CS395T Project 2: Shift-Reduce Parsing

Due date: Tuesday, October 17 at 9:30am

In this project you'll implement a shift-reduce parser. First you'll implement a greedy model, then you'll extend that model to be a global model using beam search, with appropriate structured training.

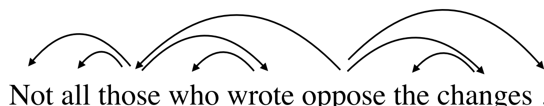
Goal: The primary goal of this assignment is to expose you to different ways of doing approximate inference and learning in intractably large state spaces. You'll learn how to manage complex inferential state spaces at both training time and test time.

Background

Dependency trees are a form of tree-structured syntactic annotation. In this formalism, each word is assigned one parent word – these parent assignments have to form a directed acyclic graph.

The standard dataset used in English parsing is the Penn Treebank. This dataset is annotated with constituency syntax, but several different rule-based dependency conversions have been proposed; we use Stanford dependencies in this project. Here's an example of a sentence from the dataset:

```
1   Not     _   ADV    RB    _   3   neg     _   _
2   all     _   DET    PDT   _   3   predet  _   _
3   those   _   DET    DT    _   6   nsubj   _   _
4   who     _   PRON   WP    _   5   nsubj   _   _
5   wrote   _   VERB   VBD   _   3   rcmod   _   _
6   oppose  _   VERB   VBP   _   0   root    _   _
7   the     _   DET    DT    _   8   det     _   _
8   changes _   NOUN   NNS   _   6   dobj    _   _
9   .       _   PUNCT  .     _   6   punct   _   _
```



Not all those who wrote oppose the changes .

Figure 1: Dependency parse. Arrows are drawn from parents to children.

The columns are (1) sentence index; (2) word; (3) blank; (4) coarse part-of-speech tag; (5) part-of-speech tag; (6) blank; (7) index (1-based) of syntactic parent; (8) dependency label; (9-10) blank.

Dependency parsing is the task of predicting the syntactic parents (column 7) and dependency labels (column 8). In this project, we'll be doing *unlabeled* dependency parsing (unless you choose to explore labeled parsing), which just involves predicting the parents. Moreover, we will be assuming access to *gold* part of speech tags for feature extraction.[1]

Systems for dependency parsing either operate in a graph-based or transition-based paradigm. Graph-based systems search over all trees using dynamic programs and are typically trained in structured logistic regression or max-margin frameworks, similar to CRFs for NER. Transition-based systems make local decisions and process the trees incrementally from left-to-right. In this project, you'll implement both a purely greedy transition-based system as well as a global transition-based system that approximately optimizes the sequence of transitions using beam search, with scores summed over all transitions.

Building a tree from a sequence of local decisions involves a shift-reduce system similar to deterministic parsing algorithms used in compilers. The parser's state is captured by a *stack* of partial dependency trees

[1] In practice, systems either run a part-of-speech tagger to produce predicted part-of-speech tags or do tagging as well as parsing, which can be implemented as an extension of the shift-reduce framework (Bohnet and Nivre, 2012).

and a *buffer* of words waiting to be processed. There are several related systems for modeling trees as decisions in this framework. We use the arc-standard framework; see Nivre (2004) or Huang et al. (2009) for more details on how this works. Code is provided to you that produces a transition sequence from a parse using the lowest stack depth heuristic: attach left children eagerly rather than keeping them on the stack until right children are finished.

Dependency parsing is evaluated according to two heuristics: unlabeled attachment score (UAS) and labeled attachment score (LAS). UAS counts how many words in the evaluation set are assigned the correct parent, and divides this by the total number of words—that is, it's simply the model's per-word accuracy at parent prediction. LAS is the same but also requires the dependency label to be correct; LAS is therefore always lower than UAS. In this project, you're only expected to do unlabeled parsing, so you can return a placeholder dependency label whenever one is required.

Your Task

In this project, you'll be building two related shift-reduce parsers:

1. A purely greedy parser that is implemented via a classifier over shift/reduce decisions
2. A global beam-search parser that searches over the same space but does inference over whole sequences

You will also be extending your project in some way beyond this.

Getting Started

To start, try running

```
python parser.py
```

This simply synthesizes the sequence of transitions for each sentence in the development set and checks that these reconstruct the original tree—they do nearly 100% of the time.

The code in `treedata.py` loads the data into instances of `ParsedSentence`. This class contains both a list of `Token` objects similar to the representation used in Project 1, though this time `Tokens` contain the word, part-of-speech, and a coarse part-of-speech. It also contains a list of `Dependency` objects, indexed according to words in the sentence, with each object containing the index of the parent (0-based, -1 is the root) and the dependency label.

You are provided with code that finds a canonical transition sequence given a syntactic parse. Specifically, the `get_decision_sequence` function takes a `ParsedSentence` object and returns two parallel lists of decisions ("S", "L", or "R") and states (`ParserState` objects). This conversion follows the arc-standard transition system.

`ParserState` objects consist of three data structures: the stack (a list of word indices, top of the stack last), the buffer (a list of word indices, next word first), and the current set of dependencies (a dictionary mapping child word indices to parent word indices). Several convenience methods are included to (a) access various parts of these data structures, (b) check conditions about them, and (c) execute transitions and return new `ParserState` objects. You should familiarize yourself with this representation of parser state.

Part 1: Greedy Shift-Reduce Parsing

In the first part of this project, you will implement a purely greedy version of a shift-reduce parser. This parser is a simple n-way classifier among decisions based on a current `ParserState`. The function

extract_features provides a baseline feature set that looks at the top two words of the stack and the first few words of the buffer. GreedyModel is provided as a convenience class to store your code that can be run from the main parser.py

Applying decisions to the parser state will lead to new parser states. Keep in mind that not every decision will be legal in every state. At the end of decoding, you can use the state.get_dep_objs method to convert the dependencies associated with a ParserState into Dependency objects suitable for evaluation.

You are given more freedom here to decide how to build the greedy classifier: you can choose your classifier (perceptron, structured SVM, logistic regression), optimization techniques, training regimen, and features. The baseline feature set is a bit more impoverished than what was given to you for NER. You'll probably still want to use some optimizations like in Project 1, such as caching features and

The minimum performance requirement on the development set here is 78 UAS. This is a pretty low bar and is achievable with the current feature set using SGD. Our reference implementation on this part got 86.8 UAS using Adagrad and additional features looking at children of items on the stack. With additional feature tuning (see (Zhang and Nivre, 2011) for some ideas of better features), you can get close to 90 UAS. Training on 1000 sentences, you should still get over 70, so you can develop features on a smaller dataset.

Part 2: Global Beam Search and Structured Learning

In this part, rather than scoring isolated classification decisions, you will instead be ranking parses according to the sum of the scores of all decisions made to create those parses. Training should now also take into account that we're doing a global search.

You might want to follow these rough steps:

1. Implement beam search for global decoding. You should be able to run this with the greedy model and get decent results, though increasing the beam size might not cause things to be better (why not?)

2. Implement structured learning with either structured perceptron (non-loss augmented decode) or structured SVM (loss-augmented decode) for this purpose. You can either produce complete parses when computing your gradients or use "early updating" where you update as soon as the gold hypothesis falls off the beam.

To assist you with beam search, we've provided a Beam class in utils.py. This maintains parallel lists of elements and scores, sorted by score. It supports two operations: adding a scored item and returning the list of scored items. Only the k highest-scoring items are kept in the beam; adding an item that already exists in the beam will cause that items score to be max(new score, old score). Our implementation isn't particularly efficient, but it shouldn't be the bottleneck in your code, though you're free to optimize it further!

Your beam search parser should be able to get at least 75 UAS—depending on how well-tuned your greedy model is you may find it hard to beat your greedy model's performance. However, you should observe that increasing the beam size from 1 to higher values gives better results within your structured learning model. Running with a beam size more than 10 or 20 might be prohibitively slow, but see if you can get some small beam sizes (single digits) working well! Try training on smaller training sets for debugging if the full dataset still proves too slow.

Implementation Tips

- One aspect you'll need to deal with is the fact that you'll encounter novel ParserState configurations during training, potentially leading to features you've never seen before. Do you want to expand

the feature vector on-the-fly? Pre-extract a certain set of features (this could be the set of features from part 1) and ignore any features not in this set? How do you want to handle feature caching?

- Feel free to reuse learning code from Project 1 that you find useful!

Part 3: Extensions

These are suggestions for extensions. You also don't need to address each part of each section—these are just suggestions, use your judgment about what you think constitutes a thorough enough extension.

Training Explore additional tradeoffs between beam search and greedy search. What happens if you run the greedy parameters in the beam system at test time? Vice versa? Try a significantly larger or smaller feature set and see if the two parsers differ.

Dynamic Oracle Implement the dynamic oracle of (Goldberg and Nivre, 2012). This is pretty ambitious!

Modeling Implement a different transition systems, such as arc eager (Nivre, 2003; Nivre et al., 2006; Zhang and Clark, 2008), arc hybrid, or arc swift (Qi and Manning, 2017). How do the errors compare? You can also experiment with labeled dependency parsing.

Features Implement additional features from Zhang and Nivre (2011). What kinds of errors do these solve?

Other languages Experiment with dependency parsing in other languages. What's different about these languages? How do errors differ? Arabic, Chinese, and Japanese are included—ask the instructor if there are other languages you're especially interested in.[2]

Submission and Grading

You should submit three files on Canvas:

1. Your code as a zip file or gzipped tar
2. Your best parser's output on the blind test set in CoNLL format. Set the appropriate flag in `parser.py` to produce this.
3. A report of around 2 pages, not including references, though you aren't expected to reference many papers. Your report should list your collaborators, **briefly** restate the core problem and what you are doing, describe relevant details of your implementation, present results from your greedy and beamed parsers as well, discuss these, describe your extension and give results for that, and optionally discuss error cases addressed by your extension or describe how the system could be further improved. Your report should be written in the tone and style of an ACL/NIPS conference paper. Any LaTeX format with reasonably small (1" margins) is fine, including the ACL style files[3] or any other one- or two-column format with equivalent density.

[2] Treebanks available for Arabic, Basque, Bulgarian, Catalan, Chinese, Czech, Danish, Dutch, German, Greek, Hungarian, Italian, Japanese, Portuguese, Slovene, Spanish, Swedish, and Turkish.

[3] Available at `http://acl2017.org/calls/papers/`

Slip Days Per the syllabus, you have seven slip days to use during the semester, so you may choose to use part or all of your budget on this process to turn it in late as needed.

References

Bernd Bohnet and Joakim Nivre. 2012. A Transition-Based System for Joint Part-of-Speech Tagging and Labeled Non-Projective Dependency Parsing. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*.

Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of the International Conference on Computational Linguistics (COLING)*.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (Monolingual) Shift-reduce Parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, and Svetoslav Marinov. 2006. Labeled Pseudo-projective Dependency Parsing with Support Vector Machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*.

Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*.

Joakim Nivre. 2004. Incrementality in Deterministic Dependency Parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*.

Peng Qi and Christopher D. Manning. 2017. Arc-swift: A Novel Transition System for Dependency Parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Yue Zhang and Stephen Clark. 2008. A Tale of Two Parsers: Investigating and Combining Graph-based and Transition-based Dependency Parsing Using Beam-search. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Yue Zhang and Joakim Nivre. 2011. Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the Association for Computational Linguistics*.