Node-Based Induction of Tree-Substitution Grammars

Rose Sloan

May 4, 2016

1 Grammar Induction

Syntactic parsing is a classic problem in natural language processing that can also be a crucial step in solving other problems like question answering and machine translation. Traditionally, parsing is done using probabilistic context-free grammars (PCFGs) or variants thereof, as there are standard efficient methods for parsing with PCFGs and for extracting them from a corpus. However, PCFGs do not accurately represent long-range dependencies in natural language. To take a simple example, in English many determiners can only occur with certain nouns. Determiners like a and another can only occur with singular count nouns, while those can only precede plural nouns, and determiners like more can precede either plural nouns or mass nouns but not singular count nouns. To represent these dependencies with a PCFG, we must have many separate categories for both determiners and nouns, as a simple rule like NP → DT N (where DT stands for determiner and N stands for noun) will overgenerate noun phrases like a water or those cat.

One formalism that makes representing long-range dependencies much simpler is probabilistic tree-substitution grammars (PTSGs) (Cohn et al., 2009). While every CFG rule can be seen as one level of a syntactic parse tree, and thus a subtree of height 2, TSG rules can be any subtree of a syntactic tree, thus allowing them to concisely represent dependencies that involve more than one level of syntactic structure. For example, a TSG for generating noun phrases can represent what types of nouns common determiners can precede using rules like the following:

(1) NP
    |     |     |
    DT N NN
    | a N

(1) NP
    |     |     |
    DT N NNS
    | those NNS

(1) NP
    |     |     |
    DT N
    | the

(In this example, N stands for any noun, NN specifically refers to singular count nouns, and NNS refers to plural nouns.)
Given a parsed corpus, the task of extracting a PCFG is straightforward. Obtaining the CFG rules is simply a matter of looking at each non-leaf node in the parsed corpus and including a rule that goes from its nonterminal label to its children. Then, the rule probabilities that maximize the likelihood for the training corpus are the rough frequency estimates, so the probability for the rule $\alpha \rightarrow \beta$ is the number of times $\alpha \rightarrow \beta$ appears in the training corpus divided by the number of times $\alpha$ appears in the training corpus.

However, unlike PCFGs, given a parsed corpus, it is non-obvious how to extract a PTSG that accurately represents the corpus. It is unclear which subtrees should be extracted from a training corpus to induce a PTSG. For example, while the rules displayed in (1) capture important dependencies, rules like those in (2) do not, either being too general or too specific to represent the dependencies we are looking at.

```
(2)  NP  NP  NP
      DT N  DT N  DT N
       those the platypus
```

Thus, we would like to induce a model that assigns higher probabilities to the rules in (1) than to the rules in (2).

One approach used to induce PTSGs is known as data-oriented parsing (Bod & Scha, 1996). This approach attempts to capture every possible rule that could have generated the training corpus. Unsurprisingly, this results in extremely large grammars that are inefficient to use for parsing and, when trained on a sufficiently large corpus, too large to even store explicitly. (For example, when trained on the Penn treebank, DOP produces a grammar with about $10^{48}$ rules.) Because of these problems, more recent DOP approaches attempt to limit the rules to some salient subset of the subtrees appearing in the corpus. One such approach, known as double-DOP, does this by obtaining rules by intersecting each pair of trees in the training set, thus obtaining the set of elementary trees that are the largest common subtree of some pair of trees in the training corpus (Sangati & Zuidema, 2011). Then, all CFG rules are added to the grammar, and the probabilities can be estimated using a number of approaches, although Sangati and Zuidema found that rough frequency estimates outperform more complicated methods. In this case, the resulting grammar is still large (about a million rules when trained on the Penn treebank), but it is small enough that it can be stored explicitly and can be used for parsing.

In this project, I present a novel approach for inducing a PTSG from a trained corpus. Unlike the data-oriented parsing approaches, my approach of node-based induction attempts to optimize the set of elementary trees in the induced
PTSG.

2 Node-Based Induction

Node-based induction attempts to optimize the set of elementary trees in the induced PTSG by optimizing the set of substitution nodes in the training corpus. I do this by assigning a probability to each internal node in the corpus, which I train over many iterations of my algorithm.

Initially, this probability is set to the same value for each node in the training set. Experimentally, I determined that an initial value of 0.55 for each of the node probabilities tended to give the best results, although small changes in this initialization parameter have little effect on the final results, as later steps of the algorithm mitigate any initial bias.

\begin{algorithm}
\textbf{Algorithm 1} Initialize node probabilities
\begin{algorithmic}
\FORALL{n in nodes}
\STATE $p(n) \leftarrow 0.55$
\ENDFOR
\end{algorithmic}
\end{algorithm}

Once every node has a probability, an intermediate PTSG is induced. For every tree in the training set, a set of elementary trees is generated by randomly selecting substitution nodes, based on their probabilities, and decomposing accordingly. For example, if we start with the tree in (3) and randomly select the DT and NN nodes to be substitution nodes and the N node to not be a substitution node, we get the trees in (4).

(3)
\begin{center}
\begin{tikzpicture}
  \node (NP) {NP};
  \node (DT) [below left of=NP] {DT};
  \node (N) [below right of=NP] {N};
  \draw (NP) -- (DT) -- (N);
  \node (a) [below of=DT] {a};
  \node (NN) [below of=N] {NN};
  \draw (DT) -- (a) -- (NN);
  \node (cat) [below of=NN] {cat};
  \draw (N) -- (cat);
\end{tikzpicture}
\end{center}

(4)
\begin{center}
\begin{tikzpicture}
  \node (NP) {NP};
  \node (DT) [below left of=NP] {DT};
  \node (N) [below right of=NP] {N};
  \draw (NP) -- (DT) -- (N);
  \node (a) [below of=DT] {a};
  \node (NN) [below of=N] {NN};
  \draw (DT) -- (a) -- (NN);
  \node (cat) [below of=NN] {cat};
  \draw (N) -- (cat);
\end{tikzpicture}
\end{center}

Once we have extracted all these elementary trees, we set their probabilities simply using a relative frequency estimate. That is, we count the number of times that that elementary tree appears in our set of decomposed trees, and we divide that by the number of elementary trees with the same root node.
Algorithm 2 Obtain intermediate grammar

for all \( t \) in \( \text{training} \) do
  for all \( n \) in \( \text{nodes}[t] \) do
    if \( \text{random.random()} < p(n) \) then
      add \( n \) to \( \text{subnodes} \)
    end if
  end for
  add \( \text{decompose}(t, \text{subnodes}) \) to \( \text{elementary} \)
end for
for all \( e \) in \( \text{elementary} \) do
  add \( (e, \text{count}(e)/\text{count}(\text{root}(e))) \) to \( \text{int.gram} \)
end for

Having obtained this intermediate grammar, we can now use it to parse the training set. This is done using a standard CKY parser, using an alternate set of operations that represent motion within and between elementary trees, instead of simply using the “combine” operation used in CKY parsing using CFGs. These operations are “move-unary”, which corresponds to moving up one level within an elementary tree while still covering the same lexical items, “move-binary”, which corresponds to combining two children by moving up within an elementary tree to their parent node (analogous the the “combine” operation used with CFGs), and “substitute”, which represents remaining on the same node, but treating it as a substitution node and looking at a new potential tree it could be in (Kallmeyer, 2010).

As with a standard CKY parser, the complexity of this CKY parser is cubic in the length of the sentence and linear in the size of the grammar. In this case, I only tested the algorithm on statements that were generally about 2 or 3 (and no longer than 4 or 5) words long, so the grammar size term had the largest effect on the running time. Given no unary-branching nodes, the maximum number of elementary trees that can come from a given tree in the corpus is one less than the number of leaves of that tree. In this case, pre-lexical nodes are unary-branching, as are nodes labeled N, but those tend to be the only unary-branching internal nodes, making the number of elementary trees produced by a given tree proportional to the length of the phrase that tree represents. As these trees are small, this means that the size of the grammar can be considered linear in the size of the training set. This means that the time it takes to parse the entire training set is quadratic in the size of the training set. This can be somewhat sped up by removing any rules containing lexical items that do not appear in the phrase being parsed before parsing (Schabes et al., 1988). However, especially with larger training sets, this is the slowest step of the algorithm.

This parsing will produce one or more parses for each tree in the training set. Based on these parses, we can then update the probabilities of each node in the tree. Each of the parses has an associated probability, generated from the
probabilities of the rules in the parse. Additionally, we assign a weight to each parse to favor parses in which about half of the internal nodes are substitution nodes in order to avoid both CFG rules and rules that memorize entire trees in the training set. Specifically, the weight of a parse is \( \frac{s}{t} \) where \( s \) is the number of substitution nodes in a parse and \( t \) is the total number of internal nodes (i.e. the number of potential substitution nodes). Then, for a node \( n \), we can compare the probabilities and weights of the parses in which \( n \) is a substitution node and those where it is not. If \( p(x) \) is the probability of a parse, \( w(x) \) is the weight of a parse, \( S \) is the set of all parses in which \( n \) is a substitution node, and \( T \) is the set of all parses for the tree that \( n \) appears in, we can compute \( p_{\text{int}} \) using the following formula:

\[
p_{\text{int}}(n) = \frac{\sum_{x \in S} w(x)p(x)}{\sum_{x \in T} w(x)p(x)}
\]

Finally, in order to prevent one iteration from having too strong an effect on the final result, we take a weighted average of \( p_{\text{int}} \) and the node’s old probability to get the new probability for the node. Experimentally, I determined that a weight of 0.4 for \( p_{\text{int}} \) tended to strike the best balance between not overweighting any one iteration while still converging after 100 to 200 iterations. (Lower weights for \( p_{\text{int}} \) led to the algorithm taking over 300 iterations to finish.)

Additionally, if the difference between \( p_{\text{int}} \) and the old value of \( p(n) \) is less than 0.05, we say that it has converged. Then, if the number of “converged” nodes divided by the total number of internal nodes in all trees in the training set is over 0.95, training comes to an end and the node probabilities set at the end of the last round of training are used to sample the final grammar. Otherwise, we sample a new intermediate grammar using the updated probabilities and perform another iteration of training.

Once training comes to an end, we use the final node probabilities to get a final grammar. Using Algorithm 2, the same algorithm used to sample the intermediate grammar, we decompose each tree in the training set 100 times. Based on the elementary trees produced through these 100 decompositions, we then use relative frequency estimates to get the final rules and their probabilities.

Then, once we have determined the rules and probabilities for the final PTSG, we parse each rule in this PTSG using the other rules of the PTSG. If there is a parse for the rule made up of smaller rules and if the probability of this parse is greater than the probability of the larger rule, the rule is determined to be superfluous. Superfluous rules are removed from the grammar, and the probabilities are renormalized.

Finally, in order to account for unknown words, for each part of speech appearing in the training set, a tree with height 1 with a root labeled with the part of speech tag and with one leaf node labeled “unk” (short for “unknown”). The probability of these rules was set according to the number of types and tokens
Algorithm 3 Update node probabilities

for all $t$ in training do
    parses ← parse($t$, int_gram)
    for all $n$ in nodes[$t$] do
        for all $par$ in parses do
            if $n$ is a substitution node in $par$ then
                nprob ← nprob + $p(par)w(par)$
            end if
            totprob ← totprob + $p(par)w(par)$
        end for
        $p_{int} ← \frac{nprob}{totprob}$
        if $|p_{int} - p(n)| < 0.05$ then
            converged ← converged + 1
        end if
        $p(n) ← 0.4p_{int} + 0.6p(n)$
    end for
if converged > 0.95count(internal nodes ∈ training) then
    generate final grammar
else
    get new intermediate grammar and repeat
end if

for the part of speech so that a part of speech with many distinct lexical items, such as count nouns, would have a relatively high probability of unknown words compared to a part of speech with relatively few distinct lexical items, such as determiners. Specifically, the probability of the rule $POS \rightarrow unk$ was set to:

$$\frac{\text{types}(POS)}{\text{types}(POS) + \text{tokens}(POS)}$$

After adding these rules, the probabilities of all other rules whose roots were part of speech tags were renormalized.

3 Methods

The trees used for training and test sets were taken from the Adam portion of the Pearl-Sprouse corpus, a parsed version of the child-directed portions of the Brown subcorpus from CHILDES (Pearl & Sprouse, 2012). Additionally, in order to allow the algorithm to distinguish between mass and count nouns, the NN label (the POS tag for singular nouns) corresponding to any mass noun was manually replaced by an NNM label. Similarly, to allow the algorithm to have
rules applicable to all nouns, a node labeled simply N was inserted immediately above any node labeled NN, NNM, or NNS (the POS tag for plural nouns).

Furthermore, as the algorithm makes use of a CKY parser, any tree in the corpus which was not binary branching was modified to become right-branching. If an inserted node’s children were both labeled N, it was labeled with N, so that the algorithm would treat compound nouns the same way as other nouns, and similarly, if the first child was labeled JJ (the POS tag for adjectives) and the second was labeled N, the inserted node was labeled N, as adjective-noun pairs distribute similarly to nouns in this dataset. All other inserted nodes were labeled by concatenating the labels of their children.

4000 noun phrases were then extracted from this modified corpus. None of these noun phrases included smaller internal noun phrases, so as to allow the algorithm to focus on dependencies between determiners and nouns, and all of them included at least one node labeled N (so as to eliminate single pronouns from the data set). They were also selected so that at least 30% of them contained mass nouns. 3200 of these nouns were randomly chosen to be the training set. The remaining 800 became the test set. Furthermore, every lexical item in the test set that did not appear in the training set was replaced with the word “unk” so that it could be properly parsed by the induced grammar. Larger training sets were not attempted, partially because doing so would have required manually labeling more mass nouns and partially because, as mentioned earlier, each iteration of training is quadratic in the size of the training set, so expanding the training set significantly increases run time.

Using this data set, I also induced a few grammars to use as baselines. Setting all node probabilities to 1 and using Algorithm 2 to decompose the training set resulted in a PCFG, with probabilities set using relative frequency estimates. Adding the same rules for unknown words to this grammar as were used in the node-based PTSG results in a PCFG that can be used to parse the test set. In fact, to account for the fact that some potential tree structures might not be captured by the node-based grammar or by some of the other baselines, when testing all other grammars, each tree was parsed both with the grammar being tested and with the PCFG. The probability of the tree was then calculated to be a weighted average of the two probabilities, with the PCFG weighted at 0.05.

Similarly, setting all node probabilities to 0 and using Algorithm 2 to “decompose” the training set produced a grammar whose rules are simply the trees in the training set, with probabilities again set using relative frequency estimates (or, in other words, probabilities were set based on the number of times each tree appeared in the training set). Finally, the third baseline was induced by setting all probabilities to the initial value of 0.55 and then using Algorithm 2 to decompose the grammar 100 times, as is done at the end of training for node-based induction. This is a grammar obtained simply by sampling without first training the node probabilities.

Lastly, Sangati and Zuidema’s code for double-DOP was run on training set to
obtain their set of fragments and CFG rules with counts. Using these counts, probabilities for each rule were obtained using relative frequency estimates. Then the same rules for unknown lexical items with the same probabilities as in the induced grammar were added, and the probabilities were renormalized. In this case, because the process of extracting the elementary trees for double-DOP includes extracting all possible CFG rules from the test set, I did not also parse with the PCFG and average the probabilities when testing with the grammar induced through double-DOP.

4 Results

Table 1 displays the results for how node-based induction compares to the baselines with a training set of size 3200 and a test set of size 793. (Initially, the test set was of size 800, but 7 noun phrases were removed because they contained structures that were unseen in the training set and thus could not be parsed by any of the grammars.) The numbers provided here were obtained by summing the log probabilities of the best parses for each tree in the data set. (In every case except for the PCFG baseline, these probabilities were also computed by taking a weighted average of the probability of the best parse with the chosen model and the best parse with a PCFG, as explained in the methods section). Thus, larger (i.e. less negative) numbers correspond to higher probabilities and therefore better results.

| Method       | Training | Test  | Grammar Size |
|--------------|----------|-------|--------------|
| Node-Based   | -25263   | -6770 | 1359         |
| PCFG         | -30905   | -7091 | 990          |
| Full Trees   | -22280   | -6814 | 1572         |
| Sampling     | -30135   | -7266 | 1721         |
| Double-DOP   | -28882   | -7032 | 2404         |

Table 1: Log probabilities of training and test sets on different grammars

As these results show, node-based induction assigned a higher probability to the training set than any of the other approaches. Additionally, while (unsurprisingly) the grammar that simply memorizes the training set assigns the highest probability to the training set, the node-based grammar outperforms all the other grammars on the training set. Furthermore, apart from the PCFG (which has less variety in possible rules), the node-based grammar is the smallest grammar induced. This is important, because, as previously mentioned the CKY parser is linear in the size of the grammar and cubic in the length of the sentence parsed, so, especially as sentences get longer, large grammars can lead to very slow parse times.

In order to gauge whether the phrases the induced grammar generated were grammatical, all 1442 noun phrases of the format “determiner noun” were ex-
tracted from the training set and, for each determiner that appeared more than 5 times, the probability distribution over different types of nouns (count nouns, mass nouns, and occurring with that determiner was computed. These distributions are shown in table 2. Then, 1442 noun phrases of the form “determiner noun” were generated using the PTSG induced with node-based induction, and the same distributions were computed, shown in table 3. The same was done for the PCFG.

| Determiner | Count | Mass | Plural |
|------------|-------|------|--------|
| a          | 0.983 | 0.015| 0.002  |
| an         | 0.952 | 0.048| 0.000  |
| another    | 0.714 | 0.286| 0.000  |
| any        | 0.048 | 0.714| 0.238  |
| no         | 0.571 | 0.286| 0.143  |
| some       | 0.000 | 0.913| 0.087  |
| that       | 0.857 | 0.143| 0.000  |
| the        | 0.712 | 0.230| 0.058  |
| this       | 0.960 | 0.040| 0.000  |

Table 2: Probability distributions of noun types co-occurring with common determiners in the training set

| Determiner | Count | Mass | Plural |
|------------|-------|------|--------|
| a          | 0.82  | 0.15 | 0.03   |
| an         | 0.71  | 0.18 | 0.11   |
| any        | 0.32  | 0.53 | 0.16   |
| some       | 0.18  | 0.79 | 0.03   |
| that       | 0.85  | 0.10 | 0.05   |
| the        | 0.74  | 0.21 | 0.05   |
| this       | 0.86  | 0.03 | 0.10   |

Table 3: Probability distributions of noun types co-occurring with common determiners in noun phrases generated by the PTSG

In order to determine how well these generated distributions mirrored the original, the Kullback-Leibler (K-L) divergence was computed between the distributions generated from each of the PTSG induced through node-based induction and the PCFG and the true distribution from the training corpus (Coeurjolly et al., 2007). K-L divergence is a metric used to compute the information lost when using a distribution $Q$ to approximate or model a “true” distribution $P$ and is calculated as

$$\sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In this case $P$ is the empirical distribution from the training set and $Q$ is the
distribution from the generated phrases. In order to avoid zero probabilities, add-one smoothing was used when computing the probability distributions that the divergences were calculated on. The values of the K-L divergences are shown in table 4.

| Determiner | Node-Based | PCFG |
|------------|------------|------|
| a          | 0.13       | 0.40 |
| an         | 0.11       | 0.80 |
| any        | 0.16       | 0.52 |
| some       | 0.18       | 1.03 |
| that       | 0.03       | 0.09 |
| the        | 0.00       | 0.03 |
| this       | 0.05       | 0.33 |

Table 4: K-L divergences of noun phrases generated by the node-based PTSG and the PCFG compared to the empirical distribution

The distributions produced by the node-based PTSG are not as strongly skewed as the empirical distributions, where many of the probabilities are over 0.9, but they do reflect dependencies between determiners and noun types. Furthermore, the K-L divergences are much smaller than those generated by the PCFG, which can only reflect the relative frequencies of each determiner or noun type in the training set separately and cannot capture the likelihood that a noun type will co-occur with a given determiner.

In summary, node-based induction produces grammars that capture many of the dependencies between nouns and determiners. These grammars can more accurately represent the probabilities of potential parses for non-recursive noun phrases than traditional PCFG-based approaches or grammars induced from DOP-based approaches. Future work could adapt this algorithm to work with larger grammatical structures, including full sentences, and it could then be used to induce grammars that more accurately model language and generate more accurate parses.

References

Bird, Steven, Ewan Klein & Edward Loper. 2009. Natural language processing with python. O’Reilly Media, Inc.

Bod, Rens & Remko Scha. 1996. Data-oriented language processing: An overview. Computing Research Repository.

Coeurjolly, J-F, Rémy Drouilhet & J-F Robineau. 2007. Normalized information-based divergences. Problems of Information Transmission.

Cohn, Trevor, Sharon Goldwater & Phil Blunsom. 2009. Inducing compact but accurate tree-substitution grammars. In Proceedings of human language
technologies: The 2009 annual conference of the North American chapter of the Association for Computational Linguistics, 548–556. Association for Computational Linguistics.

Kallmeyer, Laura. 2010. Parsing beyond context-free grammars. Springer Science & Business Media.

Pearl, Lisa & Jon Sprouse. 2012. Computational models of acquisition for islands. Experimental syntax and island effects 109–131.

Sangati, Federico & Willem Zuidema. 2011. Accurate parsing with compact tree-substitution grammars: Double-DOP. In Proceedings of the conference on empirical methods in natural language processing, 84–95. Association for Computational Linguistics.

Schabes, Yves, Anne Abeille & Aravind K. Joshi. 1988. Parsing strategies with ‘lexicalized’ grammars: Application to tree adjoining grammars. In Proceedings of the 12th conference on computational linguistics - volume 2, 578–583.

A Code

The code used to implement and test node-based induction is included with this project. This code is all written in Python and makes heavy use of the Natural Language Toolkit for Python (Bird et al., 2009). The file probparsing.py contains the core methods for implementing and testing node-based induction. The file dptesting.py extracts the training and test sets from the corpus brown-adam-tagged (which is included here and has all mass nouns labeled as such) and calls methods from probparsing to induce and test a grammar from these sets. It can also take the flag –w, which causes it to additionally output the training set and test set to files so that the results can be replicated. Additionally, the flag –r followed by a filename reads in and tests the grammar in that file instead of inducing a new grammar. It assumes that there are files named training.txt and test.txt in the current directory (used to induce the baselines) containing the training and test sets that produced this grammar. Finally, the file doubledop.py takes the filename of a “fragments and CFG rules” file output from Sangati and Zuidema’s code (which is available at http://homepages.inf.ed.ac.uk/fsangati/EMNLP11_Sangati_Zuidema.tar.gz) as a command line argument, turns it into a usable PTSG, and tests it. Again, this assumes that the relevant training and test sets are in the current directory as training.txt and test.txt.