Assignment 3

Due: Saturday 11/02/2024 @ 11:59pm EST

Disclaimer

I encourage you to work together, I am a firm believer that we are at our best (and learn better) when we communicate with our peers. Perspective is incredibly important when it comes to solving problems, and sometimes it takes talking to other humans (or rubber ducks in the case of programmers) to gain a perspective we normally would not be able to achieve on our own. The only thing I ask is that you report who you work with: this is **not** to punish anyone, but instead will help me figure out what topics I need to spend extra time on/who to help.

Setup: The Data

In this assignment you will build and improve a statistical parser from the ATIS (Air Traffic Information System) portion of the Penn Treebank. This portion consists of short queries and commands spoken by users of a fake robot travel agency. To do this, we will need the following data files (located in the `data`) directory.

- `train.trees`: This file contains our training trees. Each line in this file contains a single tree, and brackets are used to specify the tree structure. Capitalized words are nonterminals, and lower-case words are terminals.
- `dev.strings`: This file contains raw sentences (not trees) that will be our development data inputs. Your model will convert these sentences into trees.
- `dev.trees`: Ground-truth trees for each sentence in the development data.
- `test.strings`: This file contains raw sentences (not trees) that will be our test data inputs. Your model will convert these sentences into trees.
- `test.trees`: Ground-truth trees for each sentence in the test data.

Section: Starter Code

In addition to the `data` directory, there are quite a few code files included with this pdf. Some of these files are organized into sub-directories. The files and directories are:

- `eval/`: This directory contains all code files used to evaluate a model's performance. In this assignment we will use one flavor of performance: **f1** score:
  - `eval/evalb.py`: This file will be used to evaluate the trees produced by your model. It calculates how many "brackets" (i.e. subtrees) match between your trees and the ground truth.
- `preprocess.py`: This file contains code that will be used to convert our training data into trees that obey chomsky-normal form.
- `postprocess.py`: This file contains code that will be used to undo the preprocessing made by `preprocessing.py`.

- `unknown.py`: This file contains code that will count the number of times a terminal symbol is used in the dataset, and convert any terminal symbol that has a count of 1 to the special unknown symbol.

- `trees.py`: This file contains a tree library. A majority of this code is necessary for converting/unconverting to/from chomsky normal form. A `Tree` contains a `Node` root, where each `Node` has a label (either a nonterminal or a terminal symbol), a parent `Node`, and a list of children `Nodes`. There are also some utility functions to create a tree from a string, etc. The `__str__` and `__repr__` methods have been overwritten for custom `Tree` serialization. A `Tree` is serialized into the bracket form that the Penn Treebank uses.

- `[part1/part2].py`: These files correspond to part1 and part2 of this assignment. I have implemented some of it for you so you can get the feel of how using the starter code should go. I also implemented these parts to provide more structure to the assignment. Some of the functions in these files I have left up to you. Even though there is a part 3 of this assignment, I have not provided you starter code for it. See the instructions below for more details.

- `models/`: This directory contains the meat of this assignment. It contains two files, only one of which you will need to complete:

  - `models/grammar.py`: This file contains an implementation of a Probabilistic Context Free Grammar (a `PCFG` object). A grammar in this assignment must be in chomsky normal form. I have implemented it in a similar manner as how I implemented a `fst` from the previous assignment. Suffice to say that there is a field called `productions_from` which is a dictionary of dictionaries. The first dictionary is key'd by a nonterminal symbol, and the value is a dictionary which itself is key'd by a `Production` object. The value of the second dictionary is the weight of that production.

    Even though this class has some fields that look similar to a `fst`, it is possible to do this assignment without referencing a single field in the `PCFG` class directly.

  - `models/parser.py`: This file contains a partial implementation of a statistical parser. There are a few methods in this class that you will need to complete. See below for more details, as well as see the methods themselves. Each method you need to complete is marked with a `TODO`.

Preprocessing

Before you start fiddling with the implementation, you need to preprocess the Penn Treebank. Please complete the following steps:

1. Create a directory called `generated` in the top-level directory of your workspace. This directory should appear in the same level as the `data` directory.
2. Run `data/train.trees` through `preprocess.py` and write the output to `generated/train.trees.pre`. This script makes the trees strictly binary. When it binarizes, it inserts nodes with labels of the form `X*`, and when it removes unary nodes, it fuses labels so they look like `X_Y`.
3. Run `generated/train.trees.pre` through `postprocess.py` and verify that its output is identical to the original `data/train.trees`. This script reverses all the modifications made by `preprocess.py`.
4. Run `generated/train.trees.pre` through `unknown.py` and write the output to `generated/train.trees.pre.unk`. This script replaces all words that appear once with the unknown symbol `<unk>`

Part 1: Grammar (45 points)

First, we will learn a probabilistic Context Free Grammar from trees, and store it in the following format:

```
NP -> DT NN # 0.5
NP -> DT NNS # 0.5
DT -> the # 1.0
NN -> boy # 0.5
NN -> girl # 0.5
NNS -> boys # 0.5
NNS -> girls # 0.5
```

a Write code to read in trees and to count the all the rules used in each tree. To be specific, you will need to complete the `_update_grammar_dfs` method in the `models/parser.Parser` class. In the `part1.parta` function I have already wrote the code necessary to load in the training data and train your model with it, but it will not work until `_update_grammar_dfs` is completed. How many unique rules are there? What are the top five most frequent rules, and how many times did each occur? Include answers to these questions in your report.

b Write code to compute the conditional probability of each rule and print out the grammar in the above format. What are the top five highest-probability rules, and what are their probabilities? Include answers to these questions as well as the printout in your report. I have implemented this functionality in the `part1.partb` function, but it will also not work until you have completed `_update_grammar_dfs`. Note that to convert between counts and conditional probabilities, we invoke the `normalize_cond` method.

Part 2: CKY Parser (45 points)

In this section you will implement the cky functionality of the `models/parser.Parser` class. Whenever we're running the cky algorithm, don't forget to use log-probabilities to avoid underflow, and don't forget to replace unknown words with `<unk>`. In some of the methods I have already added a line that does this in the starter code, so be sure to notice it.

a Run your parser on `data/dev.strings` and save the output to `generated/dev.parses`. Remember that if you cannot find a parse for a sentence, to write a blank line for that sentence. I have implemented much of this functionality in the `part2.parta` function, but it will not work until you have completed several methods:

- `_cky_traverse`. As explained in the comments of this method, while there are many flavors of the cky algorithm (the vanilla cky algorithm, cky viterbi, etc.), all of them use the same chart traversal. In other words, the order in which cells in a chart (or multiple charts in the case of some of the flavors of cky) is always the same no matter the flavor of cky being implemented. So, this method is for getting that cell traversal correct, regardless of how the cells are updated. This method takes two arguments, the sentence itself (split into a list of strings) and a function pointer. This function pointer is responsible for performing the chart update, so the flavor of cky being calculated depends on what the function pointer does when called. This function is supposed to traverse the cells in the cky-ordering, and call the function pointer for each iteration in the traversal. I have implemented a function called `cky` which implements the vanilla cky algorithm. This method calls `_cky_traverse`, so please use it to help test whether or not your cell traversal works. My implementation of vanilla cky uses a specific indexing scheme, of which there are many other choices. You are welcome to change it if you wish.
- `cky_viterbi`. This method, just like the vanilla cky algorithm, should call your `_cky_traverse` method. Instead of implementing the vanilla cky algorithm, this method should implement the cky viterbi algorithm that we can use to find the best parse tree. This means you need to keep track of backpointers for the best choice (so far) as well as the best log probability. Don't forget that the output of this method is the best parse tree along with the log probability of the best parse tree. So, you will need to call another method (that you also need to complete) called `generate_best_tree` to find this parse tree.
- `generate_best_tree`. This method should use your backpointer structure to walk the backpointers and create a `Tree` object for the parse tree with the highest logprob calculated in your `cky_viterbi`. I envision you doing this with a DFS expansion of the backpointers, and have setup a separate method for this to happen in: `_traverse_backptrs_dfs`.
- `_traverse_backptrs_dfs`. As mentioned, this method is used to do the work of generating the parse tree by walking the backpointer structure you created and populated in your `cky_viterbi`. I implemented this using recursion, but you can do this however you like.

Show the output of your parser on the first 5 lines of `data/dev.strings` along with their log probabilities (base 10) in your report.

b Finish the `part2.partb` function. This function should use a trained parser on the development data. However, when calculating the best parse for each dev string, I want you to measure the runtime that your `cky_viterbi` algorithm takes to generate the best parse. Only keep these time measurements when the best parse is found. Generate a plot where the y-axis is the runtime, and the x-axis is the sentence length. Use a log-log scale. Estimate the value of k where $y \approx cx^k$ (I would recommand doing a least-squares fit). Is it close to 3? Why or why not? Include the answers to this question, along with your estimate of k and your log-log plot in your report.

c Run `dev.parses` through `postprocessor.py` and evaluate your parser output against the ground truth for the dev set using `evalb`. I have already completed this functionality for you in `part2.partc`. Report the printout of your performance in your report, including your f1 score. Your f1 score should be at least 88%.

Part 3: Parse Improvement (10 points)

Make at least three modifications to your parser. You can try any techniques we talk about in class, or any others that you find in literature. If you choose to modify `preprocess.py`, then you should (probably) also modify `postprocess.py` accordingly. I would recommend creating a `part3.py` file and creating functions `parta` and `partb`. Feel free to recycle any code you want from `part1.py` and `part2.py`:

a For each of your three modifications, describe in your report what you did. Run your modified parser on `data/dev.strings` and report your new f1 score. **Do NOT run your parser on `data/test.strings` yet**. What helped, or what didn't? Why? Include these details in your report.

b Finally, run your best parser on `data/test.strings`. What f1 score did you get? Report your score as well as what your best parser is in your report. For full credit, your score should be at least 90%.

Submission

Please turn in all of the files that you modified to gradescope. In your report, you should include all of the items asked for by the questions. If an instruction in this prompt says to report something, you should include those elements in your pdf. Please do not handwrite these: typeset pdfs only. There will be separate links on gradescope for your code and your pdf. Your code will be autograded, while your report will be graded by us.