601.465/665 — Natural Language Processing
Assignment 4: Parsing

Prof. Kevin Duh* — Fall 2019
Due date: Monday 21 October, 11 a.m.

Last compiled: 2019-10-17 at 15:09:47 UTC

In this assignment, you will build a working CKY parser.

Collaboration: ***You may work in pairs on this assignment,*** as it is programming-intensive and requires some real problem-solving. That is, if you choose, you may collaborate with one partner from the class, handing in a single homework with both your names on it. However:

1. You should do all the work *together*, for example by pair programming. Don't divide it up into "my part" and "your part."
2. Your `README` file should **describe at the top what each of you contributed**, so that we know you shared the work fairly.
3. Your partner **must not be the same partner** as you had for HW3.

In any case, observe academic integrity and never claim any work by third parties as your own.

Materials: All the files you need can be found in `/home/arya/hw-parse` on the `ugrad` machines.

On getting programming help: Same policy as on HW3. (Roughly, feel free to ask anyone for help on how to use the attributes of the programming language and its libraries. However, for issues directly related to NLP or this assignment, you should only ask the course staff or your partner for help.)

How to hand in your written work: Via Gradescope as before. Besides the comments you embed in your source files, put all other notes, documentation, and answers to questions in a `README.pdf` file.

How to test and hand in your programs: Similar to the previous homework. An autograder will test your code on some new sentences. We will post more detailed instructions on Piazza.

1. Parsing is nowadays done as a preliminary step, giving explicit additional information to downstream tasks. Find four papers that use parsing to help a downstream task. Give a one-sentence summary of how parsing is involved in each and why it could help, citing them with URLs and paper titles.

   A good place to look is the Anthology of the Association for Computational Linguistics. ACL is the largest organization for researchers in NLP, and they make all of their journal, conference, and workshop papers available for free. You can restrict Google searches with the term `site:aclweb.org`.

*Some questions adapted from a previous offering by Prof. Jason Eisner.

2. Familiarize yourself with parsing. Try out a context-free parser of English from the literature. You don't have to spend a long time on this question, but experiment by having it parse at least a few sentences.

   ☜2 In your README, discuss what you learned. Some examples:

   - Was there anything interesting or surprising about the style of trees produced by the parser?
   - What were some things that the parser got wrong? What were some hard things that it managed to get right? (I.e., "ordinary" sentences of the kind that it probably saw in training data.)
   - Can you design a grammatical sentence that confuses the parser in a way that you intended, even though the parser is familiar with the words in your sentence? (I.e., "adversarial" sentences that are intended to stump the parser, and may be different from the ones in training data.)

   Hints: The following parsers have online demos, so they are easy to try:

   - Berkeley Parser: `http://tomato.banatao.berkeley.edu:8080/parser/parser.html`
   - Stanford Parser: `http://nlp.stanford.edu:8080/parser/` (for English, Chinese, and Arabic)

   We're not asking you to read the manual for the Penn Treebank.[1] Just tell us what looks "reasonable" or "unreasonable". You can find a simple list of nonterminals here:

   `http://cs.jhu.edu/~jason/465/hw-parse/treebank-notation.pdf`

   You can experiment to figure out the other conventions. For example, if you are not sure whether the parser correctly interpreted a certain phrase as a relative clause, then try parsing a very simple sentence that contains a relative clause. This shows you what structures are "supposed" to be used for relative clauses (since the parser will probably get the simple sentence right). You can also discuss with other students on Piazza.

3. Not all parsers produce parse trees of the sort we've been studying in class (constituency parses). They may produce other kinds of sentence diagrams. One such parsing style is in vogue these days: ☜3 dependency parsing.[2] Try out an online dependency parser. Explain what you notice about it. Also compare and contrast it with constituency parses.

   Please use the spaCy parser: `https://explosion.ai/demos/displacy`. It may be more educational to you if you disable both "Merge XXX" checkboxes beneath the input field.

4. Implement a CKY parser that can be run as

   ```
   python3 parse.py MODE foo.gr foo.sen
   ```

   where:

[1] This is the corpus these parsers were trained on, containing about a million English words in 40,000 sentences from the *Wall Street Journal*.

[2] Nowadays, most dependency parsing algorithms for English leverage neural networks.

- each line of `foo.sen` is either blank (and should be skipped) or contains an input sentence whose words are separated by whitespace
- `foo.gr` is a grammar file in homework 1's format, except that
  - you can assume that the file format is simple and rigid; predictable whitespace and no comments. (See the sample `.gr` files for examples.)
  - the number preceding rule $X \rightarrow YZ$ is the rule's estimated *probability*, $\Pr(X \rightarrow YZ \mid X)$, which is proportional to the number of times it was observed in training data. The probabilities for X rules already sum to 1[3]—whereas in homework 1 you had to divide them by a constant to ensure this.
  - you may assume that the grammar is in Chomsky Normal Form—which permits only rules expanding into two nonterminals or one terminal.
- These files are case-sensitive; for example, `DT` $\rightarrow$ `The` and `DT` $\rightarrow$ `the` have different probabilities.
- `MODE` is one of these:
  - `RECOGNIZER`, which returns whether the sentence can be parsed according to the grammar
  - `BEST-PARSE`, which gives the minimum-weight parse and its weight.
  - `TOTAL-WEIGHT`, which gives the $-\log_2$ of the total probability of a sentence according to the grammar, in bits. (This is different from the sum of weights!) Execution in this mode is called the **inside algorithm**.

We have provided a sample grammar in `flight.gr`. It corresponds to the textbook example of J&M in Chapter 12, except that Suzanna graciously binarized it for you.[4] To sanity-check your implementation, make sure you are able to reconstruct the best parse and the total weight for this sentence:

```
book the dinner flight .
```

should have two parses, namely

```
(ROOT (S (Verb book) (NP (Det the) (Nominal (Nominal dinner) (Noun flight)))) (Punc .))
(ROOT (S (XB (Verb book) (NP (Det the) (Nominal dinner))) (NP flight)) (Punc .))
```

The probabilities for the top two parses above are 2.16×10^{-06}, and 6.075×10^{-07} respectively.

Assuming a three-line `.sen` file, your output should be formatted as follows, depending on execution mode:

RECOGNIZER	BEST-PARSE		TOTAL-WEIGHT
True	3.511	Sentence1-Parse	4.255
False	-	NOPARSE	-
True	6.325	Sentence3-Parse	20.789

- Each parse of a sentence should be preceded by its weight, separated by a TAB.

[3] When you read these in, the sums might not be exactly 1. This is a consequence of floating point arithmetic.
[4] You can obtain the textbook freely from here.

- Print each weight with 3 decimal places.
- The parse should be bracketed in the same format as Assignment 1.
- If there is no parse, print `NOPARSE`.

The *weight* of a rule or a parse means its negative $\log_2$probability, measured in bits. Thus, the minimum-weight parse is the maximum-probability parse. Your parser should convert probabilities to weights as it reads the grammar file. It can work with weights thereafter.[5]

Hint: All three execution modes can share nearly all of their code! The only difference is a few parameters. (Can you think of what these are? If so, you'll save yourself a lot of time.)

Not everything you need to write this parser was covered in detail in class! You will have to work out some of the details. Make sure not to do anything that will make your algorithm take more than $O(n^2)$ space or $O(n^3)$ time. **Please explain briefly (in your `README` file) how you solved the following problems, as well as any others that you deem relevant to justify your runtime:**

(a) You only have $O(1)$ time to add the entry to the appropriate column if it is new, so you must be able to append to the column quickly. (This should be trivial in Python.)

(b) For each entry in the parse chart, you must keep track of that entry's current best parse (in `BEST-PARSE`) and the total weight of that best parse. Note that these values may have to be updated if you find a better parse for that entry.

(c) Storing an entire tree at each cell of the chart would be tremendously inefficient, space-wise. What's a better strategy?

What to hand in: Submit your `parse.py` program (as well as answers to the questions above). (Feeling ambitious? Generate sentences with `randsent`, then parse them with your parser!)

The autograder will test your program on new grammars and sentences that you haven't seen. You should therefore make sure it behaves correctly in all circumstances. Remember, your program must work not only with the sample grammar, but with any grammar file that follows the correct format, no matter how many rules or symbols it contains. So your program cannot hard-code anything about the grammar, except for the start symbol, which is always ROOT.

5. Explain why the complexity of CKY is $O(n^3|G|)$. Your answer should elaborate on why each term is present and necessary, not just give the definition of each variable.

   Propose (or identify from the literature) an improvement. Explain it in depth. (More depth than Wikipedia's summaries—we want to see you comprehend and summarize complex ideas related to what you've learned, not paraphrase Wikipedia.)

6. *Extra credit:* CKY parsers rely on a binarized grammar—one in Chomsky Normal Form. In lecture, we touched on how to binarize a grammar. Now, it's time to try your hand at it. Write a script `binarize.py` that accepts a grammar in the format from Assignment 1, then outputs a binarized form of the grammar.

[5] Alternatively, if you prefer, you can do the conversion at output time instead of at input time. That is, following Appendix G of the previous assignment, your parser can work with probabilities as long as you represent them internally by their logs to prevent underflow. You would then convert log-probabilities to weights only as you print out your results. This is really the same as the previous strategy, up to a factor of $-\log_2$. **Hint:** If you *don't* do it at the end, you may again want to use a library's implementation of log-sum-exp in `TOTAL-WEIGHT`.

Let the binarizer be run as `python3 binarize.py grammar.gr --outfile grammar.grb`; the `outfile` argument is optional. If it is not provided, print the binarized grammar to standard output.

Hint: Binarization is meant to give you another *form* of the same grammar. It should be no more or less expressive than the original. It's easy to accidentally "loosen" the grammar as you binarize it.

Hint: Many of the rules you introduce will have a conditional probability of 1.

Hint: There are several possible binarizations of a single grammar. Finding a minimal binarization (the fewest rules) is NP-hard. We're not asking for the minimal one; any *correct* binarization will do.

✌8 Turn in both your `binarize.py` script and the output of running on a path to be announced soon, saved as `wsj_grammar.grb`.

7. *Extra credit:* Now that you've written a binarized grammar `wsj_grammar.grb`, run it on some test sentences found at a path to be announced soon.

   To help check your binarized grammar (assuming your parser is correct), for the first two sentences in `wallstreet.sen`, their lowest-weighted parses have weights of 34.224 and 104.909 respectively.

✌9 Turn in BEST-PARSE output of your parser on the first five sentences, along with the probability of the best parse. How long did your parser take to process the entire file?

8. *Extra Credit:* English has been the large focus of natural language processing for decades. As such, it is high-quality tools have been developed for several NLP tasks that use English data. But Ethnologue recognizes about 7,000 languages in the world. Nearly all of them lack these tools.

✌10 List some reasons why high-quality NLP tools do not exist in other languages.

   One strategy for combating the lack of tools and annotations in other languages is to **project** annotations generated by English tools onto other languages across **word alignments**; e.g. Yarowsky et al. (2001). These alignments can be found by using bilingual dictionaries (some of the first tools created by linguistic field workers), or learned from parallel text as cross-lingual word embeddings or statistical word alignments. Transferring syntactic information is particularly popular, as in Rasooli and Collins (2017).

   Help a low-resource language out, projecting your information into the language to improve tagging. Write a script `crosslingual.py` that leverages your CKY parser, perhaps via an `import` command. Call your script like this:

   ```
   python3 crosslingual.py mygrammar.gr eng-deu.dict eng.sen deu.sen
   ```

   Feeling ambitious? Find a bilingual dictionary (or create it from sentence-aligned corpora with a freely available tool like `fast_align`). Find some text in the language you care about.

   Feeling stuck? We can give you a toy dictionary and some toy parallel text.

   If your script reads this English sentence:[6]

   The sun shone , having no alternative , on the nothing new .

   and this (questionable) German translation:

[6] From Samuel Beckett's *Murphy*

Die Sonne schien , ohne eine Alternative zu haben , auf die nichts Neues .

then you should parse the English sentence according to your grammar, then use the dictionary to project **preterminals**[7] across any alignments you find, in this two-line format (separating each word or preterminal with a tab, marking missing info with a hyphen):

Die	Sonne	schien	,	ohne	eine	Alternative	zu	haben	,	auf	die	nichts	Neues	.
DET	N	-	PUNCT	PREP	DET	N	-	AUX	,	PREP	DET	-	ADJ	PUNCT

✌11 Tell us which language and resources you chose. Submit `crosslingual.py` via Gradescope. Show us some examples of projected annotations in your writeup.

[7] These are a particular subset of the nonterminals: in a CNF grammar, these are the left-hand sides of the unary rules.