ANLP Assignment 2 2018

Due: Monday 5 November, 3pm, electronic submission only (see end of instructions)

Overview

In this assignment, you will work with a CKY recogniser and grammar of English that we provide to you. With these, you will do the following:

1) Demonstrate your understanding of the recogniser code by adding comments;
2) Review the output produced for some example sentences and comment on one of your choice;
3) Extend the CKY recogniser so that it records sufficient information to allow you to extract parse trees from the chart.

The goals of this assignment are to help you better understand some of the issues involved in natural language parsing, to give you practice with understanding grammars and parser implementation, and to write descriptions of computational problem solving.

This assignment is purely formative which means that you are free to discuss it with whomever you like. We will therefore not be assigning new partners for it, although we do find that many students are more likely to work on the assignment if they have a buddy (or two). You could choose to continue with your current partner, or a find a new one if you want. We will be officially reassigning partners for Assignment 3, and your A3 partner must be different from your A1 partner, so you could choose to use A2 to find someone new and work with them for A3 as well if it works out.

You do _not_ need to submit your work, and it will not be reviewed by human markers: you will be able to have it automarked as often as you like. Details of how you can do this will be available in a few days.

The numerical "marks" next to each task below are just a rough indication of their relative difficulty.

Running the code

We have provided basic code for you to modify, in the hw2.py1 and cky.py2 files, which you should download, along with a file that implements printing and tracing, cky_print.py3 and a file of patches for NLTK, cfg_fix.py4.

cky.py defines the CKY recogniser. You will edit this to add comments and extend the functionality.

cky_print.py augments CKY recogniser to provide pretty-printing functionality. You shouldn’t need to change it at all.

hw2.py is the top-level file, which contains various grammars, and calls to the CKY recogniser. You will edit this in various ways.

Once you have downloaded those files, you should activate the ANLP virtual environment, then you can run the program in Spyder, or at the command line as follows:

```
conda activate anlp
python hw2.py
```

It will print a toy grammar, the trace of a sentence being recognised by the CKY algorithm, and the final chart.

Your main jobs are to explore the larger grammar defined in hw2.py (grammar2) and extend the CKY implementation to collect and display parse trees.

1http://www.inf.ed.ac.uk/teaching/courses/anlp/hw/2018/code/hw2.py
2http://www.inf.ed.ac.uk/teaching/courses/anlp/hw/2018/code/cky.py
4http://www.inf.ed.ac.uk/teaching/courses/anlp/hw/2018/code/cfg_fix.py
5http://thomas-cokelaer.info/tutorials/sphinx/_modules/template.html#MainClass1.function1
Task Specification

Task 1 (35 marks)

The code we have given you is (intentionally) poorly documented. Your first task is to provide appropriate documentation showing that you understand what the code is doing, and how it does that. To do so, add comments (docstrings) at the top of following methods:

- CKY.buildIndices
- CKY.unaryFill
- Cell.unaryUpdate
- CKY.recognise
- CKY.maybeBuild

Your comments should include three parts, used consistently across every method.

1. A brief description of what the method accomplishes, that is, its postcondition: what has been done after the method finishes that was not done before? (This is similar to the type of documentation you would normally write for a user of the method). This part should not describe processes, which go in the next part. However, if there are important data structures that are created in the method you are describing, make sure your comment describes the type of that data structure and what it holds. In the case of the buildIndices method, also say why it makes sense to set up the data structures in this way (i.e., what will they later be used for?)

2. A brief description of how the method accomplishes its postcondition. That is, what is the procedure by which the computation happens? You may need to mention some data structures, variables, or helper methods in this description, but a good description will not describe every variable or line of code in detail. (This part of the comment is not something you would normally include in documentation for a user, but you would include it in documentation for developers). See the comment at the top of CKY.binaryScan for an example of the Postcondition and How parts.

3. A description of the arguments and their return types, if there are any arguments other than self. The comments at the beginning of hw2.tokenize and CKY.__init__ illustrate the recommended quasi-standard for doing this. Note in particular how the type is documented for data structures such as lists, in order to be as informative as possible: for example the type of [3, 4] should be documented as list(int), not simply list. (See sphinx examples for much more documentation about this approach to documenting Python)

Important: You will want to automark cky.py as it is with your changes so far. Before continuing with task 3, you should copy hw2.py and cky.py to hw2_3.py and cky_3.py respectively at this point, making sure that the former imports the latter, so you can edit those copies during task 3.

Task 2

Review grammar2 in hw2.py, which includes rules that generate one or more trees for the following sentences:

1. John gave a book to Mary.
2. John gave Mary a book.
4. John ate salad with mushrooms with a fork.
5. Book a flight to NYC.
6. Can you book a flight to London?
7. Why did John book the flight?
8. John told Mary that he will book a flight today.
Run the recogniser on these sentences using chart2, and look at the resulting matrices using chart2.pprint(). You may wish to uncomment or modify the commented-out code in hw2.py to help you.

You might notice that the code we gave you is not a correct implementation of CKY because it inserts multiple instances of the same category into the same cell. Your next task is to fix this issue.

**Task 3 (10 marks)**

Start from the hw2_3.py and cky_3.py files you copied after task 1.

Now, edit the CKY.recognise method in cky_3.py so that it still returns False if the input is not recognised, but returns the number of successful analyses if it is recognised (by looking at what is in the top right cell of the chart). Uncomment the relevant lines of hw2_3.py so the result is printed for each of the 8 sentences.

Also make sure that nothing else is printed out or displayed. That is, if we type:

```
python hw2_3.py
```

at the command line inside the ANLP virtual environment, your code should print out each of the sentences along with its number of parses, and that is all.

Again, you will want to automark your hw2_3.py and cky_3.py files as they are now.

**Task 4 (10 marks)**

First, copy your hw2_3.py and cky_3.py files to hw2_4.py and cky_4.py respectively at this point, making sure that the former imports the latter, so you can edit those copies going forward.

As implemented, the recogniser has two problems:

1) It is vulnerable to infinite recursion (in the case of unary rules such as X -> X, or X -> Y, Y -> X);

2) It wastes effort by multiplying the effect of local ambiguities upward. That is, for a recogniser, once we know there is one instance of a label in a cell, adding another instance of the same label to that cell not only doesn’t change the eventual outcome, it just causes more unnecessary repetition higher up, potentially exponentially so.

Both of these problems can be fixed by 1) only adding a label to a cell if that label isn’t already in the cell, and 2) only looking to add unary rules if a label was actually added.

Refactor the code so that:

a) the only call to unaryUpdate is in addLabel

b) addLabel does nothing if its argument is already there.

You do not need to worry about updating the tracing parts of the code to work correctly with your refactoring (though you’re welcome to do so if you wish). Just make sure the recogniser itself works.

Note that the result of this change should be that the code at the end of hw2_4.py which prints the number of parses, should always show 1! (Of course, there are still multiple analyses stored in the chart, but the naive way we asked you to count them in the previous part is no longer sufficient to do so.)

You will want to automark your hw2_4.py and cky_4.py files as they are now.

**Task 5 (25 marks)**

First, copy your hw2_4.py and cky_4.py files to hw2_5.py and cky_5.py respectively at this point, making sure that the former imports the latter, so you can edit those copies going forward.
Now, add a method called `firstTree` to the CKY class in `cky_5.py` which constructs an NLTK `Tree` (use `help(Tree)` for details) for the ‘first’ complete parse as the start symbol of the grammar out of all of the parses represented in the top-right-hand corner of the chart’s matrix after a successful run. You will need to

a) Edit the `Label` class definition to hold the necessary child information;

b) Edit some of the existing CKY and Cell methods to construct/exploit this richer label structure;

c) Return the results from the upper-right corner of the matrix as the result of a parse;

d) Just for terminological accuracy, rename `CKY.recognise` as `CKY.parse`.

What is meant by ‘first’ parse? If there is only one parse, it’s obviously also the first. If however there is some ambiguity present, this means that in `some` Cell(s) in the matrix at least one call to `addLabel` will have done nothing, because more than one rule for some category was satisfied and/or more than one child/pair of children satisfied a rule for that category between the cell’s start- and end-points. So, what you did for Task 4 should make it easy to only keep track of one parse---the first one.

Your `cky_5.py` and `hw2_5.py` will be the last things you can have automarked.

**Optional extension**

Only for those who are interested

Add another method called `allTrees` to the CKY class, which returns an iterator over `Trees` for all the complete parses as the start symbol represented in the top-right-hand corner of the chart’s matrix after a successful run.

Why does this require more reworking of the code than just finding one parse? Again, you can make some progress thinking about this even if you are unable to implement the answer.