Lab for week 7: Probabilistic Parsing

Author: Henry Thompson
Author: Bharat Ram Ambati
Author: Sharon Goldwater
Date: 2016-10-30, 2018-10-24

Use the html version[2] of this lab if you want to easily access the links or copy and paste commands.

Or use the pdf version[3] if you want nicer formatting or a printable sheet.

Goals and motivation of this lab

In this lab we explore probabilistic phrase structure grammars and chart parsers which use such grammars. We're using the same data as the previous lab, 3914 treebanked sentences from the Wall Street Journal.

As always, we have written most of the code for the lab already and included a lot of explanatory comments, but we will ask you to add a few things here and there. For students with more programming background, the 'Going Further' section will give you a chance to explore some more advanced topics.

Preliminaries

As usual, create a directory for this lab inside your `labs` directory:

```
cd ~/anlp/labs
mkdir lab6
cd lab6
```

Download the files lab6.py[4], lab5_fix.py[5] BetterICP.py[6] into your `lab6` directory: From the Lab 6 web page[7], right-click on the link and select *Save link as...*, then navigate to your lab6 directory to save.

Launch our environment and start up spyder (or ipython, or just use python: by now you probably know how you prefer to work):

```
conda activate anlp
spyder &
```

[1] http://creativecommons.org/licenses/by-nc/4.0/.
[2] lab6.html
[3] lab6.pdf
[4] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab6.py
[5] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab5_fix.py
[6] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/BetterICP.py
[7] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab6.html
[8] https://gist.github.com/nlothian/9240750

Running the code

You can run this week's lab as follows:

```
%run lab6.py
```

It should print the first parsed sentence and the productions in it.

Following lab 5, we use the Penn Phrase Structure Treebank for this lab as well. This data consists of around 3900 sentences, where each sentence is annotated with its phrase structure tree. We use NLTK libraries to load this data.

(Don't forget to do:

```
%run lab6.py
```

after you've done some edits, in order to get your new function definitions.)

Probabilistic Phrase Structure Grammar (PCFG)

Probabilistic Phrase Structure Grammars (PCFGs) are Phrase Structure Grammars, where each production has a probability assigned to it. Consider the toy PCFG grammar. In NLTK, the data type of this grammar is `ProbabilisticGrammar`:

```
# Grammatical productions.
S -> NP VP [1.0]
NP -> Pro [0.1] | Det N [0.3] | N [0.5] | NP PP [0.1]
VP -> Vi [0.05] | Vt NP [0.9] | VP PP [0.05]
Det -> Art [1.0]
PP -> Prep NP [1.0]
# Lexical productions.
Pro -> "i" [0.3] | "we" [0.1] | "you" [0.1] | "he" [0.3] | "she" [0.2]
Art -> "a" [0.4] | "an" [0.1] | "the" [0.5]
Prep -> "with" [0.7] | "in" [0.3]
N -> "salad" [0.4] | "fork" [0.3] | "mushrooms" [0.3]
Vi -> "sneezed" [0.6] | "ran" [0.4]
Vt -> "eat" [0.2] | "eats" [0.2] | "ate" [0.2] | "see" [0.2] | "saw" [0.2]
```

Each production has a probability assigned to it. Change the probability of the production `NP -> NP PP` from 0.1 to 0.01. Re-run the code. What is the error given by the code?

Change the value back before continuing!

Comment out the three lines that are printing out the first sentence and its productions.

PCFG Parser:

We made a simple class called `BetterICP`. It uses NLTK's `InsideChartParser` module which is a probabilistic chart parser (`help(nltk.InsideChartParser)` for more details).

BetterICP class has a method `parse` which parses a sentence and prints all possible parses. This method takes three arguments. First argument is `tokens`, which is a list of words in the sentence. Second argument is `notify`. `notify=True` will print each parse as it is found without waiting until the end of the process. Third argument `max` defines the number of possible parses to be printed. The parsing process will stop once `max` number of parses have been found.

Un-comment the lines in `lab6.py` which initialise the `BetterICP` class with our simple grammar and call the method `parse`, then reload.

The figures to the right of the parse gives the probability, the total cost of the tree and the cost of the spanning `S` edge.

Note that our toy grammar has an `NP -> NP PP` rule which created problem with our recursive descent parser. Why didn't it pose a problem now? There is another rule in our grammar which might have created a left recursion problem. What is that rule?

Run the parser on sentences "he sneezed" and "he sneezed the fork". What is the output for each of these sentences ?

Ranking parses

Uncomment the next two parses, so the PCFG parser runs on the sentences "he ate salad with mushrooms", and "he ate salad with a fork". Note that the parses are ranked based on their probabilities (high to low)/costs(low to high).

Observe the PP-attachment (preposition phrase attachment) and identify which VP production is preferred. In the best parse, is PP attached to NP (noun phrase) or VP (verb phrase) ?

Edit the grammar and switch the probabilities of the two rules involved. Re-load, with appropriate commenting so that only sentence2 and 3 are parsed. What changes ?

Turn tracing on, by adding the following *before* the `sppc.parse` lines in `lab6.py`:

```
sppc.trace(1)
```

and rerun the three parses you've done so far.

Can you see how the order in which edges are added is determining what analysis gets found first?

Can you see now the parse is effectively breadth-first, as discussed in lecture? Try to spot where a shorter edge that will eventually not be part of the best overtakes a longer one that will.

Try changing the rule probabilities back to where they started, and watch again.

Working with the real grammar

In the above sections, we worked with a toy grammar created by hand. For example, consider `Prep -> "with" [0.7] | "in" [0.3]` production in our toy grammar. This means that out of all the possibilities there are only two prepositions in the language, and that e.g. when generating they should be chosen with the associated probabilities.

In this section, we shall work with the grammar from the treebank. `psents` contains all the parsed sentences in the treebank. `prods = get_costed_productions(psents)` gives all the productions in the treebank with their costs.

And with the treebank-based grammar, as we see from `get_costed_productions`, the cost is not some made-up number, but the -base-2-log of the maximum-likelihood estimate of the probability of each production, based on its frequency in the treebank.

Create a PCFG by uncommenting the `prods=...` and `ppg=PCFG(Nonterminal('NP'), prods)` lines in `lab6.py` and re-loading.

The first argument to `PCFG` is the start symbol and the second argument is the productions with their costs. The first argument *just* determines the start symbol for parsing purposes, it doesn't restrict the productions that are included. That is, *all* the productions in `prods` will be in `ppg`.

Now try `sprods = ppg.productions(Nonterminal('S'))`, which gives a list of all the productions that start with 'S'. How many productions are there in the treebank that start with 'S'?

There is a useful complete listing of the tagset used in the Penn Treebank[8] online.

Print the first 10 productions in this list to get a feel of the kind of productions used in the treebank, and why we're going to start with NP for our little experiment.

Note also that punctuation symbols do appear both in the trees and the grammar. Many common punctuation marks occur as pre-terminals for themselves (and sometimes some close friends), that is, for example, `, -> ','` and `. -> '.' | '?' | '!'`, but in order to avoid confusion with the tree displays, this does *not* apply to brackets, which use three-letter acronyms, as follows:

```
-LRB- -RRB- -RSB- -RSB- -LCB- -RCB-
  (     )     [     ]     {     }
```

Finally, irritatingly, the `-digits` which regularly appear at then end of some tags are coreference markers which pair up e.g. a relative pronoun and a subsequent gap, as in:

```
(NP
  (NP (DT the) (NN executive) (NNS functions) )
  (SBAR
    (WHNP-2 (WDT that) )
            (S
              (NP-SBJ (DT the) (NNP Confederation) (NNP Congress) )
              (VP (VBD had)
                (VP (VBN performed)
                  (NP (-NONE- *T*-2) ))))))
the executive functions that the Confederation Congress had performed
```

Un-comment the following lines and run the code:

```
ppc=BetterICP(ppg,1000)
print "beam = 1000"
ppc.parse("the men".split(),True,3)
```

What happened? Why do you think that is?

Now comment out the last two of the above three lines, and remove the comments from the *next* three lines, to widen the beam to 1075, and run again.

What's surprising about the results, given our discussions in class about the relative frequency of 'the' as DT and 'the' as JJ??

Uncomment three more lines to try width 1200 (but *do not* uncomment the `trace` lines).

You should now be able to see three different parses for this simple noun phrase. These parses are ranked based on increasing costs. The number printed with the parse tree is the cost. So, the lower the number, the higher the probability of that parse. We're *still* not seeing the simple `DT NNS` structure we might expect.

Finally uncomment on the last three lines to widen the beam rather a lot more, and run again.

What has finally happened?

How could *increasing* the beam width have improved the results the way it did?

Finally turn tracing on by uncommenting the:

```
ppc.trace(1)
```

line, and arranging the other comments so only the beam-width 1900 parse will be attempted. *Do not* run this from inside ipython (or use Control-C to interrupt it if you do :-).

From the terminal command line, do:

```
> python lab6.py > ptrace.txt
```

Use:

```
> less ptrace.txt
```

to look at the results, in the form of a line for each edge. The ASCII art in the left-hand column is a bit hard to make sense of at first, but the remaining columns are easy:

- the interval covered (e.g. "[1:2]" for an edge between vertex 1 and vertex 2)
- the word or dotted rule on the edge -- if the `*` (dot) is at the end, the edge is inactive, otherwise active
- the cost

You can also use 'grep' to find what's happening to the two rules we care about, as follows:

```
> egrep -n 'the|men|NP -> ([*]\s)?(DT|JJ)\s([*]\s)?(NNS|NP)(\s[*])?\s*\[' ptrace.tx
```

Note that tracing at this level shows edges as the come *off* the agenda and into the chart, so we can't see exactly why the result we're looking for never appeared.

To see more detail edit the file again to switch to beam-width 1200, and up the level of detail in the trace to show pruning:

```
ppc.trace(3)
```

Again, run from *outside* ipython:

```
> python lab6.py > ptrace2.txt
```

Use the `less` and the `egrep` command on `ptrace2.txt` this time to see what went wrong. Can you see why we're 'losing' the correct answer? Where are the edges coming from that are filling up the beam?

You may want to try exploring `ptrace2.txt` with `less -N` and its `/regexp` command which skips forward to the next occurrence of that regexp. Also note that you don't have to quit and type `less -N` to back up and look for something else, you can just use `b` to *b* ack up a page, and `g` to *g* o back to the beginning.

If you're more comfortable searching inside an editor, you can just load `ptrace2.txt` into an editor and do that.

Aside from the speed of our old DICE machines, what's wrong with the grammar that's causing so much wasted effort? (Hint: A simpler answer than the ones we've looked at in class is all that's needed.)

Going Further:

1. Further to the last question, try sorting `prods` in increasing order of cost -- what stands out about many of the lowest cost productions? (Hint: what's the easiest way for some production to have cost 0?)

You can check your guess by referring to the counts you get from:

```
pd=production_distribution(psents)
```

using the supplied `production_distribution` function, similar to the one we used last time, but which just counts productions without separating lexical from non-lexical.

2. Draw the 'correct' phrase structure tree for the sentence "he ate salad with a fork" manually according to the toy grammar. Compare your tree with the output trees when parsed with `sppc`.

To draw your correctly bracketed structure, represented as a string, use NLTK commands as shown below, using your own string:

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

3. Update the grammar to handle imperative sentences such as "eat the salad".

4. Take some sample sentences, draw their correct phrase structures, get the first best output from the PCFG parser (set the beam width as low as you can!) for these sentences and evaluate them using parseval.

5. Create a text file called "input.txt" and write three sentences "he ate salad", "he ate salad with mushrooms", and "he ate salad with a fork" into this file, where each sentence is in a separate line. Write a function which reads sentences from this file, parses them and print the output into "output.txt" file.

Extra credit: Recompute the productions while ignoring all the `-digit*` suffixes on Nonterminal symbols which occur throughout the treebank. How many productions now?. Rebuild the grammar. Re-run some tests -- any changes?

Even more extra credit: Edit the code to a) actually store costs, not probabilities and b) use a figure of merit as discussed in class rather than the simple sum of costs. This *should* fix the problem observed at the end of the lab above.