Report TP2 - Construction of a Parser

Philippe Ganshof
ENS Paris-Saclay
Cachan 94230
philippe.ganshof@hotmail.com

1. Introduction

We built in this practical work a parser for parsing sentences using the *SEQUOIA tree bank v6.0* as our training dataset. Our system is divided in three main parts namely the PCFG, OOV and the CYK class. The role of the first model is to extract the grammar from the training corpus. OOV handles out-of-vocabulary words and we finally use the probabilistic CYK algorithm for parsing.

2. Extraction of PCFG

The first module *pcfg.py* enables us in a first time to extract all the rules and words from the training corpus and assign to each of them a probability. These probabilities are stored in two dictionaries *grammar* (G) and *inv_lexicon* (L). More precisely, for each rule $X \to Y_1...Y_n$:

$$\mathrm{G}[X][Y_1...Y_n] = \mathbb{P}(X \to Y_1...Y_n) = \frac{C(X \to Y_1...Y_n)}{C(X)}$$

and for each word W with tag X:

$$L[W][X] = \mathbb{P}(X \to W) = \frac{C(W)}{C(X \to W)}$$

where C defines the number of times a rule or a word appears in the training corpus. For that purpose, we designed a module *pcfg_tree.py* to apply this process to a single sentence using a tree structure in order to walk around in the bracket form parsed sentence. It is then used by the function *build_pcfg* that builds G and L given a corpus of parsed sentences. Since the CYK only handles binary grammars, the module also contains a function *to_chomsky_form* to transform G in Chomsky normal form. The transformation consists of two steps. We first replace all rules with more than 2 children with a chain rule, carried out by the function *to_binary_rules*. More precisely, each rule $X \to Y_1...Y_n$ with $n > 2$ becomes:

$$X \to Y_1 \ Y_2|Y_3...Y_n, \quad Y_2|Y_3...Y_n \to Y_3 \ Y_4|Y_5...Y_n$$

$$, ..., \ Y_{n-2}|Y_{n-1} \to Y_{n-1} \ Y_n$$

where $Y_k|Y_{k+1}...Y_n$, $k = 1, ..., n-2$ are new symbols and we assign a similar probability to all of them. We then remove all unites rules $X \to Y$ and create new rules by assigning X to each child Z of Y. We update the probability of the new rule $X \to Z$ by multiplying both $\mathbb{P}(X \to Y)$ and $\mathbb{P}(Y \to Z)$ which makes sense in terms of occurence. If Y is terminal, we replace all rules containing the tag X by the new symbol $X\&Y$.

These artificial symbols are defined in this particular way to be able to retrieve the initial tags after we are done parsing. Finally, the *pcfg* model also extracts additional information about the corpus such as the number of tags or words and their frequencies that turns out to be useful for computing the CYK algorithm.

3. OOV Words

To be able to parse sentences containing words that do not exists in the training corpus, we replace out-of-vocabulary words by the *closest* word in the lexicon when using the CYK algorithm. The *oov.py* module rightly takes care of finding this *closest* word in the lexicon. But how do we choose it? In general, we would like the word in the lexicon to have the same tag (or list of tags) and a similar meaning than the out-of-vocabulary word.

For that purpose, we used two different metrics. We first implemented the Levenshtein distance defining the minimum cost to transform a word M into P by carrying out only elementary operations such as insertion or deletion. Since there is a high chance for a random word to have Levenshtein distance less or equal than 2 with a word in the lexicon, we only consider candidates within edit distance 2. To distinguish candidates in the lexicon having the same Levenshtein distance from the word, we simply choose the word with the highest frequency. This is handled by *closest_word_levi*.

We also used the *Polyglott embedding lexicon for French* to compare embedding similarity between words. Since not all words in the lexicon have an embedding, we created an embedding matrix only for those in the *Polyglott Lexicon* with the function *build_embeddings_lexicon*.

Finally, given a query, the function *closest_word* returns the word in the lexicon with the closest embedding if the query is contained in the *Polyglott Lexicon* and otherwise returns the word with the closest Levenshtein distance. In this setup, the Levenshtein distance is here to backup the embedding similarity in case the word is not contained in the *Polyglott Lexicon*. This might define the simplest combination of both metrics but offers garanties.

4. Parsing with CYK

The previous parts has enabled us to apply the CYK algorithm to a random sentence and the module *cyk.py* combines the previous parts for parsing the sentence.

We implemented the probabilistic CYK algorithm that returns the most probable parsing tree among those that starts with *SENT*, named *CYK_algorithm*. The function takes as input a sentence only containing words from the lexicon. It consists of recovering the probability of all possible substrings of the input string and taking the parsing tree with maximum probability. The algorithm is based on dynamic programming with the following recursion formula:

$$P[s,l,X] = \max_{0 \leqslant c \leqslant l-1, X \to Y\ Z} P[X \to Y\ Z]$$

$$\cdot P[s,c,Y] \cdot P[s+c+1, l-c-1, Z]$$

where $P[s,l,X]$ denotes the probability of the most likely parsed subsentence of length $l+1 \in [1,n]$, starting at word position $s \in [0, n-l-1]$ with tag X.

Since we add artificial tags to the PCFG when transforming it into Chomsky normal form, we also implemented a function for retrieving the initial tags. The algorithm takes as input a parsed sentence (in bracket form) and clean all tags using a tree structure and the particular construction of the artificial tags. Finally, *parse* combines everything and parse the sentence in the desired form.

5. Results and Discussion

We trained on the first 80% of the sequoia corpus and tested on the last 10% containing 310 sentences. We obtained a tagging accuracy of 91.2% and our parser was able to parse 249 sentences. Our parser took approximatively 10 hours to parse all sentences and we believe that it can be significantly improved by optimising the efficiency of the code. Most of the sentences for which our parser was giving an obvious wrong parsing tree were essentially short sentences with proper names or dates. This is not surprising as proper names usually have a large Levenshtein distance from each other and so makes the tagging more difficult. To resolve this problem, we could build a specific model for detecting proper names.

Words having a similar embedding in the *Polyglott Lexicon* usually have a similar meaning but it does not necessarily means that they have the same tag. Words having both a small Levenshtein distance and a similar embedding have a much higher chance of having the same tag in general. Hence, another idea to improve the accuracy would be in the function *closest_word* not only to consider the word with the closest embedding but a set of candidates from both metrics and distinguish them by combining both metrics in a smart way.

Finally, there does exists other algorithms than the CYK in the literature. For instance the LR [1] is much faster and could lead to better results.

References

[1] A. V. Aho and S. C. Johnson Lr parsing.ACM Comput. Surv., 6(2):99?124, June 1974.

[2] Rami Al-Rfou, Bryan Perozzi, and Steven Skiena Distributed word representations for multilingualnlp. InProceedings of the Seventeenth Conference on Computational Natural Language Learning, pages 183?192,Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[3] T. Kasami An efficient recognition and syntax analysis algorithm for context-free languages. Technical ReportAFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA?, 1965.

[4] V. I. Levenshtein Binary Codes Capable of Correcting Deletions, Insertions and Reversals.Soviet Physics-Doklady, 10:707, February 1966.