CS114 (Spring 2015) Homework 5 Part Two
Statistical Parser: Extracting PCFG

Due March 27, 2015

Introduction

The starter code for this part of the homework is the same as the one from part one. You can download all the code and supporting files (including this description) by updating your local github repo.

Files you will edit and turn in:

- `extractGrammar.py` where you'll put your code for problem 3.
- `grammar.py` where you'll put your code for problem 2.

Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code.

2 Effects of the P in the PFCG

The analysis from before gave us a specific interpretation of the time flies sentence. Your task is to adjust the grammar so that we get a different (specific) interpretation.

Open up `grammar.py` and take a look at the definition of `timeFliesPCFG` and the desired analysis in `desiredTimeFliesParse`. Create a new grammar called `timeFliesPCFG2` that gives the same parse as in `desiredTimeFliesParse`.

Note: you may NOT change the rules of the grammar, and none of your probabilities may be less than 0.1 (and they must sum to one in the appropriate places). You can compare your output to the desired tree with:

```
>>> myTree = parse(timeFliesPCFG, timeFliesSent)
>>> print myTree
(TOP:
  (S:
    (NP: (Noun: 'time'))
```

```
    (VP:
      (Verb: 'flies')
      (PP: (Prep: 'like') (NP: (Det: 'an') (Noun: 'arrow'))))))

>>> print desiredTimeFliesParse
(TOP:
  (S:
    (VP:
      (Verb: 'time')
      (NP: (Noun: 'flies'))
      (PP: (Prep: 'like') (NP: (Det: 'an') (Noun: 'arrow'))))))

>>> evaluate(desiredTimeFliesParse, myTree)
0.75
```

This is the result using the grammar in the starter code. You should be able to get an evaluation (accuracy) of 1.0 after you modify the probability of the PCFGs.

3 Grammar writing or what I call "So you think you know syntax."

Your final task is to improve English parsing by modifying the nonterminals in the trees from which we extract the PCFGs. We provide a demo to show how to use the provided code to extract and evaluate PCFGs.

3.1 Horizontal Markovization

Your first task is to implement horizontal Markovization (i.e., parent annotations) in the `extractGrammar` file. You should be able to test this with:

```
>>> pcfg = computePCFG('wsj.train', horizSize=2)
>>> evaluateParser(pcfg, 'wsj.dev', pruningPercent=0.00001, horizSize=2)
```

3.2 Vertical Markovization

Your second task is to implement vertical annotation (i.e., ancestor annotations). Likewise, you can test this with:

```
>>> pcfg = computePCFG('wsj.train', verticSize=2)
>>> evaluateParser(pcfg, 'wsj.dev', pruningPercent=0.00001, verticSize=2)
```

3.3 Grammar writing and splitting nonterminals

The final challenge is to make the best unlexicalized grammar that you can. First, you should try to get the best results of varying `horizSize` in the range {1, 2, 3, 4} and `verticSize` in {1, 2, 3, 4}.

Use your knowledge of syntax and linguistic intuition to modify or subcategorize the non-terminals of the trees in the treebank. Some thoughts and ideas:

- Does it make sense to have a separate non-terminal for a finite verb vs non-finite verb? How about transitive verbs?
- How about "proper noun phrase"? Should it behave different from a noun phrase? What are other subcategories of noun that we can easily annotate.
- Are all prepositions created equal? Does each different preposition deserve its own tag e.g "from phrase"? (It is OK to "lexicalize" words that are from a closed class such as preposition or determiner. But you are not allowed to explicitly lexicalize nouns.)
- 1988 and 10,500 are both tagged as CD. But they really appear at very different places. Any other subcategories of CD?

To evaluate your grammar, you can also `runFancyCode=True` to `evaluateParser`, which will pass this flag down to the `binarizeTree`function, and you can do whatever you want in there to try to get better performance except lexicalizing the grammar for words from open classes e.g. verbs, nouns.

Submit `extractGrammar.py` and `grammar.py`. Also write a short report on what you have tried and give a reason why it is a sensible idea to try. Include a table that summarizes F_1 score and the size of the grammars (the number of rules) resulting from each strategy you try and each combination of values of `horizSize` and `verticSize` .

Grammar extraction and evaluation demo

Open up `extractGrammar.py` and take a look at `computePCFG`. This will read data and compute a PCFG out of it. For instance:

```
>>> pcfg = computePCFG('wsj.dev')
>>> len(pcfg)
650
```

```
>>> print str(pcfg)
...
PP => VBG PP | 1
PP => TO NP | 23
PP => IN ADJP | 1
PP => IN_NP NP | 2
PP => TO S | 1
PP => VBN PP | 3
PP => IN SBAR | 1
PP => IN NP | 139
...
```

This shows that there are 650 unique rules in this PCFG, and that the most frequent PP rules were "TO NP" (count of 23) and "IN NP" (count of 139). We can look at a larger data set:

```
>>> pcfg = computePCFG('wsj.train')
>>> len(pcfg)
6048
```

By default, this will learn a completely unlexicalized PCFG, which means that it can only parse POS sequences, as in the following two time-flies-esque examples:

```
>>> parse(pcfg, ['NN', 'VBZ', 'IN', 'DT', 'NN'])
(TOP: (S: (NP: 'NN') (VP: 'VBZ' (PP: 'IN' (NP: 'DT' 'NN')))))

>>> parse(pcfg, ['VBZ', 'NN', 'IN', 'DT', 'NN'])
(TOP: (S: (VP: (_VBZ_NP: 'VBZ' (NP: 'NN')) (PP: 'IN' (NP: 'DT' 'NN'))))))
```

You'll notice that the tree that came out the second time is binarized. We can de-binarize it:

```
>>> print nonBinaryTree
(TOP:
  (S:
    (NP:
      (DT: 'the')
      (RB: 'really')
      (JJ: 'happy')
      (NN: 'computer')
      (NN: 'science')
```

```
      (NN: 'student'))
    (VP: (VBD: 'loves') (NP: (NNP: 'CL1')))
    (.: '.')))

>>> print binarizeTree(nonBinaryTree)
(TOP:
  (S:
    (_NP_VP:
      (NP:
        (_DT_RB_JJ_NN_NN:
          (_DT_RB_JJ_NN:
            (_DT_RB_JJ:
              (_DT_RB: (DT: 'the') (RB: 'really'))
              (JJ: 'happy'))
            (NN: 'computer'))
          (NN: 'science'))
        (NN: 'student'))
      (VP: (VBD: 'loves') (NP: (NNP: 'CL1'))))
    (.: '.')))

>>> print debinarizeTree(binarizeTree(nonBinaryTree))
(TOP:
  (S:
    (NP:
      (DT: 'the')
      (RB: 'really')
      (JJ: 'happy')
      (NN: 'computer')
      (NN: 'science')
      (NN: 'student'))
    (VP: (VBD: 'loves') (NP: (NNP: 'CL1')))
    (.: '.')))
```

You'll note that this implementation of binarization does NO parent annotations (i.e., vertical order is 1) and complete markovization (i.e., horizontal order is infinity). One very important thing is that the debinarization assumes that any rule that's been binarized starts with `_` . So please maintain this invariant or debinarization won't work! You can evaluate this PCFG by loading parser.py and running:

```
>>> evaluateParser(pcfg, 'wsj.dev')
```

You might notice this takes forever. So to make it faster, you can specify a pruning threshold. A pruning threshold of 0.1 means that once a cell is filled up, take the probability of the best item in it. Say that's probabiliy 0.23. Multiply it by 0.1 to get 0.023. Now, anything else in that cell whose probability is less than 0.023 will be deleted. This will make parsing faster at the expense of accuracy. You can pass this threshold to evaluateParser:

```
>>> evaluateParser(pcfg, 'wsj.dev', pruningPercent=0.00001)
0.62969931444237581

>>> evaluateParser(pcfg, 'wsj.dev', pruningPercent=0.001)
0.49305803094859446

>>> evaluateParser(pcfg, 'wsj.dev', pruningPercent=0.1)
0.019573492787778504
```