INF4820; Fall 2017: Obligatory Exercise (3b)

High-Level Goals

- Understand probability estimation for PCFGs, and implement PCFG training.
- implement the ParsEval metrics and evaluate quantitatively the performance of the parser.
- Understand how the Viterbi algorithm works over a packed parse forest.
- Investigate various aspects of parser performance and how to improve them.

Background

This is the second and *final* part of the third obligatory exercise in INF4820. You can obtain up to ten points for this problem set, and you need a minimum of twelve points (or 60,% of the total available) for (3a) and (3b) in total. If you have any questions, please post them on our Piazza discussion board or email `inf4820-help@ifi.uio.no`, and make sure to take advantage of the laboratory sessions.

Solutions must be submitted via Devilry by midnight (23:59) on Friday, November 17. Please provide your code and comments in a single '`.lisp`' file. For the theoretical questions, you can either include your answers as Lisp comments in the same file, or submit an additional text file.

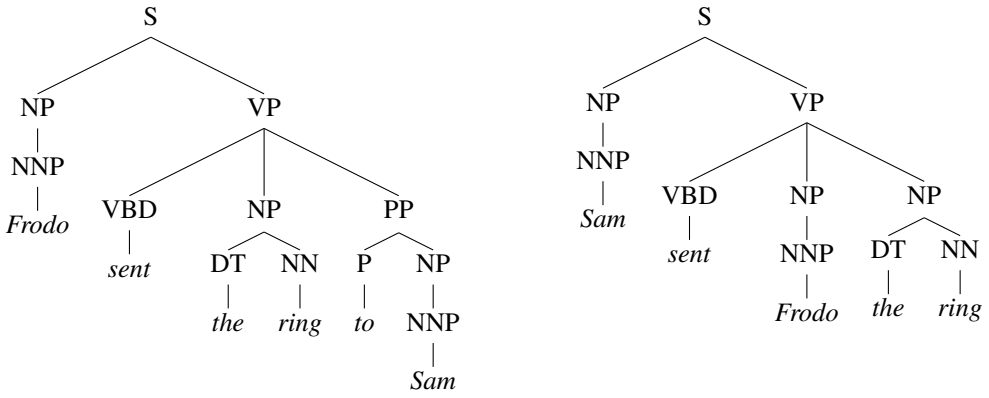
Starting Package

For this exercise we provide four files—`chart.lsp`, `toy.mrg`, `wsj.mrg`, and `test.mrg`. The file `chart.lsp` implements the generalized chart parser, which you are expected to take into use (and optionally improve) during this exercise. The files of type '`.mrg`' are data for training and testing your parser and contain PTB phrase structure trees in the form of Lisp s-expressions.

To find our data files and code for this problem set, please obtain updates from the SVN repository for the course, using the same basic procedure as for previous problem sets. You will find the data and skeleton code inside the new sub-directory '`3b/`' of your SVN checkout. Note that `wsj.mrg` and `test.mrg` are parts of the Penn Treebank, for which UiO holds a license for unlimited research use at the university. Please do not re-distribute these files.

1 Theory: PCFG Maximum Likelihood Estimation

Given the following treebank, list the rules, their counts and the maximum likelihood estimation of their conditional probabilities. Hint: You should find six lexical rules and six syntactic rules.



2 Training a PCFG from Treebank Data

In this section you will define a grammar structure and write a function to train the grammar from a treebank. You will also implement two accessor functions for the grammar that are required by our parser. Read the whole section before starting to code, since your grammar definition will be influenced by the accessor functions we need. Our implementation of the chart parser (provided to you in the file `chart.lsp`) presupposes the interface exactly as specified below, i.e. it will be important that data structures and accessor functions and such work the exact same way we ask for.

(a) The skeleton code contains structures for representing syntactic rules (*rules*) and lexical rules (*lexemes*), and a partially defined *grammar* structure. Using these structures (and augmenting as you see fit), implement the `read-grammar()` function, which takes a treebank file as an argument and returns a *grammar* object. This function should:

- Read in each tree—since the trees are represented as s-expressions, you might the Lisp function `read()` useful here—and
- recursively process each tree, extracting the rules it represents, and recording the counts in the probability slots.
- For lexical rules, add a *lexeme* to the grammar (indexed by the word) if this rule has not been seen before. Otherwise increment the count of the appropriate lexeme.
- For syntactic rules, add a *rule* to the grammar, if this rule has not been seen before and the rule is not a unary recursive rule (e.g. NP $\rightarrow$ NP). If the rule has already been added, just increment the count. Note that in the next step you will write an accessor function that retrieves rules by the first element of the right hand side. This may influence how you store or index your rules.
- Turn the *rule* and *lexeme* counts we just collected into estimates of conditional probabilities, using the standard relative frequency calculations. Store the probabilities as log probabilities.

(b) Our parser requires two accessor functions to retrieve the necessary rules from the grammar efficiently.

- `rules-starting-in()` takes a *category* and a *grammar* and returns a list of all rules in *grammar* which have *category* as the first element of the right hand side (i.e. the first non-terminal after the arrow).

- `get-lexemes()` takes a *word* and a *grammar*, and returns the list of lexemes relevant for *word* in *grammar*.

Implement these accessor functions. Consider if your *grammar* definition could be changed to make these functions more efficient.

To test your grammar implementation, first train a grammar on the `toy.mrg` file.

```
? (setf toy (read-grammar "~/toy.mrg"))
? (rules-starting-in 'NP toy)
→ (#S(RULE :LHS START :RHS (NP) :PROBABILITY -1.5404451)
 #S(RULE :LHS S :RHS (NP VP) :PROBABILITY 0.0))
? (get-lexemes "flies" toy)
→ (#S(LEXEME :CATEGORY VBZ :PROBABILITY -1.9459101)
 #S(LEXEME :CATEGORY NNS :PROBABILITY -2.0794415))
```

Once you are getting the right outputs, try the larger file `wsj.mrg`. This will take longer, but should still finish in under a minute (remember to compile your code first.) If it takes significantly longer, re-work your code to make it more efficient. You should end up with just under 15,000 rules and a little over 44,000 lexemes. If we check the usage of *flies* in this grammar, you should see that The Wall Street Journal rarely reports on insects:

```
? (setf wsj (read-grammar "~/wsj.mrg"))
? (get-lexemes "flies" wsj)
→ (#S(LEXEME :CATEGORY VBZ :PROBABILITY -8.192017))
```

However, the Penn Treebank analyses contain many more rules with a noun phrase as the first element on the right hand side:

```
? (length (rules-starting-in 'NP wsj))
→ 1783
```

3 Parser Evaluation

To determine how well our parser performs (on unseen inputs, i.e. sentences not contained in the training data), implement the ParsEval metric.[1] To compute ParsEval scores, it might be convenient to have an auxiliary function that decomposes one tree into a set of labelled bracketings, where each bracketing $\langle C, i, j \rangle$ captures the syntactic category C assigned to the sub-string of input starting at i and ending at j. Note that it is customary to *not* include PoS tags (i.e. the preterminal nodes of the tree, or categories of lexemes) in ParsEval scores. In case it seems wasteful to you to explicitly enumerate two sets of bracketings only to count overlapping and non-overlapping elements, consider an alternate scheme of computing, for a pair of trees, the relevant counts.

(a) When testing the parser, it will be convenient to have a function available that extracts the leaf nodes of a tree (as a flat list), i.e. the surface string 'underlying' the parse tree. Implement the body of the function `leaves()`, e.g.

```
? (leaves
   '(START
     (S (NP (NNP "Frodo"))
      (VP (VBZ "eats")
       (NP (NN "wasabi") (PP (P "with") (NP (NNS "chopsticks"))))))))
```

[1] Section 14.7 in Jurafsky, D., & J. H. Martin: *Speech and Language Processing*, 2008 (Second Edition).

```
→ ("Frodo" "eats" "wasabi" "with" "chopsticks")
```

(b) Next, implement the function `parseval()`, which takes two trees (represented as s-expressions) as arguments, treeating its first argument as a parser hypothesis and its second argument as the 'gold' standard. The function returns three values, which are in turn: (a) the count of correct bracketings in the parser hypothesis, (b) the total count of nodes in the parser hypothesis, and (c) the total count of nodes in the 'gold-standard' tree. This may well be the first time that you make a function return more than one value; if so, please look up the documentation for the special form `values()` in Common Lisp, which should be used in the body of `parseval()` to return multiple values to the caller.

```
? (parseval
   '(START
     (S (NP (NNP "Frodo"))
      (VP (VBZ "eats")
       (NP (NP (NN "wasabi")) (PP (P "with") (NP (NNS "chopsticks")))))))
   '(START
     (S (NP (NNP "Frodo"))
      (VP (VP (VBZ "eats") (NP (NN "wasabi")))
       (PP (P "with") (NP (NNS "chopsticks")))))))

→ 7
→ 8
→ 8
```

Which labeled bracketing in the parser hypothesis is not correct (with respect to the 'gold' standard), and which bracketing from the 'gold' standard is missing in the parser hypothesis?

(c) Our pre-defined code in `chart.lsp` already provides a function `evaluate()` that takes a '`.mrg`' file and a PCFG as its two parameters, reads a sequence of trees from the file, submits the leaf nodes from each tree to the parser (using the specified grammar and invoking one-best Viterbi decoding from the packed parse forest), and then compares the parsing result to the original tree using the ParsEval metric.

Recall that the combined F_1 measure in ParsEval is the harmonic mean of precision and recall of labelled bracketings. Owed to unknown words in the test data, our parser will likely fail to parse some sentences, i.e. return an empty tree. What should be the contribution of these test inputs to the overall ParsEval scores?

(d) To put parser evaluation figures into perspective, it will be useful to construct a so-called *baseline*, i.e. a score reflecting what would be the result without the Viterbi step. Add an optional parameter `baselinep` to our `evaluate()` function that will make the parser skip the one-best decoding from the forest, i.e. effectively ignore the rule probabilities in the grammar. Instead of computing the top-ranked Viterbi parse, this 'baseline' variant of the evaluation should simply subject the tree that the parser happened to find first to `parseval()` scoring. Does the probabilistic (one-best Viterbi) chart parser improve over this baseline in terms of ParsEval scores?

4 Generalized Chart Parsing and Viterbi Decoding

The code that we provide in `chart.lsp` contains a complete implementation of the generalized chart parser that we will discuss in the next lectures.[2] While it may appear complicated at first sight, most of it maps quite clearly to the algorithm we went through. There are the three structures—*chart*, *edge* and *agenda*—and the main `parse()` function. Look at the `parse()` function and identify our three stages: initialization, main loop, termination. Also look at the `pack-edge()` function and try to understand how it helps us deal with ambiguity.

Following is an example of how to train a (small) grammar and use it to parse one sentence (in pre-tokenized form, represented as a list of strings) at a time:

```
? (setf toy (read-grammar "~/toy.mrg"))
? (pprint (edge-to-tree (parse '("Frodo" "lives") toy)))
→ (START (S (NP (NNP "Frodo")) (VP (VBZ "lives"))))
```

(a) When does a new edge get added to the agenda? For each of the situations where a new edge is pushed on to the agenda, in a few sentences describe the conditions that must be true (if any) and the properties of the new edge.

(b) The `parse()` function produces a packed edge, which could represent multiple trees. Our previous example only had one complete analysis according to our toy grammar. Now we can try an ambiguous input:

```
? (edge-to-tree (parse '("Frodo" "adores" "the" "ring" "in" "Oslo") toy))
→ (START
    (S (NP (NNP "Frodo"))
     (VP (VBZ "adores")
        (NP (DT "the") (NN "ring") (PP (P "in") (NP (NNP "Oslo")))))))
```

Depending on how the rules are stored in your grammar, you may see a different tree, since only the first tree found will be printed. The `viterbi()` function runs the Viterbi algorithm over the packed forest represented by an edge (which can contain packed alternatives) and returns a new edge representing the most likely tree:

```
? (edge-to-tree
   (viterbi (parse '("Frodo" "adores" "the" "ring" "in" "Oslo") toy)))
→ (START
    (S (NP (NNP "Frodo"))
      (VP (VBZ "adores") (NP (DT "the") (NN "ring"))
       (PP (P "in") (NP (NNP "Oslo"))))))
```

Study the `viterbi()` function in our skeleton code. In a few sentences, describe what the function is doing. Compare how the Viterbi algorithm works over a packed forest and an HMM trellis. What is the same? What is different?

[2] We recommend watching the screencast from last year to get started on this part of the assignment: `https://www.youtube.com/watch?v=rTrzuWEacrI`

5 Towards a More Realistic Treebank Parser (Optional)

There are a number of standard optimizations in statistical parsing that our current implementation lacks. It is actually not uncommon for a state-of-the-art parser trained on the PTB to use about one second for parsing a (comparatively complex) input, but nevertheless our current parser probably is one or two orders of magnitudes less efficient than cutting-edge statistical parsers. Furthermore, we still fail to provide any syntactic analysis when there is a single unknown word in our input, where unknown tokens include, for example, all names and numbers not observed in training.

(a) To improve robustness to unknown words, combine the chart parser with our HMM tagger. Recall that the tagger was trained on the exact same data and achieved a tagging accuracy of around $96\,\%$ on the unseen test data from Section 23. One could imagine at least two different ways of using the tagger to preprocess parser inputs: (a) for unknown words, i.e. input tokens for which there is no lexical entry in our grammar, the PoS of the token could be provided by the tagger; this should effectively make it possible to parse all sentences (within a suitable upper bound on input length, as before) from '`test.mrg`'; (b) to further improve parser efficiency, the tagger could be used to reduce lexical ambiguity, where lexical look-up in the grammar would effectively be replaced with the PoS sequence obtained from the most probable path through the HMM. Experiment with both methods of coupling the tagger and parser, and report on your experimental findings.

(b) Another technique to improve the efficiency of the parser is so-called *chart pruning*. The basic idea is, for each chart cell, to discard partial analyses with very low relative probabilities. Typically, this is accomplished by assuming a cut-off beam θ, where for each cell edges whose probability is less than $1/\theta$ of the best-scoring edge in that same cell are discarded. Assume that, at some point during forest construction, for cell $\langle i, j \rangle$ there is an edge e_1 whose probability p_1 is the highest probability for all edges in that cell. The basic intuition in chart pruning is that a new edge e_2 in that same chart cell, with a probability $p_2 < 1/\theta p_1$, is very unlikely to give rise to a tree whose total probability is larger than any tree built using e_1. Think about the assumptions we are making here, and explain why it is possible in principle that a tree containing e_2 might end up with a higher probability than all trees containing e_1.

If you have not done so already, rework the global chart as an abstract data type providing the various types of efficient indexing we need for our chart parser. Revise the probability computations during forest construction, add a suitable level of accounting of per-cell maximum probabilities, and then experiment with chart pruning and various levels of θ.

Happy coding!