Final Project

Fall 2020

1 General instructions

1. This is the final project of the Advanced CL (fall 2020) course. It is worth 30% of the final grade in the course.
2. A full solution of the project without the bonus question will be graded out of 100. Up to 15 additional points will be given for solving the bonus question. Therefore the total maximal grade for the project is 115.
3. Please take some time in advance to carefully read the project, and make sure that the basic idea, the notations and the requirements are clear to you. It is recommended not to postpone this preliminary reading until you actually start working on the project.
4. You may discuss the project with your friends, but **the writing must be done separately**.
5. Write clear code, document your methods, variables and any piece of code that does anything complex. Assume your reader is not well-versed in the algorithms you are writing. Remember the DRY principle and adhere as much as possible to the PEP-8 style guide, and particularly the issues discussed in the recitation and given to you as feedback on assignments.
6. **Submit your solution files (as described in section 3 below) by email, no later than February 23 at 23:59. There will be no extensions.**

2 The project

Subjects: PCFG, CFL, CNF, near-CNF, probabilistic CKY algorithm.
Note: when working on the project you may use the implementation you created in problem set #9.

The files in the 'Final Project' section on Piazza contain relevant code and content as follows:

- The `pcfg.py` file deals with implementation of a probabilistic context-free grammar, and contains three classes:
  - The `PRule` class - represents a PCFG rule. It contains several already implemented methods.
  - The `PCFG` class - represents a PCFG. It contains the following methods:

    - The constructor (`__init__`), `add_rule` and `remove_rule` methods, which are already implemented.
    - The `cky_parser`, `to_near_cnf` and `is_valid_grammar` methods, which you are required to implement (as described below).
  - The `PCFGChange` class - represents a change to a PCFG and might be useful for solving the bonus question.

- The `ptree.py` file deals with parse tree implementation. It contains the classes `Node` and `PTree`, each of which contains several already implemented methods.
- The `grammar.txt` file contains a PCFG example of a fragment of English.
- The `data.txt` file contains 5 sentences.

Read and understand the given code, and:

1. **Implement the `to_near_cnf` method of the `PCFG` class**, which receives a `PCFG` instance and returns a new `PCFG` instance that represents an equivalent PCFG in near-CNF.

   Use the definition of equivalence for PCFGs and the near-CNF conversion algorithm discussed in the recitation. Remember that since we are dealing with a PCFG now, the process of converting to near-CNF should take care of probabilities as well; during the conversion process, carefully update the probabilities of the rules when needed.

   In your project PDF file, describe the probability updates you have decided on, and explain why the grammar it provides assigns the same total probability to each string as the original grammar.

2. **Implement the `cky_parser` method of the `PCFG` class**. For a PCFG instance representing a PCFG in near-CNF, and given an input string as an argument, this method parses the input string in the grammar. If the input string can be generated by the grammar, the method returns a most likely parse tree for this string. Otherwise it returns `None`.
   Duo to rounding errors close to zero, we say that the string can not be generated by the grammar (and return `None`) if the total probability for the input string is lower than 2^{-50}.
   Implement this method based on the probabilistic CKY algorithm.

3. **Implement the `is_valid_grammar` method of the `PCFG` class**, that decides if the given `PCFG` instance represents a valid grammar. That is, it checks if for each variable V in the grammar, the sum of the probabilities of the rules in which V appears on the LHS is 1.
   Due to the limitations of the Python interpreter, this sum would not always be exactly 1, but it should be very close to 1. So as long as you get a sum of $1 - \varepsilon$ where $|\varepsilon| < 0.0001$, consider the grammar to be valid.

4. **Create a file `main.py` that contains a `main` method**, which works as follows:

   (a) Reads from the `grammar.txt` file, and creates a `PCFG` instance that represents the grammar written in this file.

(b) Converts the `PCFG` instance to a new instance that represents an equivalent PCFG in near-CNF.

(c) Reads the sentences from the `data.txt` file, and for each sentence w it parses w by the grammar, and prints the representation[1] of a most likely parse tree for w, or some informative message if w has not been generated by the grammar.

5. **Bonus question**:
   As we have seen in class, one of the disadvantages of the CKY algorithm is that it returns a tree in CNF format (or in near-CNF in our case). In this bonus question, you will overcome this disadvantage by representing the parse tree as derived by the original grammar.

   (a) **Implement the `adjust_near_cnf_ptree` method of the PCFG class**, which takes a `PTree` instance and a dictionary of changes as arguments. The changes' dictionary represents all the changes that have been made during the conversion of the original PCFG instance to the instance in near-CNF. The method returns the PTree instance of the original PCFG, that is equivalent to the given `PTree` instance.
   Note that in order to implement this method you should first update your implementation of the `to_near_cnf` method, so it returns also the dictionary changes needed for `adjust_near_cnf_ptree`. For this update you may use the `PCFGChange` class, which may be useful for documenting the changes and creating the changes dictionary.

   (b) **Update your answer to question (4), so that for each sentence it also prints out a parse tree in the original grammar.**

3 Submission

Submit the following files:

1. A zip file called `cl_project_id.zip` with `id` replaced by your ID number, containing the code files: `pcfg.py`, `ptree.py`, `main.py`, as well as the input files: `grammar.txt` and `data.txt`.
2. A PDF file called `cl_project_id.pdf`, with `id` replaced by your ID number, which contains:
   (a) A short description of your implementation.
   (b) A description and explanation regarding the probability updates, as required in question (1).
   (c) The printed tree representations, as required in question (4).

   If you solved the bonus question - submit in addition the printed representations of the trees in the original grammar, as required in question (5).

   In order to visualize the parse trees printed by the `PTree` class, you can use Miles Shang's

[1] Recall that if you use `print` to print out an instance of a class (in particular the `PTree` class), by default that class' `__repr__` method will be called to 'stringify' the instance for printing. No need to call `__repr__` explicitly.

syntax tree generator. This is a useful tool for development / debugging, and you may also attach the visualizations to your project PDF if you wish (not mandatory).

3. Mail the files to Alma at almaf@mail.tau.ac.il.