ANLP Assignment 2 2017

Due: **Wednesday 8 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.

You will need to write a report that explains what you did and answers some questions (see all the points labelled **Report** below).

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.

Policy on working with others and plagiarism is the same as for assignment 1. We do strongly encourage you to work in a pair but you must choose a different partner for each assignment.

Please read the requirements for submission and marking carefully, *before* the last minute---see below.

Running the code

We have provided basic code for you to modify, in the hw2.py[1] and cky.py[2] files, which you should download, along with a file that implements printing and tracing, cky_print.py[3] and a file of patches for NLTK, cfg_fix.py[4].

`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.

`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 can run the program in Spyder, or at the command line as follows:

```
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.

[1] http://www.inf.ed.ac.uk/teaching/courses/anlp/hw/2017/code/hw2.py
[2] http://www.inf.ed.ac.uk/teaching/courses/anlp/hw/2017/code/cky.py
[3] http://www.inf.ed.ac.uk/teaching/courses/anlp/hw/2017/code/cky_print.py
[4] http://www.inf.ed.ac.uk/teaching/courses/anlp/hw/2017/code/cfg_fix.py
[5] http://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.tokenise` 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 sphynx examples[5] for *much* more documentation about this approach to documenting Python)

Note: Your comments will be marked both for clarity and for conciseness. If you understand well what these methods are doing, each of the three parts of the comment should take only a few lines. Part of writing good technical description is using good judgement about what is important, and a too-long description often makes it harder to understand the key ideas.

In your report, include the signature and docstring for each method, formatted to be easily readable. Do **not** include the rest of the code for each method. For example, the following would be acceptable for `CKY.binaryScan`:

binaryScan(self)

(The heart of the implementation.)

Postcondition: the chart has been filled with all constituents that can be built from the input words and grammar.

How: Starting with constituents of length 2 (because length 1 has been done already), proceed across the upper-right diagonals from left to right and in increasing order of constituent length. Call maybeBuild for each possible choice of (start, mid, end) positions to try to build something at those positions.

Important: You must also submit your `hw2.py` and `cky.py` as they are now. Before continuing with task 3, you should copy these files to `hw2_3.py` and `cky_3.py` respectively at this point, making sure that the latter imports the former, so you can edit those copies for that question.

Task 2 (20 marks)

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.
3. John gave Mary a nice drawing 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()`. (Do not include these in your report.) 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. We'll ask you to fix this issue later. For now, it makes it easier to see how many analyses there are for each sentence.

Choose *one* sentence from the above for which `grammar2` gives more than one distinct parse. In your report,

a) Give a drawing of *two* of the parse trees generated by the grammar for this sentence. (If there are more than two, just give two.) One of these should be the tree that best corresponds to what you think the meaning of the sentence is (Tree A), and one should be any other parse of the sentence (Tree B).

b) Briefly discuss whether you judge the alternative to be legitimate (that is, the sentence really is, or easily could be, ambiguous) or inappropriate (the other parse implies a meaning which is impossible, or at least extremely unlikely). Paraphrase the meaning of your preferred reading and the alternative one in ways which show why each one either is or is not reasonable. Can you think of a sentence of English for which Tree B is the correct parse, and Tree A is not? That is, can you think of a way to replace one or more words in the sentence with other words that have the same parts of speech, so that the opposite parse is preferred? You don't need to limit yourself to the words in the grammar we gave you (in other words, you can imagine adding additional rules to generate more terminal symbols if you like).

Instructions for producing the drawings: First sketch the right tree by hand (working bottom-up may be easier than top-down, at least to get started), then use NLTK commands in Python, by substituting your own bracketed string for the one in the example below:

```
import nltk
treeStr = "(S (NP (Pro he)) (VP (Vt ate) (NP (N salad) ) ) )"
tree = nltk.tree.Tree.fromstring(treeStr)
tree.draw()
```

From the resulting display, increase the size of the tree using `Zoom`, then select `File/Print to Postscript` to save. Then from the command line:

```
convert [yourfile].ps [yourfile].png
```

where `[yourfile]` is whatever name you used to save the postscript. You can then include the `png` version in your report.

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.

Nothing is required in the report for this task, but you must submit 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 latter imports the former, 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.)

Nothing is required in the report for this task, but you must submit 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 latter imports the former, 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 **S** 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.

In your report, document your design and implementation. Include a brief introduction to CKY parsing as necessary background. You do not need to have a complete working implementation to write this section of the report: if necessary, just document what you achieved, and where you fell short.

This task is not easy, but sensible attempts will be rewarded even if complete success is not achieved.

You should also submit your `cky_5.py` and `hw2_5.py` as they are at this point.

Optional extension

Only for those who are interested, please don't hand anything in but it may be a fun learning exercise.

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

Why does this requires 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.

Marking

Your report will be graded based on the following criteria:

a) quality of the comments you add as instructed;

b) correctness of your parse trees;

c) correctness of your extensions to the parser;

d) quality of your written report.

"Quality" here means the three c's: clarity, correctness, and conciseness.

We will not be assigning any marks based on the quality of your code (provided it works).

Your code *must* run on a standard DICE machine using the virtual environment for this course, without additional package installations beyond what we've used in labs.

Each version of `hw2.py` that you submit *must* print out or display all and only the information necessary to demonstrate your solution for the relevant task, if run from the command line as follows:

```
python hw2_X.py
```

for the relevant value of `X`.

Note well: If we have to edit your files to get them to run, or if your `CKY.recognise` doesn't run when called as shown in `hw2.py`, your final mark will be reduced by 10. This is a serious, but easily avoided, penalty. Do *not* assume that you are OK if your code runs on a personal machine.

As for `CKY.parse` and `CKY.firstTree`, the same penalty applies if you claim to have working versions of these but we can't call them as expected. If you have not achieved a full implementation, you must in any case make sure your own `hw_5.py` runs (i.e., don't call any code that would return an error), and state clearly in your report if your `CKY.parse` and `CKY.firstTree` return errors when called as shown in `hw2.py`.

Submitting your assignment

Submit *one* report for your pair, in .pdf format, along with your code, using the following command on a DICE machine:

```
submit anlp cw2 hw2.py hw2_3.py hw2_4.py hw2_5.py cky.py cky_3.py cky_4.py cky_5.py report.pdf
```

where `hw2.py` and `cky.py` etc. are your edited versions of those files, and `report.pdf` is the file with your report.

Please convert other files types (eg Word documents) into .pdf format before submitting. Make sure both student's ID numbers are included in the report! Please include *only* your ID numbers, not your names, to help avoid unconscious bias by the marker. Both members of the pair will receive the same mark.

You can submit more than once up until the submission deadline (but not after: see Late Submissions below). All submissions are timestamped automatically. Identically named files will overwrite earlier submitted versions, so we will mark the latest submission that comes in before the deadline.

Make sure you **save** your code and report **before** submitting, and that you go through the whole submission process (you may need to answer some y/n questions) and get a message in the terminal that submission succeeded.

Late submissions

If you submit anything before the deadline, you may not resubmit afterward. (This policy allows us to begin marking submissions immediately after the deadline, without having to worry that some may need to be re-marked).

If your pair does not submit anything before the deadline, you may submit *exactly once* after the deadline, and a late penalty will be applied to this submission unless you have received an approved extension. Please be aware that late submissions may receive lower priority for marking, and marks may not be returned within the same timeframe as for on-time submissions.

Warning: Unfortunately the `submit` command will technically allow you to submit late even if you submitted before the deadline (ie. it does not enforce the above policy). Don't do this! We will mark the version that we retrieve just after the deadline, and (even worse) you may still be penalized for submitting late because the timestamp will update.

For additional information about late penalties and extension requests, see the School web page below. Do **not** email any course staff directly about extension requests; you must follow the instructions on the web page.

url{http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests}