Automatic PCFG Generator

Todd Layton, Anastasia Uryasheva

December 13, 2012

1 INTRODUCTION

Here will be an introduction

2 BACKGROUND

The theoretical background of the project is based on the article "Automatic learning of context-free grammar" written by T.Chen, C.Tseng, C.Chen. The article tells about the problem of learning context-free grammar from a corpus. The solution for the problem is investigated based on the notion of minimum description length of the corpus. Basic problem is to find a set of rules, which will derive original sentences in the best way. Article tells about the way of creating CFG from corpus of sentences and suggests a cost function as a measure of the quality of resulting CFG. With the help of cost function, goal is rewritten as: to find a set of rules that can derive original language with the minimum cost.

2.1 COST FUNCTION

Cost function depends on the CFG description of the corpus. There are two types of cost functions:

1. Rules for giving words their POS-tags.

   For rules consisting of a non-terminal symbol on the left-hand side and a string of symbols on the right-hand side the cost of a rule is the number of bits needed to represent the left-hand side and right-hand side. For example, if we have the rule:

$$A-> \beta \tag{2.1}$$

then the cost function for this rule will be:

$$C_R = (1 + |\beta|) log(|\Sigma|) \tag{2.2}$$

where Σ is the symbol set and $|\Sigma|$ is the number of symbols in Σ .

2. Rules for deriving sentences.

To derive sentence W we need a sequence of rules: $S \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow W$. This sequence of rules always starts with one of S-derivation rules: $S \Rightarrow \alpha$. This step results in a derived string α. If there is no non-terminal symbols in α , we are done with the derivation. Otherwise, we expand the left-most non-terminal symbol, say X, in α by one of its derivation bodies. The process continues until there is no non-terminal symbols in the derived string, which will be the sentence W at that point. So if we have a set of rules: R1(S) : S âĘŠ âĂę , âĂę , R1(X) : X âĘŠ âĂę , âĂę and a sentence for derivation W then the cost for rules to derive W will be:

$$C_D = \sum_{k=1}^{m} log(|R(s_k)|) \tag{2.3}$$

where m is the number of rules in the derivation sequence, s_k is the non-terminal symbol for the kth derivation and $|R(s_k)|$ is the number of rules in the CFG using s_k as the left-hand side.

Combining (2.2) and (2.3), the total cost is

$$C = \sum_{i=1}^{p} C_R(i) + \sum_{j=1}^{q} C_D(j) = \sum_{i=1}^{p} n_i log(|\Sigma|) + \sum_{j=1}^{q} \sum_{k=1}^{m_j} log(|R(s_k)|) \tag{2.4}$$

2.2 SPECIAL CASES

Additionally, the article investigates two special cases:

1. Exhaustive CFG

Exhaustive CFG is a CFG that uses every distinct sentence in the corpus as a direct derivation body of the start symbol S. The number of symbols for a rule is simply the number of words of the corresponding sentense n_w, plus 1 (for the start symbol S), and $|\Sigma|$ is the vocabulary size $|V|$ of the corpus plus 1 (for the start symbol). The rule cost is:

$$C_R = nlog(|\Sigma|) = (n_w + 1)log(|V| + 1) \tag{2.5}$$

In this case, each sentence is derived from S in one step, by specifying the correct one out of the $|R(S)|$ rules. Thus the derivation cost for a sentence is:

$$C_D = log|R(S)| \tag{2.6}$$

Total cost for that case is:

$$C = \sum_{i=1}^{|R(S)|} C_R(i) + \sum_{j=1}^{q} C_D(j) = \sum_{i=1}^{|R(S)|} (n_w(i) + 1)log(|V| + 1) + qlog|R(S)| \tag{2.7}$$

2. Recursive CFG

   Recursive CFG is a CFG that uses recursive derivation for S: S âĘŠ AS where the non-terminal A can be expanded to be any word in the vocabulary. The rule cost in this case is:

   $$C_R = n log|\Sigma| \tag{2.8}$$

   where n can be 1, 2 or 3 depending on rule The derivation cost in this case is:

   $$C_D = n_w(1 + log|V|) + 1 \tag{2.9}$$

   Total cost in this case:

   $$C = \sum_{i=1}^{2+|V|} n_i log|\Sigma| + \sum_{j=1}^{q} C_D(j) = (4 + 2|V|)log(|V| + 2) + \sum_{j=1}^{q} (n_w(j)(1 + log|V|) + 1) \tag{2.10}$$

   The exhaustive CFG is too restricted in the sense that it covers only those sentences seen in the learning corpus. The recursive CFG is too broad in the sense that it covers all sentences including the non-sense ones. The goal is to strike a balance between these two extremes.

3 DESIGN

This program uses a POS-tagged training corpus to generate a PCFG, through an iterative process of production rule creation and modification by transforming an initial trivial grammar into a more generalized one.

3.1 SIMPLIFYING ASSUMPTIONS

This PCFG generator program make a couple basic assumptions to reduce the complexity of the problem, allowing it to focus more effectively on the core of the automated generation concept.

3.1.1 PRE-TAGGED TRAINING CORPUS

Like many statistical parsers, the PCFGs generated by this program only describe production rules down to the level of parts of speech; the grammars' terminal symbols are not specific words, but rather POS tags. This means that the important training input to the generator is not a set of sentences in themselves, but a set of POS-tag sequences corresponding to a set of sentences. While this could be achieved by incorporating a separate tagging preparation step into the program itself, we chose to instead reduce the scope of the generator to exclude tagging; instead, the program's training corpus input is a set of POS-tagged sentences, which can have been tagged in any way the user sees fit.

3.1.2 Flat-Probability Training Corpus

The program calculates the relative frequency of various features in the grammar, as it is generated, on the assumption that each of the tagged sentences in the training corpus has the same flat probability in the source language (namely, a probability of 1 over the size-in-sentences of the corpus). While this reduces the amount of input data necessary, it also removed the possibility of the generation process taking into account the true relative probabilities of the training corpus's content, which could have a non-negligible impact on the resulting PCFG when training on larger corpora.

3.2 Initial Grammar

The program uses the input training corpus to create a exhaustive grammar as a starting point for the rule modification. For each POS-tagged sentence in the training corpus, a corresponding production rule is added to the grammar. The left-hand side of this rule is `ROOT`, the right-hand side is the sequence of POS tags constituting that sentence, and the probability is 1 over the total size, in sentences, of the corpus.

3.3 Iteration

As described above, exhaustive grammars are maximally specific to their generating corpus, so once the initial grammar is created, the program repeatedly looks for modifications to make in order to produce a grammar which is not so closely fitted to the training corpus. Generation of the PCFG is complete once there are no remaining modifications to be made. These modifications consist of two types of grammar transformations: 2-gram expansion and rule joining.

3.3.1 2-gram Expansion

The program finds the most frequently-occurring symbol 2-gram in the training corpus which is not already the right-hand side of a production rule in the grammar. A new nonterminal symbol is created, specified as an expansion of that 2-gram's symbols and with a unique numeric identifier, and each instance of that 2-gram is replaced with that nonterminal. An additional production rule is added to the grammar, of which the left-hand side is the new nonterminal, the right-hand side is the 2-gram, and the probability is 1. This step does not change the functional structure of the grammar, but rather decomposed it into a larger number of simpler rules in order to enable rule joining.

3.3.2 Rule Joining

Before each 2-gram expansion step, the program checks whether any multiple production rules in the corpus, with more than one symbol in the right-hand side, consists of the exact same right-hand side except for the symbols at one particular index (for example, the third symbol in each rule's right-hand side). If such rules are found, then a new nonterminal symbol is created, specified as a rule joining of the symbols being replaced and with a unique

numeric identifier, and the symbol at that index in each rule's right-hand side is replaced with that nonterminal. Then, for each of those rules, a new rule is added to the grammar, of which the left-hand side is the new nonterminal, the right-hand side is the symbol which that nonterminal replaced in that rule, and the probability is the relative frequency of the right-hand side symbol. Unlike 2-gram expansion, rule joining actually changes the functional structure of the grammar, allowing for symbol sequence which did not previously fit.

4 IMPLEMENTATION

The `pcfggen` module's primary access point is the `generate_pcfg()` function.

4.1 CONTROL FLOW

Algorithm 1 Pseudocode program flow of `pcfggen` module's `generate_pcfg()` function
---
 function GENERATE-PCFG(S)   ▷ S: training corpus of POS-tagged sentences
  G = Initial-Grammar(S)
  loop
   while G has valid rule joining J **do**
    G = Join-Rules(G, J)
   if G has valid most-frequent 2-gram expansion E **then**
    G = Expand-2-Gram(G, E)
   else
    return G

4.2 EXAMPLE

The effects of these two types of grammar modification are best illustrated with a simple example. Consider a toy corpus of three sentences, corresponding to three POS tag sequences:
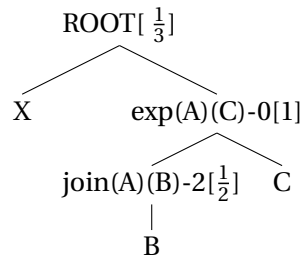
(a) `A C B C`

(b) `X A C`

(c) `Y B C`

Given this corpus as the training input, the program would perform the following steps:

1. Create the initial grammar:
   (a) `ROOT →A C B C [1/3]`
   (b) `ROOT →X A C [1/3]`
   (c) `ROOT →B C Y [1/3]`

2. Check for rule joining, but no production rules are sufficiently similar.

3. Expand the 2-gram `A C`:

   (a) `ROOT →exp(A)(C)-0 B C [1/3]`

   (b) `ROOT →X exp(A)(C)-0 [1/3]`

   (c) `ROOT →B C Y [1/3]`

   (d) `exp(A)(C)-0 →A C [1]`

4. Check for rule joining, but no production rules are sufficiently similar.

5. Expand the 2-gram `B C`:

   (a) `ROOT →exp(A)(C)-0 exp(B)(C)-1 [1/3]`

   (b) `ROOT →X exp(A)(C)-0 [1/3]`

   (c) `ROOT →exp(B)(C)-0 Y [1/3]`

   (d) `exp(A)(C) →A C [1]`

   (e) `exp(B)(C) →B C [1]`

6. Join `A C` and `B C`:

   (a) `ROOT →exp(A)(C)-0 exp(B)(C)-1 [1/3]`

   (b) `ROOT →X exp(A)(C)-0 [1/3]`

   (c) `ROOT →exp(B)(C)-1 Y [1/3]`

   (d) `exp(A)(C)-0 →join(A)(B)-2 C [1]`

   (e) `exp(B)(C)-1 →join(A)(B)-2 C [1]`

   (f) `join(A)(B)-2 →A [1/2]`

   (g) `join(A)(B)-2 →B [1/2]`

7. Check for rule joining, but no production rules are sufficiently similar.

8. Check for 2-gram expansion, but no unexpanded 2-grams remain.

9. Return the grammar.

The resulting generated PCFG is able to parse not only the training sentences, but also other sentences of a similar structure. For example, the sentence `X B C` would be parsed as follows:

$$
\begin{array}{c}
\text{ROOT}[\tfrac{1}{3}] \\
\diagup \quad \diagdown \\
\text{X} \qquad \text{exp(A)(C)-0[1]} \\
\diagup \quad \diagdown \\
\text{join(A)(B)-2}[\tfrac{1}{2}] \quad \text{C} \\
| \\
\text{B}
\end{array}
$$

Similarly, the generated grammar is capable of randomly producing sentences which were not present in the training corpus. `X B C` is a possible random-sentence output of this PCFG, with a probability of $\frac{1}{3} \cdot 1 \cdot \frac{1}{2} = \frac{1}{6}$.

5 RESULTS

Here will be a table with results and results' description

5.1 POTENTIAL IMPROVEMENTS

This implementation of a PCFG generation program could be improves significantly in several ways. At the moment, all valid 2-gram expansions and rule joinings which exists in the training corpus are executed, and the generation process only terminates once all such modifications have been completed. Allowing for the possibility of valid but statistically-negligible modification would improve the system's behavior on large training sets. Similarly, it is very possible for a generated grammar to include groups of production rules which could be simplified down to an equivalent form with fewer rules (as can be seen in the above example). This would limit the growth of the overall size of the generated PCFG's rule set, and might even create additional valid rule joinings of which the generation program could make use.

Another major improvement would be the incorporation of word and sentences probabilities into the training data. If the generation program could be run on raw sentences rather than on POS-tag sequences, and could account for the relative probabilities of each POS mapping to each word, then the generated grammar would be able to parse raw sentences directly, which would improve the accuracy of its results and massively increase its ability to generate natural-sounding random sentences. Similarly, incorporating information about the relative frequency of the training corpus's sentences would prevent less-grammatical input data from distorting the overall structure of the generated PCFG.

6 CONCLUSION

Here will be a conclusion