Prof. Ryan Cotterell

Course Assignment Episode 2

10/07/2021 - 11:46h

Question 1: CFG Refinement (8 pts)

In this question, you are given a simple training dataset (Figure 1) which consists of 4 sentences $\mathbf{s}_1, \ldots, \mathbf{s}_4$ together with their parse trees $\mathbf{t}_1, \ldots \mathbf{t}_4$.

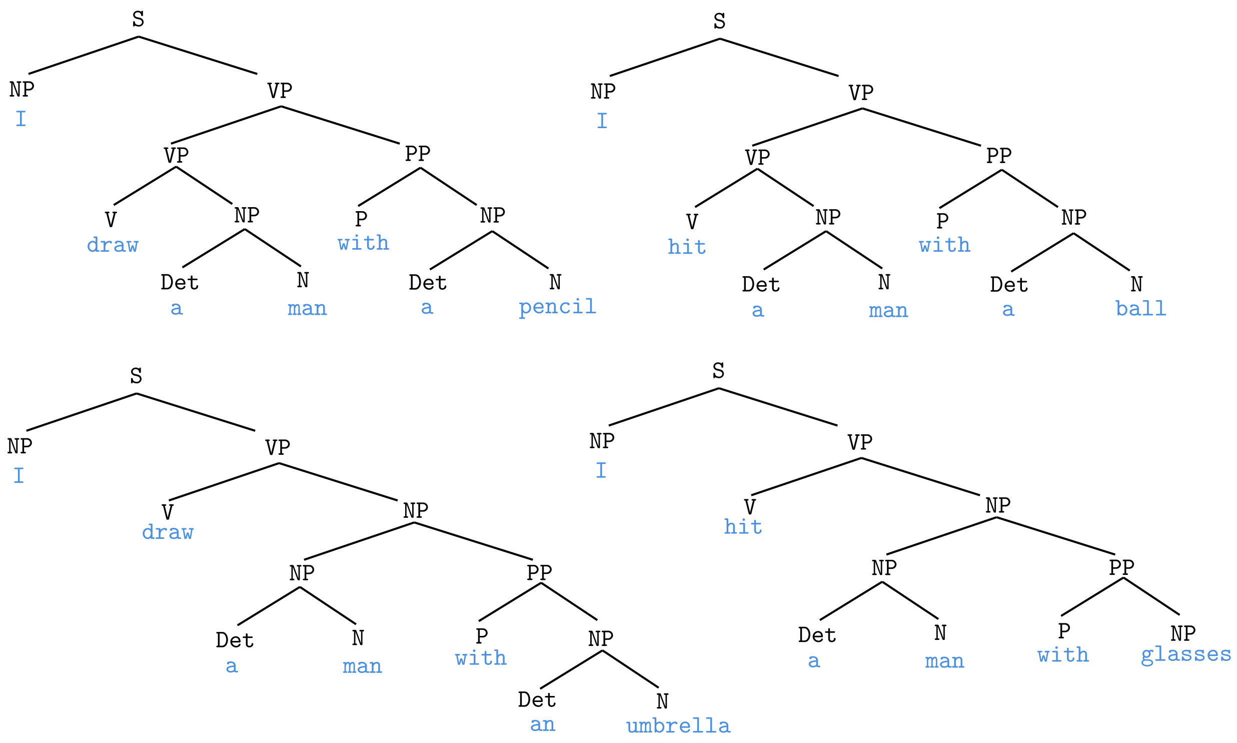

Figure 1: Training Dataset

(a) Use the training dataset to define the corresponding context free grammar, where a production rule is added to the CFG if it is used in the parse tree and the alphabet of the grammar consists of set of the words in $\mathbf{s}_1, \ldots \mathbf{s}_4$.

(b) Learn probabilities for the production rules you defined in the previous step from the training set to create a PCFG. You should use a simple counting approach where the probability of a rule $\mathrm{N}_1 \to \alpha^*$ for all $\mathrm{N}_1 \in \mathcal{N}$ is:

$$\frac{\mathrm{count}(\mathrm{N}_1 \to \alpha^*)}{\sum_{\mathrm{N}_1 \to \alpha \in \mathcal{R}} \mathrm{count}(\mathrm{N}_1 \to \alpha)}$$

and $\mathrm{count}(\mathrm{N}_1 \to \alpha)$ is the number of times that rule is used in the training set.

(c) By looking at the dataset, we might see that subject NP expansions are very different from object NP expansions in a sense that expanding NP $\rightarrow$ NP PP is more frequent when the noun phrase is the object of the sentence as in “a man with an umbrella” or “a man with glasses”. However, in situations where the noun phrase is the subject of the sentence such as “I” it is less likely to expand that noun phrase to NP and PP. How can you change the production rules and the set of non-terminals used in part (a) in a way that a simple counting approach captures these differences? Please provide the PCFG along with probabilities.

(d) For each constituent, we define the **head** of that constituent as the word that is “most useful” for determining how that constituent is integrated into the rest of the sentence. In order to find the head words, a recursive approach will be used:

- For any non-terminal rule, the head of the left-hand side must be the head of **one** of the children. To select which one, we use these deterministic rules (the child which determines the head is annotated with a plus):

$$
\begin{aligned}
&\text{S} \rightarrow \text{NP VP}^+\\
&\text{NP} \rightarrow \text{Det N}^+\\
&\text{NP} \rightarrow \text{NP}^+ \text{ PP}\\
&\text{VP} \rightarrow \text{V}^+ \text{ NP}\\
&\text{VP} \rightarrow \text{VP}^+ \text{ PP}\\
&\text{PP} \rightarrow \text{P NP}^+
\end{aligned}
$$

- The head of a terminal rule will be the terminal itself. For example, the head of N $\rightarrow$ man is “man”.

Using the head words can help resolving some ambiguities. As we see in the dataset, it is more probable to attach the prepositional phrase “with glasses” to “a man” than to the verb “hit”. Incorporate the head tokens defined by the rules above into your production rules. Show how the parse tree for the sentence “I hit a man with glasses” will change with the lexicalized rules. You don’t need to provide the complete set of rules and their probabilities. You should use the basic CFG from part (a) and not the modified version in part (c) for this question.
Hint: You may lookup *lexicalized context-free grammars.*

(e) How does the modification in part (c) and (d) affect the runtime of the CKY algorithm? Please provide the runtime and space complexity of the algorithm in Big-O notation.

Question 2: Parsing Projective Dependency Trees* (6 pts)

In the previous question, you considered *lexicalized context-free grammars* as an extension of regular context-free grammars. In this one, you will try to connect them to the problem of dependency parsing. Observe that, by determining the head word in each production, we can infer the dependency structure of the sentence; however, notice that we can only model *projective* trees since the spans of the nonterminals of a production never overlap!

* Problem adapted from Jacob Eisenstein’s Introduction to Natural Language Processing textbook.

We will now examine the relationship between lexicalized context-free grammars and dependency parsing more concretely. We use the notation from the exercises and the lectures. ψ denotes the arc-factored scoring function which we use to define the score of each possible dependency tree.

Suppose you have a set of unlabeled arc scores $\psi(i \to j)_{i,j=1}^{M} \cup \psi(ROOT \to j)_{j=1}^{M}$.

(a) Assuming each word type (i.e. each vocabulary entry) occurs no more than *once* in the input ($i \neq j \implies w_i \neq w_j$), how would you construct a weighted lexicalized context-free grammar so that the score of *any* projective dependency tree is equal to the score of some equivalent derivation in the lexicalized context-free grammar?

(b) Verify that your method works for the example *They fish*.

Question 3: Semantic Representations (Not Graded)

Let $G = (V, \Sigma, \mathrm{S}, R)$ be a context-free grammar such that:

- $V = \{\mathrm{S}, \mathrm{NP}, \mathrm{V}_t, \mathrm{VP}\}$ is the set of non-terminal symbols.
- $\Sigma = \{everyone, loves, someone, Alex\}$ is the set of terminal symbols.
- S is the distinguished start symbol.
- The set of production rules R of the grammar G is given by:
  - $\mathrm{S} \to \mathrm{NP}\ \mathrm{VP}$.
  - $\mathrm{VP} \to \mathrm{V}_t\ \mathrm{NP}$.
  - $\mathrm{NP} \to$ *everyone*.
  - $\mathrm{NP} \to$ *someone*.
  - $\mathrm{NP} \to$ *Alex*.
  - $\mathrm{V}_t \to$ *loves*.

  Notice that the context-free grammar G is in Chomsky normal form.

(a) List all the sentences that can be generated by the grammar G.

We will now use the grammar G to apply semantic analysis. We will adopt the convention that if a production rule $A \to BC$ is used, then the semantic representation of the constituent corresponding to A is constructed from those of B and C as follows:

$$A.\mathrm{sem} \ = \ (B.\mathrm{sem} \quad C.\mathrm{sem}), \tag{1}$$

i.e., we always apply the semantic representation of the left non-terminal symbol (B in the above example) to that of the right non-terminal symbol (C in the above example).

The semantic representations of the terminal symbols *everyone*, *someone* and *Alex* are fixed to be:

- "*everyone*".sem = $\lambda P.\forall x.(\mathrm{PERSON}(x) \Rightarrow P(x))$.
- "*someone.*"sem = $\lambda P.\exists y.(\mathrm{PERSON}(y) \wedge P(y))$.
- "*Alex*".sem = $\lambda P.P(\mathrm{ALEX})$.

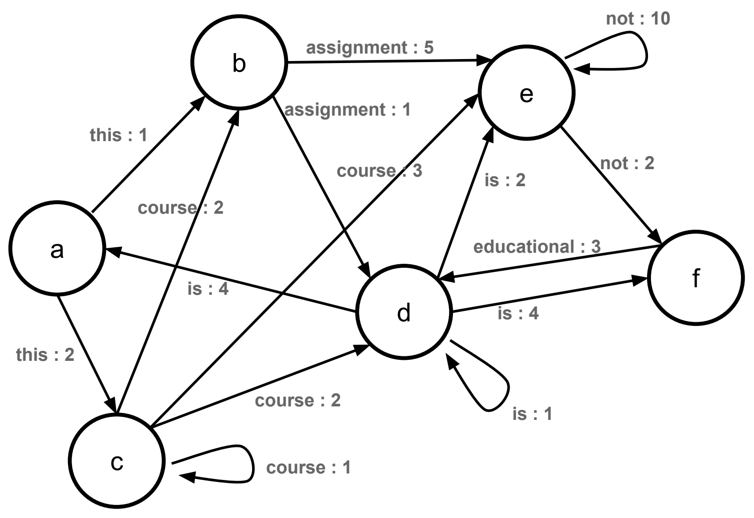

Figure 2: Graph notation for Weighted Finite State Acceptor (WFSA)

(b) Give a semantic representation for the terminal symbol *loves* in such a way that we get

$$\text{“}\textit{Alex loves Alex}\text{”}.\text{sem} = \text{LOVES}(\text{ALEX}, \text{ALEX}),$$

$$\text{“}\textit{everyone loves Alex}\text{”}.\text{sem} = \forall x.(\text{PERSON}(x) \Rightarrow \text{LOVES}(x, \text{ALEX})),$$

and

$$\text{“}\textit{Alex loves someone}\text{”}.\text{sem} = \exists y.(\text{PERSON}(y) \wedge \text{LOVES}(\text{ALEX}, y)).$$

(c) Using the semantic representation of *loves* that you obtained in Part (b), derive the semantic representation of the sentence *“everyone loves someone”*.

Question 4: Floyd-Warshall WFSA (6 pts)

We are given a vocabulary of tokens $\mathcal{V} = \{\text{this}, \text{is}, \text{not}, \text{educational}, \text{course}, \text{assignment}\}$. The number of tokens is $M = |\mathcal{V}|$. Any token sequence $\mathbf{y}$ (i.e., a string) that can be sourced from this vocabulary is contained within the Kleene closure $\mathcal{Y} = \mathcal{V}^*$. Throughout this task, we only consider a subset $\mathcal{Y}_{\geq 2, \leq 6} = \mathcal{V}^2 \cup \mathcal{V}^3 \cup \ldots \cup \mathcal{V}^6$, which contains token sequences of minimum length 2 and maximum length of $N = 6$.
Figure 2 shows a given Weighted Finite State Acceptor (WFSA). In a WFSA, we only have inputs and the integer value after the “:” symbol is the weight of the state transition. All states of this WFSA (a,b,c,d,e,f) can be starting and ending (accepting) states.

(a) Table 1 gives a sample of strings from $\mathcal{Y}_{\geq 2, \leq 6}$. Which strings are accepted under the WFSA? For accepted strings, please write down a “YES”, for not accepted, write down “NO” in the column **accepted**. For accepted strings, also write down their associated weight assigned by the WFSA in the column **weight** (note: if there is more than one accepting path for a given string, then the string with the lowest weight among all accepting paths is selected.)

(b) We are now regarding the WFSA in figure 2 as a “shortest-path / lowest-weight path problem” to find strings with the lowest weights, given a start and end token taken from $\mathcal{V}$. Put more explicitly, we are interested in “minimising” instead of “maximising”

number	sample strings	accepted	weight
1	educational is this not		
2	is this assignment educational		
3	not educational is not educational		
4	this assignment is not educational		
5	is this assignment educational		
6	this assignment course is educational		
7	is this assignment not educational		
8	this assignment not		
9	this course assignment is not educational		
10	this course is not not educational		
11	not educational is this		
12	course assignment is not educational		
13	not this assignment is educational		
14	not not not educational		
14	is this course assignment not educational		
15	course assignment is this		
16	this course is interesting		
17	this course assignment not educational		

Table 1: Some strings from $\mathcal{Y}_{\geq 2, \leq 6}$

the weight of a path. Note that the maximum string length is still $N = 6$. Execute the Floyd-Warshall algorithm with an additional matrix that allows for backtracking the "shortest path" between any two states at any iteration. Initialize the matrices in iteration $n = 0$, and then apply iterations of the Floyd-Warshall algorithm for $n \geq 1$. Do so by filling in the charts in figure 3 and 4 (note: you may not need all charts). The left column matrix should contain all weights after iteration n, the right column matrix should contain all information needed to backtrack the path (you may only store one token per matrix entry). You should find the lowest-weight path from any node to any other node, including itself. A state can directly repeat itself only if a self-loop is given, which is true for state c,d,e.

Note – we made an adjustment to this subtask: in this subtask, you have to slightly modify the Floyd-Warshall algorithm in that you should not assume self-loops if they are not explicitly given in figure 2. For instance, state "b" cannot repeat itself. Thus, you should not initialise the diagonal with all zeroes in $n = 0$. We would please like to apologise for this belayed specification – **if you already completed this subtask and initialised the diagonal with zeroes as in the original Floyd-Warshall, we will not deduct any points. This course is running only for the second time, hence mistakes may still occur occasionally.**

(c) Is the number of iteration of your algorithm bound and what necessary condition needs to be met? After which iteration ($n =$?) does the algorithm terminate?

(d) What is the time and space complexity for executing the Floyd-Warshall algorithm with an additional second matrix for path backtracking (for time complexity, giving the order is sufficient)? After Floyd-Warshall has terminated, what is the maximum time complexity for backtracking the "lowest-weight" path between any two nodes (giving the order is sufficient)?

iteration: n = 0, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 0, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 1, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 1, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 2, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 2, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 3, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 3, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

Figure 3: Floyd-Warshall algorithm, iteration 0 to 3; left column matrix should contain weights after iteration n; right column matrix should be iteratively filled for backtracking each path

iteration: n = 4, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 4, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 5, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 5, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 6, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 6, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 7, weight matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

iteration: n = 7, backtracking matrix

	a this	b assign-ment	c course	d is	e not	f edu-cational
a this						
b assignment						
c course						
d is						
e not						
f educational						

Figure 4: Floyd-Warshall algorithm, iteration 4 to 7; left column matrix should contain weights after iteration n; right column matrix should be iteratively filled for backtracking each path