HM2: NLP

Gabriel Fedrigo Barcik - gabriel.fedrigo-barcik@polytechnique.edu

March 10, 2020

The goal of this assignment is to develop a basic probabilistic parser for French that is based on the CYK algorithm and the PCFG model and that is robust to unknown words.

1 Design choices

1.1 Probabilistic context-free grammars (PCFG)

In order to create the probabilistic context-free grammar, I used the NLTK library to pre-processed the input string which represents a tree of a parsed sentence in the following way:

1. Remove functional labels to avoid sparsity issues
2. Create NLTK Tree from string, removing empty top bracketing
3. Collapse the unary transformations, collapsing POS tags
4. Transform tree to Chomsky Normal Form (CNF)

After that, with our training corpus of the first 80% of the data, we extract all the grammatical productions and calculate the probability of the binary productions, unary productions and POS tag to words productions.

The way we store this probability table is special to optimize the CYK algorithm. This choice will become clear when discussing the CYK algorithm in the corresponding section.

Three Python dictionaries were built to store those probabilities, more precisely we use dictionary of dictionaries.

The first one contains the transition probability related to POS tag to word. As keys, to every word of the lexicon, we store a Python dictionary containing all the TAGs that can form this word with the corresponding probability.

The second one contains the unary transition probability of grammatical token to POS tag. For a rule A to B, we store B as key and as a value a dictionary containing all A's that can form B with the corresponding probability.

The third dictionary contains the binary transition probability of grammatical token to grammatical tokens. For a rule A to B C, we store as key (B, C), and as value a dictionary containing all A's that can form B, C with the corresponding probability.

1.2 Out of vocabulary module (OOV)

There are two distances implemented in the OOV module, the Damerau-Levenshtein (optimal string alignment distance) distance and the cosine similarity of word embeddings using the polyglot dataset.

When parsing a sentence we check if a word in the sentence is in the training corpus, if yes, we know its grammatical production probabilities and we can continue the CYK algorithm. If it is not in the training corpus, we check if this word is in the polyglot vocabulary, if yes, we use its embedding to find the closest embedding of the words of our training corpus and we continue the CYK algorithm replacing the OOV word with the one found. In the case where the word is not in the training corpus neither in the polyglot vocabulary, we compute the Damerau-Levenshtein of this word and all words of polyglot vocabulary, and pick randomly one word from the set of the smallest Damerau-Levenshtein distance. Then we find the nearest neighbour of this embedding with the words of the training corpus to replace the OOV word with a word of the training corpus and continue the algorithm.

1.3 CYK algorithm

MY CYK implementation handles unary transitions and follows the convention of interleaved indices. Two data structures are used to store the scores of a given tree and how to retrieve a solution once the maximal parse tree is found, they are the *scores* dictionary and the *backpointers* dictionary.

The *scores* dictionary stores as key a (begin, end) tuple, which represent the range of words analyzed, and as value a dictionary of all tokens that can be generated at this level.

The *backpointers* stores the argmax path to achieve a given token in a position defined by (i, j). It is a dictionary mapping tuples of the form (i, j, token) to another tuple or pair of tuple, depending if the (i, j, token) originates an unary production or binary production. The way we construct the parse tree is, we start from the tuple (0, nb_words, 'SENT') and keep going backwards with the backpointers map recursively until we find words, the leaves of the parse tree.

The algorithm works as follows, first initialize the scores dictionary with the terminal tokens, loop through the words of the sentence, handling OOV words with the policy described in the OOV section, adding for all POS tags that can form this word the corresponding probability in the *scores* dictionary, saving the necessary information in the backpointers dictionary.

After filling the scores dictionary, we update the probabilities with unary production rules that can possible augment the probability to achieve a given token.

Then, use a dynamic-programming bottom-up strategy to fill the parser diamond. For every begin and end position we split it in all possible possible positions, giving a cubic complexity in the number of words in a sentence. For a given (begin, split), (split, end) ranges we loop through the tokens associated, say B's for the first one and C's for the second one. Then we check if a token A can create B and C using a binary production, if yes or the probability to create A is higher than the one stored we store it in *scores* and save the path in *backpointers*. For every (begin, split, end) triples we update the scores probability with unary productions which can possible increase the probability of a given token.

The most interesting part of using dictionary of dictionary to store the probability table is to do fast look-ups in the inner loop of the bottom-up fill strategy. We loop through a smaller number of tokens compared to all the grammatical tokens of the dataset, $O(G^3)$, where G is the number of grammatical tokens.

Since the size of the grammatical production tokens is fixed, the time complexity of the CYK algorithm is $O(n^3)$ in the number of words in a sentence.

2 Results

The split used was 80% of the first lines to the training set, corresponding to 2480 sentences, and 10% of the last lines to test set, corresponding to 310 sentences.

The results when applying our CYK algorithm with PCFG to our training set is fairly good. The recall achieved is 0.8581 and the precision 0.8351. As all the phrases belong to the grammatical rules all sentences are successfully parsed.

The results on the test set, last 310 phrases: we are not able to parse 92 sentences, 29% of the sentences, mainly due to bad OOV replacement, therefore the algorithm builds an inconsistent sentence where the start token 'SENT' cannot be found. An easy way to evaluate the performance is to evaluate the part-of-speech accuracy only, i.e. via the percentage of tokens for which your parser chooses the correct part-of-speech tag. For the successfully parsed sentences the mean accuracy of the POS tags is 0.7053.

I would improve my system by enlarging the training corpus to capture more grammatical rules and better estimate probabilities and also define a better OOV policy to replace OOV words with words from the lexicon. Using a language model, such as the bi-gram model, can help to predict a word given the last seen word, we could incorporate these to our model, weighting the closest neighbours for example. To better handle misspelled words we can use better heuristics, to find the intended word instead of randomly choosing one close word as I did.

Some OOV problems are related to replace a noun of a person to another word that breaks the grammatical structure of the sentence.