Prof. Ryan Cotterell

Assignment 4: Context-Free Parsing

The first half of the assignment is a series of theoretical questions about the material. When you have answered the questions to your satisfaction, you should upload a pdf file with your solutions (preferably written in LaTeX) to Moodle. The second half of the assignment is a series of coding questions. Please also upload your code to Moodle, adhering to all naming and formatting requirements as detailed in the question description.

1 Theory

Question 1: Calculating Prefix Probabilities (50 pts)

In this question, we will discuss how to use probabilistic context-free grammars (PCFGs) as language models. To start our journey, we have review what exactly a language model is. In class, we defined an alphabet Σ as a non-empty, finite set of symbols (words), and we said that a language model was *any* distribution over strings in Σ^*, the Kleene closure of Σ. We also discussed a simple strategy to define such a distribution: **local normalization**, which makes use of the chain rule of probability to factor the distribution into "local" chunks. However, there is a pesky little problem. For instance, if $\Sigma = \{a, b, c\}$, then you might think that probability of $p(aabc) = p(c \mid aab)p(b \mid aa)p(a \mid a)p(a \mid \text{BOS})$ i.e., a direct application of the chain rule of probability where $\text{BOS} \notin \Sigma$ is a distinguished beginning-of-sentence symbol.

Unfortunately, there is an ever so subtle bug in that factorization that stems from the fact that we are modeling *variable-order* sequence. For this reason, we introduce an end-of-sequence token EOS, which, itself, is *not* in Σ, but whose occurrence can be predicted (note that BOS is also not in Σ, and cannot be predicted). The correct decomposition is actually $p(aabc) = p(\text{EOS} \mid aabc)p(c \mid aab)p(b \mid aa)p(a \mid a)p(a \mid \text{BOS})$. The next two questions will help you build intuition about exactly why we need EOS.

a) **(5 pts)** In this question, we are going to see what can go wrong if you accidentally forget EOS. Let's assume an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_K\}$ of K symbols (words). Assuming a unigram model, i.e., where each word is predicted independently of context, we define the following EOS-less quantity

$$\widetilde{p}(w_1 \cdots w_M) \stackrel{\text{def}}{=} \prod_{m=1}^{M} p(w_m) \tag{1}$$

where $\sum_{w \in \Sigma} p(w) = 1$. Show that $\sum_{\mathbf{w} \in \Sigma^*} \widetilde{p}(\mathbf{w}) \to \infty$, i.e., it is not normalizable, and, thus, it is not a probability distribution.

b) **(5 pts)** Now show that including EOS (where $p(\text{EOS}) > 0$) remedies the situation. Define $p(w_1, \ldots, w_M) = p(\text{EOS})\widetilde{p}(w_1, \ldots, w_M)$ and show that now $\sum_{\mathbf{w} \in \Sigma^*} p(\mathbf{w}) = 1$. We now have that $\sum_{w \in \Sigma \cup \{\text{EOS}\}} p(w) = 1$ with EOS is included in the sum.

The above arguments can be extended to n-gram language models or even recurrent neural language models that look at an arbitrary amount of context.

Now that we discussed the necessity of having EOS, we are going to more closely inspect what it means if we omit it. We stated that the probability of a sequence is $p(aabc) = p(\text{EOS} \mid aabc)p(c \mid aab)p(b \mid aa)p(a \mid a)p(a \mid \text{BOS})$. However, the quantity $p(c \mid aab)p(b \mid aa)p(a \mid a)p(a \mid \text{BOS})$ is also useful in its own right despite *not* being the probability of the $aabc$. It is typically called the **prefix probabiliy** of a language model. More formally, we define

$$p_{\text{pre}}(\mathbf{w}) \stackrel{\text{def}}{=} \prod_{n=1}^{N} p(w_n \mid w_0 \cdots w_{n-1}) \tag{2}$$

where $w_0 \stackrel{\text{def}}{=} \text{BOS}$.

c) **(5 pts)** Let p be a locally normalized language model over the alphabet Σ, i.e., we have

$$p(\mathbf{w}) \stackrel{\text{def}}{=} p(\text{EOS} \mid \mathbf{w}) \prod_{n=1}^{N} p(w_n \mid w_0 \cdots w_{n-1}) \tag{3}$$

Show that

$$p_{\text{pre}}(\mathbf{w}) = \sum_{\mathbf{u} \in \Sigma^*} p(\mathbf{wu}) \tag{4}$$

From now on, we will *define* the prefix probability using Equation (4). To see why, ask yourself whether Equation (4) holds if the model is model is not locally normalized. Explain, intuitively, why this fact motivates calling p_{pre} a *prefix* probability.

In the terminology of the lecture, PCFGs are locally normalized generative models of text:

Definition 1.1. A PCFG is a five-tuple $\mathfrak{G} = (\mathcal{N}, \Sigma, \text{S}, \mathcal{R}, p)$, made up of:

- A finite set of non-terminal symbols $\mathcal{N}$;
- A finite alphabet of terminal symbols Σ;
- A distinguished start symbol S;
- A set of production rules $\mathcal{R} : \mathcal{N} \times (\mathcal{N} \cup \Sigma)^*$, i.e., it is of the form $\text{N} \to \boldsymbol{\alpha}$;
- A probability function $p : \mathcal{R} \to [0, 1]$.

Importantly, for a CFG to be *probabilisitc*, the rule probabilites must be locally normalized, meaning that for a series of rules with the same nonterminal X on the left-hand side, $\text{X} \to \boldsymbol{\alpha}_1, \ldots, \text{X} \to \boldsymbol{\alpha}_K$, the probabilities of the corresponding expansions of X must sum to one:

$$\sum_{k=1}^{K} p(\text{X} \to \boldsymbol{\alpha}_k) = 1 \tag{5}$$

Definition 1.2. A grammar is in **Chomsky normal form** (CNF), if the right-hand side of every production rule either consists of two non-terminals $\mathrm{Y}, \mathrm{Z} \in \mathcal{N}$, a terminal $a \in \Sigma$, or ε:

$$\mathrm{X} \to \mathrm{YZ} \tag{6}$$
$$\mathrm{X} \to a \tag{7}$$
$$\mathrm{S} \to \varepsilon \tag{8}$$

Note that $\mathrm{S} \to \varepsilon$ is the *only* nullary production and is often omitted in the context of parsing as we are rarely interested in parsing the empty string.

Definition 1.3. Given a non-terminal X and a string $\mathbf{w} = w_1 \cdots w_N \in \Sigma^*$, the **inside probability** between indices i and k (where $1 \leq i \leq k \leq N$) is defined as:

$$p_{\text{inside}}(w_i \cdots w_k \mid \mathrm{X}) \stackrel{\text{def}}{=} p(\mathrm{X} \stackrel{*}{\Rightarrow} w_i \cdots w_k) \tag{9}$$
$$= \sum_{\boldsymbol{t} \in \mathcal{T}_{\mathrm{X}}(w_i \cdots w_k)} p(\boldsymbol{t}) \tag{10}$$

where $\mathcal{T}_{\mathrm{X}}(w_i \cdots w_k)$ is the set of all derivation trees producing $w_i \cdots w_k$ from X.

d) **(6 pts)** Given a PCFG in CNF, explain how to compute the probability of a sentence $p(\mathbf{w})$ under the PCFG using CKY.

e) **(4 pts)** Next we define the prefix probability of a string:

Definition 1.4. Given a context-free grammar $\mathfrak{G}$ and given a string $\mathbf{w} \in \Sigma^*$, we define the **prefix probability** for string $\mathbf{w}$ being a prefix under $\mathfrak{G}$ to be:

$$p_{\text{pre}}(\mathbf{w}) = \sum_{\mathbf{u} \in \Sigma^*} p(\mathbf{wu}) \tag{11}$$

Show:

$$p_{\text{pre}}(\mathbf{w}) = \sum_{\mathbf{u} \in \Sigma^*} p(\mathbf{wu}) = p(\mathrm{S} \stackrel{*}{\Rightarrow} \mathbf{wv}) \tag{12}$$

i.e. the probability of producing string $\mathbf{wv}$ with an arbitrary suffix $\mathbf{v} \in \Sigma^*$.

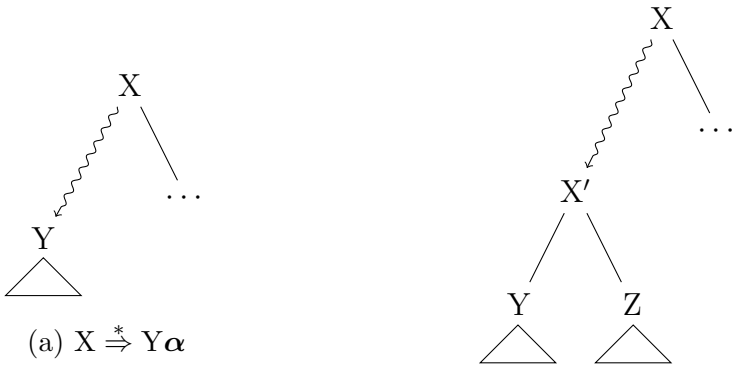
In the remainder of this assignment, we will use Equation (12) as the *definition* of the prefix probability $p_{\text{pre}}(\mathbf{w})$.

f) **(5 pts)** Our goal is to devise an efficient algorithm for computing the prefix probability under a PCFG in CNF. To do so, we need to introduce the notion of the left-corner probability:

Definition 1.5. For non-terminals $\mathrm{X}, \mathrm{Y}, \mathrm{Z} \in \mathcal{N}$, the **left-corner** probabilities $p_{\text{lc}}(\mathrm{Y} \mid \mathrm{X})$ and $p_{\text{lc}}(\mathrm{Y\,Z} \mid \mathrm{X})$ are respectively defined as:

$$p_{\text{lc}}(\mathrm{Y} \mid \mathrm{X}) \stackrel{\text{def}}{=} p(\mathrm{X} \stackrel{*}{\Rightarrow} \mathrm{Y}\boldsymbol{\alpha}) \tag{13}$$
$$p_{\text{lc}}(\mathrm{Y\,Z} \mid \mathrm{X}) \stackrel{\text{def}}{=} p(\mathrm{X} \stackrel{*}{\Rightarrow} \mathrm{Y\,Z}\boldsymbol{\alpha}) \tag{14}$$

Intuitively, the left-corner probability is the probability of reaching Y along the left corner of a tree rooted at X.



(a) $\mathrm{X} \overset{*}{\Rightarrow} \mathrm{Y}\boldsymbol{\alpha}$

(b) $\mathrm{X} \overset{*}{\Rightarrow} \mathrm{Y}\,\mathrm{Z}\boldsymbol{\alpha}$

Figure 1: Visualization of left-corner derivations

Your task is to devise an algorithm for the computation of the left-corner probabilities for all pairs of non-terminals $\mathrm{X}, \mathrm{Y} \in \mathcal{N}$. Analyze the runtime of your algorithm. You should devise an algorithm to compute both $p_{\text{lc}}(\mathrm{Y} \mid \mathrm{X})$ and $p_{\text{lc}}(\mathrm{Y}\,\mathrm{Z} \mid \mathrm{X})$. Inspecting Figure 1b will be useful to understand the relation between the two. **Hint:** The simplest way to do this is to construct a matrix that maps each pair of non-terminals X, Y to the probability $p(\mathrm{X} \to \mathrm{Y}\boldsymbol{\alpha})$, i.e., the probability of getting Y from X after one step of the derivation, and then finding the Kleene closure over this matrix.

Note: For the remainder of the assignment, you can assume the grammar is in CNF.

g) **(6 pts)** Prove the following identity

$$p_{\text{pre}}(w_i \cdots w_k \mid \mathrm{X}) = \sum_{j=i}^{k-1} \sum_{\mathrm{Y},\mathrm{Z} \in \mathcal{N}} p_{\text{lc}}(\mathrm{Y}\,\mathrm{Z} \mid \mathrm{X}) \cdot p_{\text{inside}}(w_i \cdots w_j \mid \mathrm{Y}) \cdot p_{\text{pre}}(w_{j+1} \cdots w_k \mid \mathrm{Z}) \tag{15}$$

where $i < k$.

h) **(5 pts)** Explain how, using Equation (15), and by pre-computing $p_{\text{lc}}(\mathrm{Y}\,\mathrm{Z} \mid \mathrm{X})$ and $p_{\text{pre}}(w_k \mid \mathrm{X})$ using the solution from part 1f), the time complexity of computing $p_{\text{pre}}(\mathbf{w})$ is $\mathcal{O}\left(N^3|\mathcal{N}|^3 + |\mathcal{N}|^4\right)$. Conclude that you can compute *all* the prefix probabilities $p_{\text{pre}}(w_1)$, $p_{\text{pre}}(w_1 w_2)$, …, $p_{\text{pre}}(w_1 \cdots w_N)$ in $\mathcal{O}\left(N^4|\mathcal{N}|^3 + N|\mathcal{N}|^4\right)$ time by running the above algorithm once per prefix.

i) **(6 pts)** Develop a CKY-like procedure to compute all the prefix probabilities $p_{\text{pre}}(w_1)$, $p_{\text{pre}}(w_1 w_2), \ldots, p_{\text{pre}}(w_1 \cdots w_N)$ jointly that runs in $\mathcal{O}\left(N^3|\mathcal{N}|^3 + |\mathcal{N}|^4\right)$, an improvement over the naïve $\mathcal{O}\left(N^4|\mathcal{N}|^3 + N|\mathcal{N}|^4\right)$-time algorithm. **Hint:** This simply involves additional memoization.

j) **(3 pts)** Finally, for a given string $\mathbf{w} = w_1 \cdots w_N \in \Sigma^*$, rewrite the conditional probability $p(w_N \mid w_1 \cdots w_{N-1})$ as a ratio of prefix probabilities, that is in terms of expressions of the form $p(\mathrm{S} \overset{*}{\Rightarrow} \mathbf{u})$ for some string $\mathbf{u} \in \Sigma^*$. Explain why this is a useful quantity for language generation.

2 Practice

Question 2: Implementing a Neural Constituency Parser (50 pts)

In this question, you will implement and run a modern constituency parser on real-world data in order to find the best parse of a given string. To do this, we want to model the conditional probability $p(\mathbf{t} \mid \mathbf{s})$, where $\mathbf{t}$ is a parse tree and $\mathbf{s}$ is a given sentence.

We will be using the Penn Tree Bank, a corpus of news texts that were manually annotated with POS tags and parse trees.

The notebook for this question is on Google Colab: https://colab.research.google.com/drive/1GUHilp_oiVrCL71EHHtRcJEvpcos4in8, which also contains more detailed instructions. This question is subdivided into the following parts:

a) First, you will download the data and familiarize yourself with it, implementing and running some preprocessing so it can be used to train the desired model.

   i) **(5 pts)** We want to remove -NONE-, -LRB- and -RCB- tags. For the sake of simplicity, we will remove any tree that contains these tags. Write a method that returns True if a tree passed to it contains such tags. In a comment (without implementing the code), give two other ways -NONE- tags could be handled more data efficiently without affecting the grammaticality of the sentence.

   ii) **(5 pts)** Write a method that takes in tags and only keeps the part before the first hyphen if it is hyphenated, e.g. NP-SBJ should become NP. Then write a method that traverses a tree and updates its tags in place.

   iii) **(5 pts)** To remove unary derivations and convert the training trees to binary trees, we will use the functions `collapse_unary` and `chomsky_normal_form` in nltk.tree. Write down the call you will make using horizontal Markov smoothing of 1 and vertical Markov smoothing of 0, factoring right, and using '∧' for parent and '|' for child. Explain the meaning and the purpose of these parameters in a comment.

   iv) **(3 pts)** Bringing all of these together, write a method that takes a set of trees and returns a list of (copied) cleaned trees.

b) Now we want to train our model to parse from scratch, by providing it with training examples and optimizing it. We will use a bidirectional LSTM to predict label scores and a version of CKY to find the best parse tree.

   i) **(5 pts)** First we need to convert the words and tags into embeddings that can be fed to the LSTM. Implement the following method. Unknown words should get the embedding of the UNK word.
   Hint: use the reverse index methods `tag_vocab.index()` and `word_vocab.index()` methods from `vocabulary.py` to find the keys of words and tags in the respective embedding dicts created above.

   ii) **(12 pts)** Now we can implement the parser method. It works like CKY, with some additions. During training, the gold parameter is set, representing the reference parse tree. During inference, this is not set and the method simply returns the best-scoring parse tree. Your task is to implement the CKY algorithm, iterating through spans and for each pair of span indices dynamically calculate scores and parse trees:

1) Use the method `get_label_scores` to get the scores for each label corresponding to the given span (left, right).
2) Find the best place to split a given span into two subspans given their respective label scores. (If `force_gold` is set, use the method `tree.oracle_splits` from `trees.py` to get the split positions, and use the first of them as the index to split the tree.)
3) Calculate the score of the label when splitting at the best index found above and put it into the scores `chart`.
4) Get the subtrees for the span and split position, and put the resulting new subtree into the `parse_trees` dict.

Hint: the label scores are stored as *dynet* expressions, use `.value()` to get the score value.

c) Finally, you train and evaluate the model, using a hinge loss function to judge the similarity of examples with ground truth.

i) **(8 pts)** Your first task is to implement a method that calculates the loss for a predicted parse tree given a sentence. Do this by first calculating the score and predicted parse tree, then calculating the score of a given gold parse tree. You should implement hinge loss where the loss is 0 if the predicted tree was correct and otherwise the difference between the score of the predicted tree and that of the reference tree.

ii) **(5 pts)** Now run training for 10 epochs, reporting the batch loss for each batch. In the training script, fill in the part that iterates through the training trees.

iii) **(2 pts)** Run the evaluation script and report precision, recall, and F1 score.