (where the buffer is empty and the stack contains a single tree). At each iteration of the search, your method chooses a state it has already reached, considers different actions it might take at that state, each producing a new state, which is now remembered as one that has been reached. Different search strategies can be used to decide which state to explore next, and whether to discard any states you've reached.

An alternative to search is to proceed **greedily**, following one sequence of actions from the initial state until the termination criterion is reached. At each step, a statistical classifier is used to pick the best action. The classifier has access to the current state (the buffer and the stack) to make this decision. Linear-time parsing is achievable this way, and performance can be quite good, but it requires building a really good classifier.

Question 11: Show the sequences of actions needed to parse the ambiguous sentence "*He saw her duck.*" In one version, Alice is showing Bob her pet (a duck). In the other, Carla throws a rotten tomato at Alice, who ducks, and Bob saw the whole incident. For each version, construct a visualization like Figure 1 and show the stack and buffer at each iteration.

Question 12: What features might make sense in this classifier?

Question 13: Somewhat surprisingly, the approach above was described without mention of the grammar! Suppose you have a PCFG, and you want to apply top-down transition-based parsing to find a tree with high probability for the input sentence. How might you adapt the search version of this parser to try to find a high-probability tree?

Question 14: In greedy top-down transition-based parsing, a bad decision early on can really mess things up. One danger comes about when the grammar has left-recursive rules like $A \rightarrow A\ B$ or "cycles" of them, like $A \rightarrow B\ C$ and $B \rightarrow A\ D$. Can you think of a way to constrain such a parser so that it won't go into an infinite loop?