Assignment 8

CSE 447 and 517: Natural Language Processing - University of Washington

Winter 2022

Please consult the course website for current information on the due date, the late policy, and any data you need to download for this assignment.

This assignment is designed to advance your understanding of structured problems in NLP that deal with segmentation, trees, and logical forms.

★ **problems are for CSE 517 students only.** Other problems should be completed by everyone.

Submit: You will submit your writeup (a pdf) via Gradescope. Instructions can be found here.

Data: The data you need for this assignment is available at `https://nasmith.github.io/NLP-winter22/assets/data/A8.tgz`.

1 Byte-Pair Encoding

The rare/unknown word issue is ubiquitous in neural text generation. Typically, a relatively small vocabulary is constructed from the training data, and the words not appearing in the vocabulary are replaced with a special `<unk>` symbol.

Byte-pair encoding (BPE) is currently a popular technique to deal with this issue. The idea is to encode text using a set of automatically constructed **types**, instead of conventional word types. A type can be a character or a subword unit; and the types are built through a iterative process, which we now walk you through.

Suppose we have the following tiny training data:

`it unit unites`

The first step is to append, to each word, a special `<s>` symbol marking the end of a word. Our initial types are the characters and the special end-of-word symbol: {`i`, `t`, `u`, `n`, `e`, `s`, `<s>`}. Using the initial type set, the encoding of the training data is

`i t <s> u n i t <s> u n i t e s <s>`

In each training iteration, the most frequent type bigram is merged into a new symbol and then added to the type vocabulary. Ties can be broken at random.

Iteration 1

Bigram to merge (with frequency 3): `i t`

Updated data: `it <s> u n it <s> u n it e s <s>`

Iteration 2

Bigram to merge (with frequency 2): `it <s>`

Updated data: `it<s> u n it<s> u n it e s <s>`

Iteration 3

Bigram to merge (with frequency 2): `u n`

Updated data: `it<s> un it<s> un it e s <s>`

In this example, we end up with the type vocabulary {`i`, `t`, `u`, `n`, `e`, `s`, `<s>`, `it`, `it<s>`, `un`}. The stopping criterion can be defined by setting a target vocabulary size.

Applying BPE to new words is done by first splitting the word into characters, and then iteratively applying the merge rules in the same order as in training. Suppose we want to encode text `unite itunes`, it is first split into `u n i t e <s> i t u n e s <s>`. Then

Iteration 1

Bigram to merge: `i t`

Encoded word: `u n it e <s> it u n e s <s>`

Iteration 2

Bigram to merge: `u n`

Encoded word: `un it e <s> it un e s <s>`

The above procedure is repeated until no merge rule can be applied. In this example we get the encoding `un it e <s> it un e s <s>`.

1. Is it necessary that we append a special symbol `<s>` at the end of each token? Why or why not?

2. Given the training data, is it true that only one BPE encoding can be learned? Why or why not?

3. Implement the BPE algorithm and run it on the text data in the tarball linked above. What we want you to produce is a scatterplot showing points (x, y), each corresponding to an iteration of the algorithm, with x the current size of the type vocabulary, and y the length of the training corpus (in tokens) under that vocabulary's types. Run the algorithm until the frequency of the most frequent type bigram is one. How many types do you end up with? What is the length of the training data under your final type vocabulary?

4. In the above running example, neither `unite` or `itunes` appears in our training data. We nevertheless managed to encode them meaningfully using BPE types, which would otherwise be `<unk>` if we were using conventional word types. Think of a rare English word that does not appear in the training data used in Question 3. Encode it using the BPE encoding you just trained. Do you get an `<unk>`? More generally, justify why it is less likely that we encounter an `<unk>` using the BPE encoding.

5. Another way of dealing with the rare word issue is to encode the text using characters. Comparing it to BPE, discuss the advantages and potential issues of character-level encoding.

Useful resources. BPE has an open-source implementation: `https://github.com/rsennrich/subword-nmt`. You can use it to check your implementation or in your own research. Please refrain from copy-pasting code for this assignment.

2 First-Order Logic – Eisenstein 12.2 (p. 286)

Convert the following examples into first-order logic, using the relations CAN-SLEEP, MAKES-NOISE, and BROTHER.

1. If Abigail makes noise, no one can sleep.
2. If Abigail makes noise, someone cannot slep.
3. None of Abigail's brothers can sleep.
4. If one of Abigail's brothers makes noise, Abigail cannot sleep.

3 Extra Credit: Tree 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.

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 that 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 ★ Transition-Based Parsing

An alternative to the optimization view of parsing (exemplified by the CKY algorithm we covered in lecture) 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:

- 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.
- SHIFT removes the word at the front of the buffer and pushes it onto the stack.

[1] A finite-state automaton is one kind of abstract state machine that "remembers" only one of a finite set of states, but for parsing we keep track of more information.

- 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.

1. 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.

2. What features might make sense in this classifier?

3. 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?

4. 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 $B \rightarrow A\ D$. Describe a way to constrain such a parser so that it won't go into an infinite loop.