Syntactic Parsing: Project Report

Johannes Gontrum

Department of Linguistics and Philology
Uppsala University

March 25, 2018

1 Project: Multilevel Coarse-to-Fine Parsing for PCFGs

In my project, I successfully implemented the algorithm introduced in Charniak et al. (2006) with the exception of their binarization strategy. Unfortunately, I could not reproduce the reported 10x speed improvement. In fact, the overhead introduced by calculating the inside and outside probabilities slow the parsing several orders of magnitude down compared to a CKY implementation without pruning.

The command line usage of the tool is documented in `README.md`. Please run `make` to install all requirements before and to automatically set up a local virtual environment. The project requires Python 3.6.

1.1 Implementation

1.1.1 Coarse-to-fine mapping

The mapping of coarse to fine symbols is defined in `data/ctf_mapping.yml`. It is processed by the `CtfMapper` class in `ctf_parser/parser/ctf_mapper.py` which creates a dictionary for each level to map coarse to fine symbols and another one to reverse it.

1.1.2 PCFG

I used the PCFG class I implemented for the first assignment with only a few adjustments (`ctf_parser/grammar/pcfg.py`): I introduced additional dictionaries to map the left-hand-sides to rules, as well as the first symbol of a rule and the second one. These data structures are important as they are needed to quickly lookup rules to calculate inside and outside scores later on.

1.1.3 Grammar transformation

A grammar has to be created for every level of granularity. While Charniak et al. (2006) suggest their own binarization strategy, I decided to use a grammar that is already in CNF and transform its rules by replacing every fine symbol by a coarse one. As the pre binarized grammar may contain symbols like `S_PP_:`, I also replace each symbol in the string. I used the grammar generated for the first assignment as the foundation, with the only difference that I replaced the separation symbols for binarizing and removing chain rules by more unique symbols.

The code can be found in `ctf_parser/grammar/transform.py`.

1.1.4 CKY parser

I adapted an optimized version of the CKY parser I wrote for assignment 1 in `ctf_parser/parser/cky_parser.py`. The most noticeable difference is that the class now has an *evaluation function*. Whenever an item has been found that should be entered into the chart, the function is called with the symbol, the span it covers and the used rule. If the function returns true, it is written to the chart, otherwise, it is discarded.

In my implementation the function is passed as a function object. If none is defined, a function is used that always returns true:

```
if evaluation_function is None:
    self.evaluation_function = lambda _: True
else:
    self.evaluation_function = evaluation_function
```

1.1.5 Calculating inside and outside scores

My implementation of the inside and outside score calculation is based on the algorithm in Manning and Schütze (1999). I tried to implement it as straightforward as possible in `ctf_parser/parser/inside_outside_calculator.py`. The main difference is that I use a cache to save already calculated results. In my approach, the inside cache is populated before the actual parsing, as the coarse-to-fine algorithm requires the probability of the sentence, which I assume to be the inside score of the whole sentence under the start symbol. The outside probabilities are calculated lazily.

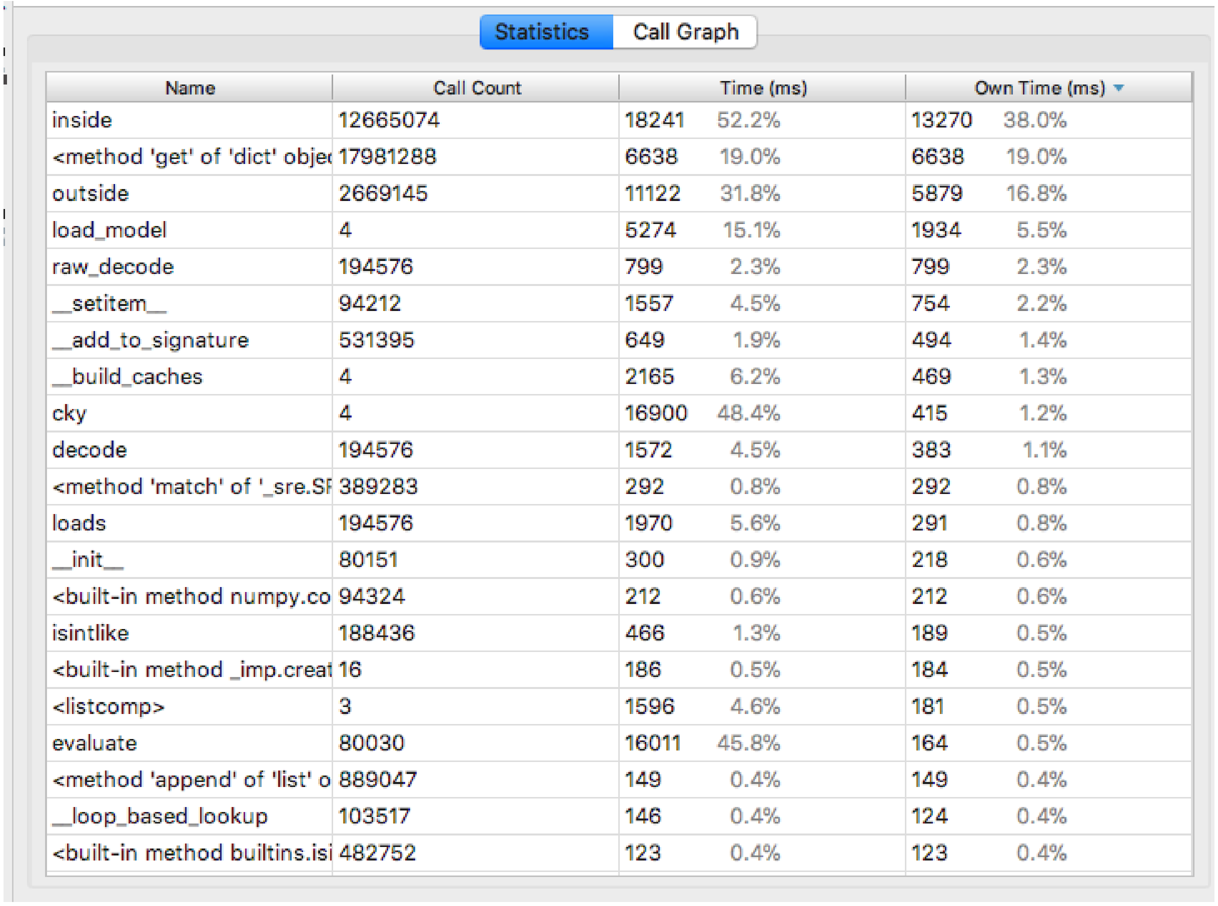
1.1.6 CtF parser

In `ctf_parser/parser/inside_outside_calculator.py` all the previously introduced classes come together: When the parser is initialized, it creates multiple coarse versions of the given grammar and saves them if needed. It then parses the input with the coarsest grammar, calculates the inside and outside scores and creates an evaluation function. Then the input is parsed again with the next finer grammar by passing the evaluation function to the CKY parser.

1.2 Performance

When using a smaller grammar or a short sentence, I could see that my implementation is actually working. In all cases, the vast majority of items were pruned (90-98%) and still, the best parse was returned.

Unfortunately, the calculation of the inside and outside probabilities introduces such a big overhead that it makes parsing long sentences basically impossible. For example, a sentence of length 5 takes 60ms to parse with the standard CKY algorithm. The pruning algorithm, however, increases the time to over 2 seconds. I also believe that this is not only a phenomenon of short sentences, as the calculation time of the inside/outside scores increases cubically with the sentence length.



Name	Call Count	Time (ms)		Own Time (ms) ▾	
inside	12665074	18241	52.2%	13270	38.0%
<method 'get' of 'dict' objec	17981288	6638	19.0%	6638	19.0%
outside	2669145	11122	31.8%	5879	16.8%
load_model	4	5274	15.1%	1934	5.5%
raw_decode	194576	799	2.3%	799	2.3%
__setitem__	94212	1557	4.5%	754	2.2%
__add_to_signature	531395	649	1.9%	494	1.4%
__build_caches	4	2165	6.2%	469	1.3%
cky	4	16900	48.4%	415	1.2%
decode	194576	1572	4.5%	383	1.1%
<method 'match' of '_sre.SF	389283	292	0.8%	292	0.8%
loads	194576	1970	5.6%	291	0.8%
__init__	80151	300	0.9%	218	0.6%
<built-in method numpy.co	94324	212	0.6%	212	0.6%
isintlike	188436	466	1.3%	189	0.5%
<built-in method _imp.creat	16	186	0.5%	184	0.5%
<listcomp>	3	1596	4.6%	181	0.5%
evaluate	80030	16011	45.8%	164	0.5%
<method 'append' of 'list' o	889047	149	0.4%	149	0.4%
__loop_based_lookup	103517	146	0.4%	124	0.4%
<built-in method builtins.isi	482752	123	0.4%	123	0.4%

In fact, I investigated the source if the bad performance using a profiler and discovered that the dictionary lookup for the cache is the bottleneck. Since dictionaries are already highly optimized and implemented in C, I did not find better ways to improve it. I experimented with just-in-time compilation for the inside and outside calculation, functions using *numba* without noticeable effect. The biggest improvement I saw was by rewriting parts of the functions in Cython and compiling it to C++ code. It decreased the parse time by about 25%, which was sadly still worse than the time using CKY alone.

References

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R Shrivaths, Jeremy Moore, Michael Pozar, et al. Multilevel coarse-to-fine pcfg parsing. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 168–175. Association for Computational Linguistics, 2006.

Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.