Syntax

Lab Session

Recursive Descent Parser

Parsing a sentence consists of finding the correct syntactic structure for that sentence using a given grammar, where the grammar contains the structural constraints of the language. Parsing is one of the major tasks which helps in processing and analysing natural language.

In this part of the lab we explore recursive descent parsing using some simple Phrase Structure Grammars (CFGs). We aim to give you a sense of how much computation is potentially involved in parsing sentences, and thus why cleverer parsing algorithms are needed. We also aim to give you some practice writing grammars, to see how the choices you make can interact with the parsing algorithm, sometimes in undesirable ways.

This part of the lab is based on a graphical demo app that runs inside NLTK and shows you a recursive descent parser in action. The second part looks at the grammar rules and parsed sentences in a more realistic broad-coverage grammar (from a subset of the Penn Treebank).

Download the file `lab5.py` and `lab5_fix.py` from the course Moodle page into your local directory. Open the file with your preferred editor and start up `ipython`.

Check that NLTK is working by downloading the corpus for the second part of this lab:

```
import nltk
nltk.download('treebank')
```

1 The RDP Parser

We'll start with the following toy grammar, implemented as `grammar1` in the lab code:

```
# Grammatical productions.
S -> NP VP
NP -> Pro | Det N | N
Det -> Art
VP -> V | V NP | V NP PP
PP -> Prep NP
# Lexical productions.
Pro -> "i" | "we" | "you" | "he" | "she" | "him" | "her"
Art -> "a" | "an" | "the"
Prep -> "with" | "in"
N -> "salad" | "fork" | "mushrooms"
V -> "eat" | "eats" | "ate" | "see" | "saw" | "prefer" | "sneezed"
Vi -> "sneezed" | "ran"
Vt -> "eat" | "eats" | "ate" | "see" | "saw" | "prefer"
Vp -> "eat" | "eats" | "ate" | "see" | "saw" | "prefer" | "gave"
```

Our code reads in this grammar and turns it into an `nltk.grammar.ContextFreeGrammar` object using `parse_grammar` (defined in `lab5_fix.py`, which you need not examine).

Launch the NLTK recursive descent parser app from ipython like this:

```
%run lab5.py
app(grammar1,sentence1)
```

This shows an app. The five coloured buttons at the bottom of the resulting app window control the parser. Try 'Step' a few time. Note how the display corresponds to the description of the recursive descent algorithm in Monday's lecture, in particular the way the current subgoal and possible expansions are highlighted.

Now 'Autostep'.

Watch progress in the tree window, and individual actions in the 'Last Operation' display. Hit 'Autostep' again to stop the parser. Try 'Step' again a few times, and see how the app indicates which options at a choice point have already been tried, and which remain.

Try some other sentences (using Edit -¿ Edit Text in the menu bar) and watch how the parser runs. You can speed things up by selecting Animate -¿ Fast Animation. The parser stops when it gets the first valid parse of a sentence. You can start it again to see if there are more valid parses, or you can reset it under the File menu.

Try parsing: `i ate the salad with a fork`

Slow the animation back down, and start with Autostep, but stop that when it gets to work on the Det N version of NP as applied to "the salad". Step through the next bit one at a time, to see just how much backtracking (and wasted effort) is involved in getting back to the correct expansion of VP, followed by a complete (and lengthy) recapitulation of the parsing of "the salad" as an NP.

2 Adding productions to the grammar

Run the parser on the sentence He ate salad (note the capital "H"). Can you parse this sentence using the toy grammar provided? What production would you have to add to the grammar to handle this sentence?

Look at the parse trees for the following sentences:

```
he ate salad with a fork
he ate salad with mushrooms
```

Is there a problem with either of the parse trees? (Hint: Observe the attachments of prepositional phrases ("with a fork", "with mushrooms") in both the parse trees).

Using the demo app, add a production `NP -> NP PP`, after the other NP rules. (Click on Edit-¿Edit Grammar to do this). Re-run the parser on one of the above sentences and take note of the parse tree. Then, change the order of the rules `NP -> N` and `NP -> NP PP`, so that the `NP PP` expansion is used by the parser first. Run the parser on the one of the sentences again.

What is the problem with this new ordering and how does the parser's behaviour differ from the other ordering?

How do you think this behaviour depends on the particular way this RD parser chooses which rule to expand when there are multiple options?

3 Exploring a treebank grammar

The previous parts of the lab dealt with a toy grammar. The remaining parts give you some idea of what a broad-coverage treebank and treebank-derived grammar looks like. We consider a small part of the Penn Phrase Structure Treebank. This data consists of around 3900 sentences, where each sentence is annotated with its phrase structure tree. Our code uses nltk libraries to load this data and extract parsed sentences.

3.1 Extracting and viewing parsed sentences

Make sure you've downloaded the corpus (see Preliminaries) and uncomment the following two lines in the lab code:

```
psents = treebank.parsed_sents()
print_parse_info(psents[0])
```

Then save and re-run the file.

It should print the first parsed sentence (at index 0), and the grammatical and lexical productions used in the first sentence.

The first line of these two extracts the list of parsed sentences from the treebank (which is imported at the top of the file), and `psents[0]` gives the first parsed sentence. When you print it (this happens inside `print_parse_info`), it shows up as a sort of left-to-right sideways tree, with parentheses around each node label and its children. Words are at the leaves, with each word dominated by a single pre-terminal label (POS category). These POS-word pairs are then grouped together into constituents labelled with nonterminal labels (phrasal categories).

You can also look at the parses and productions for other sentences by changing which sentence is passed in to `print_parse_info`. Verify this by looking at the parse and productions for the second sentence in the corpus.

Check the methods available for this object using the command `help(psents[0])`. You can ignore the ones beginning with "__", they are Python-internal ones.

Try using the `draw()` method on some of the sentences. Remember, `draw()` is a method, not a function, so the syntax is, e.g., `psents[0].draw()`.

Extract the list of words and the list of (word, pos-tag) tuples from `psents[0]` using some of the other available methods.

Probabilistic Parsing

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

Download the files `pcfg.py`, `pcfg_fix.py` and `BetterICP.py` into your local directory. You can run the code in `ipython` as follows:

```
%run pcfg.py
```

It should print the first parsed sentence and the productions in it. Following the first part of the lab, 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.

4 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`.

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.

5 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

```
# 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]
```

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 `pcfg.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 ?

6 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 `pcfg.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.

Implementing the probabilistic CKY algorithm

The goal of this exercise is to implement the CKY algorithm for a PCFG. The grammar that we will be working on will not be in Chomsky Normal Form (CNF), but simply in *binarized* form. The n-ary rules are transformed into binary rules, and the unary rules are left as they are.

Start by download and unpacking the archive `pcfg_cky.tbz2` which contains the code template and data for this exercise.

7 The code template

The main function of the provided `PCFGParserTester.py` module begins by reading trees for learning and testing. The learning trees are used to construct a default parser (`class BaselineParser`) that implements the Parser class interface, which defines two methods: `train` and `get_best_parse`. The analyzer is then used to predict the best analysis for the test sentences. The trees produced are finally compared to the reference trees in order to compute the precision, recall and F1 scores.

The default parser is simplistic: it takes as input each sentence, assigns a label to each word (using a unigram sequence labeller) and then analyses the sentence as follows: if the sequence of labels has already been seen in the training corpus in some tree, it is this tree that is chosen as the analysis. Otherwise, the analyser tries to find the best constituent, relying solely on the length of the constituents. This is obviously a terribly bad algorithm, and your goal is to implement the CKY algorithm which should allow for better analyses.

You can start by examining the classes in:

- `ling/Tree(s).py` classes relating to CFG trees
- `Lexicon` lexical rules and associated probabilities. This rule can be used as a unigram sequence labeller. This class allows to correctly handle rare and unknown in the training corpus
- `Grammar, UnaryRule, BinaryRule` grammars and rules. The Grammar class computes the Maximum Likelihood (MLE) estimate of rule probabilities:

$$p(X \to \alpha) = \frac{\text{count}(X \to \alpha)}{\sum_\lambda \text{count}(X \to \lambda)}$$

  The classes `UnaryRule` and `BinaryRule` store production rules and their probabilities.

You can start by training your system on the `miniTest` corpus which contains 3 training sentences and 1 test sentence.

8 Running the code

Use `--data` to specify the corpus to use (the directory name, miniTest or masc) and `--path` to indicate where the corpus is located. For example:

```
python PCFGParserTester.py --path ../data/ --data miniTest
```

9 Implementing CKY

Your first task is to implement the CKY algorithm, in the `PCFGParser::get_best_parse` method. First, use the corpus `miniTest` to verify that your program is correct. Based on the 3 learning trees of this small corpus, your analyser should be able to return the correct tree for the test sentence.

To adapt your algorithm to all corpora, `masc` in particular, you must be able to handle n-ary rules. There are two possibilities:

- Use the function `TreeAnnotations::binarize_tree` that binarises the n-ary rules ($n > 2$). Inconvenience: unary rules, even non-lexical, are always in the grammar. It is therefore necessary to adapt the CKY algorithm (which works on CNF grammar) to handle the unary rules, and not only the binary rules. The algorithm thus has an additional step: after having tried all the binary rules to fill the chart, it should try all the unary rules. Be careful to allow the chaining of unary rules.
- Implement the transformation of the grammar into Chomsky Normal Form (CNF). Disadvantage: in addition to transforming the grammar, we must re-estimate the weight of the rules, and furthermore, re-transform the output trees in order to be able to make the evaluation.
  Definition: a CNF grammar is a context-free grammar where the RHS of each production rule is restricted to be either two non-terminals or one terminal, and no empty productions are allowed.

You should return a tree that has been debinarized (`TreeAnnotation::unannotate_tree`).

Your analyser should be able to reach an F-score of around 73% on the corpus masc.