NLP Homework 4 (MVA 2017/2018)

Rémi Lespinet

For this project I've chosen to implement everything from scratch in Python 3. I only imported `math` for the logarithm and `math.inf`, numpy for representing multidimensional arrays more easily and `re` for regex parsing.

1 How it works

1. **Tree parsing** that transforms the given corpus to a tree form (stored in a custom implementation of a Tree called ParseTree) through the function (file `TreeParsing.py`).

2. **Functional label removal** I do a simple traversal of the tree and remove the functional labels using a regex (it cut the symbol when a non word letter is used) `NP-SUJ` → `NP` (file `TreeProcessing`)

3. **CNF Normalization** I implemented a function to format the tree in a form that can be used by the CYK algorithm. As suggested in *Jurafsky and Martin*, I implemented a partial normalization of the Tree which keeps unit production rules (of the form $A \to B$) where B is a non terminal symbol. In this regard, it is not a full CNF normalization, because the accepted rules are

$$\begin{array}{rcl} A & \to & B\ C \\ A & \to & B \\ A & \to & w \end{array}$$

   The main reason for this choice is that we can handle the unit production rules directly using a modified version of the CYK algorithm, and it is easier to reconstruct the tree from the CNF form this way. (I've also implemented the reverse operation of the CNF normalization called (file `TreeProcessing`)).

4. **Tree annotation** I do an annotation pass that provides informations on every node such as the minimum distance to a leaf. I use this information to discriminate lexicon rules (minimum distance to a leaf equal to 1) from grammar rules.

5. **Calculating probabilities (PCFG)** I use empirical probability estimation on the rules of the corpus for the grammar and the lexicon.

$$P(\alpha \Rightarrow \beta | \alpha) = \frac{\#(\alpha \Rightarrow \beta)}{\sum_\gamma \#(\alpha \Rightarrow \gamma)}$$

6. **Probabilistic CYK**

   Handling Unit production rules : I've implemented a CYK algorithm that handles unit production rules. It works by precomputing a unit production table, which given a symbol V gives a list of all the symbols that can generate this symbol using unit prouction rules, more formally, it retrieves

$$\mathcal{C}_V = \{U \mid U \underset{UPR}{\Longrightarrow}^* V\}$$

and also keep tracks of the unit production rule derivation that gave the highest probability. Each time we compute a list of possible symbols for a subsequence of word $w_i \dots w_j$ (which is correspond to $A_{i,j}$ (where A the table that we fill by dynamic programming), we can then expand the list of symbols by a simple lookup in the precomputed table (of course we need to mutiply the probabilities accordingly).

Attempt to speed up computations : I also use a precomputed reverse lookup for grammar rules and lexicon to speed up computation in some cases. In fact to compute the possible rules for the sequence $w_i, \dots w_j$ with split k, I loop over the symbols in $A_{i,k}$ and $A_{k+1,j}$ and see if there's a corresponding grammar rule by reverse lookup. If we don't have a lot of possible symbols in each cell, this is a huge improvement, because we don't need to run through all the grammar each time, but as the number of possibilities increases, the computation gets higher, which is not desirable. Appendix B presents a way to speed up the computations using sparse matrix operations.

Unknown word handling : At first, I tried to add an `UNK` symbol and set all words w as being generated by `UNK` $\Rightarrow w$ if they appear only one time in the corpus at train time (and at test time if they are unknown). This worked, but given that the corpus is really small, a lot of words became labeled as unknown.

I also tried setting all the probabilities $P(A \Rightarrow w)$ equals when encountering a new word, this has the advantage of forcing the system to guess the symbols (this is the default handling)

2 Results and limitations

Visualization : In order to visualize the generated trees, I've implemented a function that dumps trees in the Graphviz format. These can be easily converted to *SVGs*. Visualization of the examples in this section are provided in appendix A. Unknown words (words that are absent from the corpus) are marked in red. The first sentence is taken from the corpus. The first tree sentences are successfuly parsed by the algorithm, and the 2 last sentences present cases where the produced parse tree is not correct.

```
1) En_fait , ce témoignage était entièrement faux .        (Figure 1)
2) S' il faut schtroumfer , je schtoumferais !             (Figure 2)
3) J'ai relâché mon carapuce car il était trop nul !       (Figure 3)
4) C' est comme ça qu' il a battu Freezer !                (Figure 4)
5) Je m' appelle Bond , James Bond .                       (Figure 5)
```

Accuracy evaluation :for the evaluation of the model, I split the corpus in two part, 80% were used for training and 20% for the evaluation. I did the experiment in two settings : in the first one, I also use the lexicon rules of the evaluation set to estimates the probabilities $\mathbb{P}(R \to w)$, which means that the CYK algorithm does not encounter unknown words (of course the other rules have been trained only with the training set).

Lexicon model : With this version I obtain an accuracy of 68.61%. If use only the lexicon rules of the training set, this drops to 30.94%. The reason is that CYK alone does not make use of a lexicon model (this would be a model which would, given a word w, output the probability $P(R \to w)$ for each PoS R). This shows that having a good lexicon model is very important for this task. As explained previously, when an unknown word is encountered, I set the probability equal for each rule, this has a lot of problems :

1. First there are PoS symbols for such as COORD for which we know the list of word whose $P(R \to w)$ is non zero (these are close class types)

2. Second we do not make use the characteristics of the word, such as prefixes, suffixes or capitalization, which are valuable to determine the probability $P(R \to w)$.

To solve this problem we need to train a model that, given chatacteristics of a word predicts the probability $P(R \to w)$ for each rule (it would also be a good idea to use a larger dictionary to reduce the number of unknown words, in the corpus, for example the pronoun *tu* does not appear).

This explains why the algorithm fails to recognize the sentences 4 and 5, for the sentence 4, This is not suprising, because the algorithm uses the potential PoS tagging of the nearby words to determine the best rule, and can then infer the PoS tagging of the unknown word. When there's too much consecutive unknown word as in the sentence 4, it is clear that the algorithm will not handle them properly, which explains why it chooses a structure like in the sentence `C’ est comme ça qu’ il attrape le ballon` considering that `battu` is a `DET` and `Freezer` a `NC`. The sentence 5 is also interesting because `Bond` is repeated twice and misclassified only the first time (for the same reason).

Absence of context : Another problem, as mentioned in *Jurafsky and Martin*, is that PCFG does not make use of context, which means that the rule used to derive a symbol will be independant from neighboring word, and this is actually a poor asumption, for example, in the following rule

$$\texttt{SENT} \to \texttt{NP} \quad \texttt{VN} \quad \texttt{NP}$$

the NP symbols will probably have different derivation probabilities, and PCFG can't model this dependance as is. One solution to address this issue is to split the symbols by using additional context information such as the parent symbol. This solution would requires a larger corpus.

A Visualization of the trees

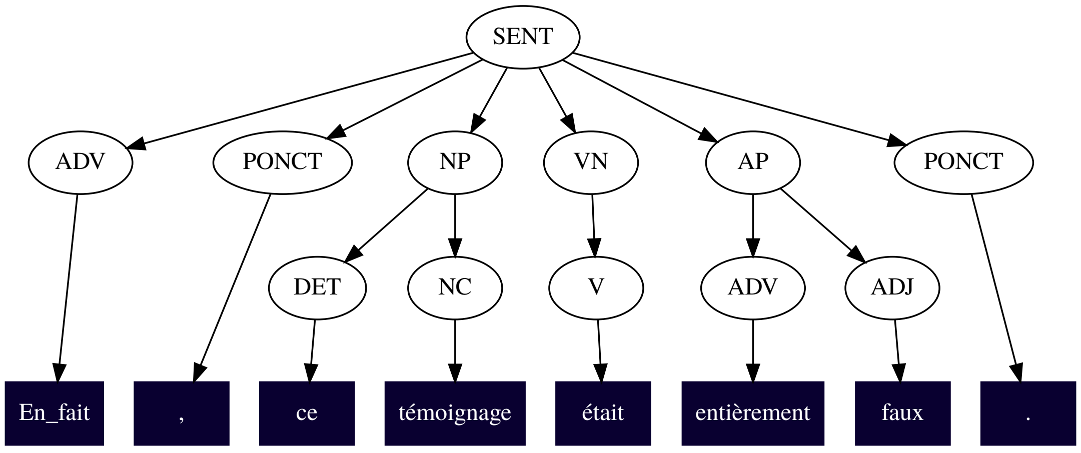

Figure 1: Parse tree for the sentence 1

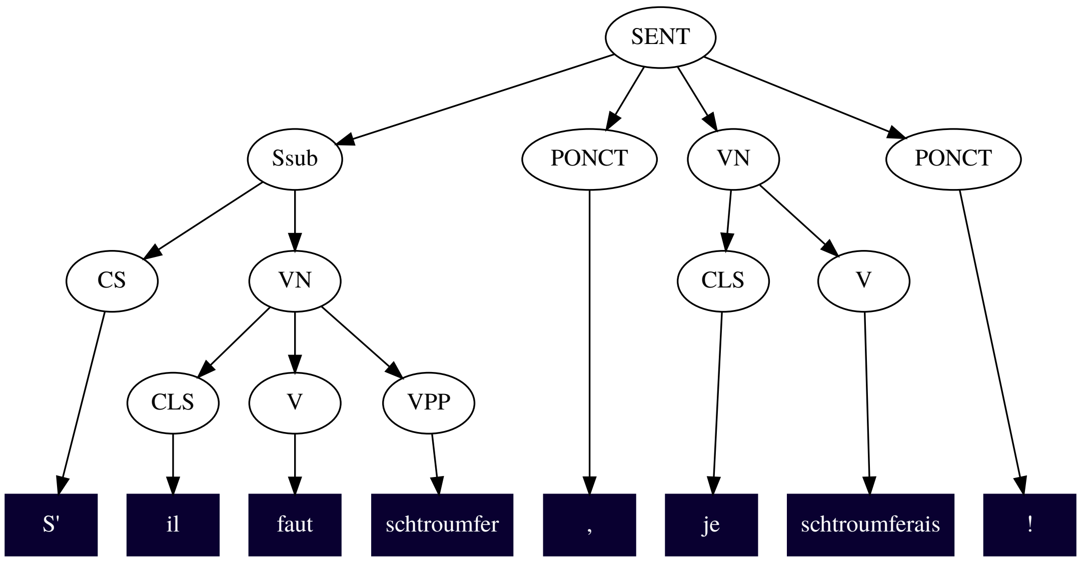

Figure 2: Parse tree for the sentence 2

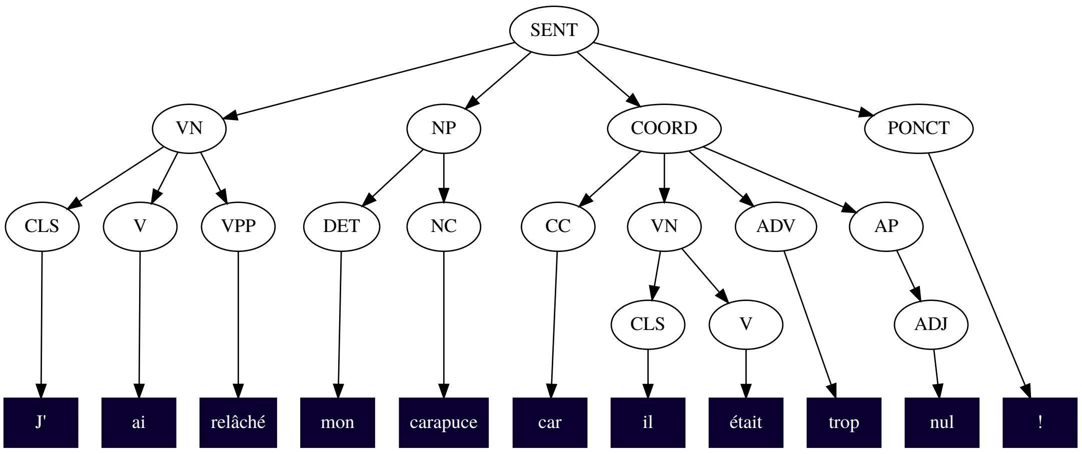

Figure 3: Parse tree for the sentence 3

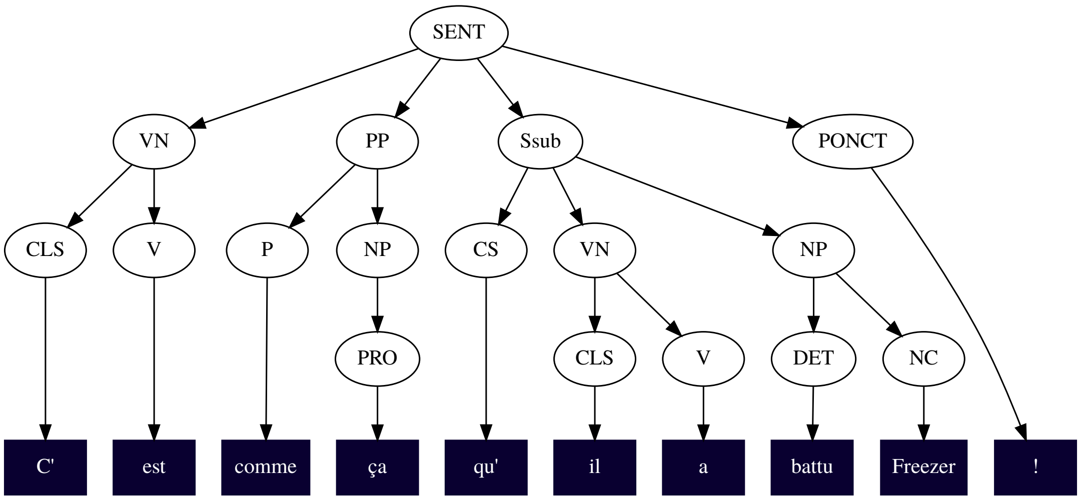

Figure 4: Parse tree for the sentence 4

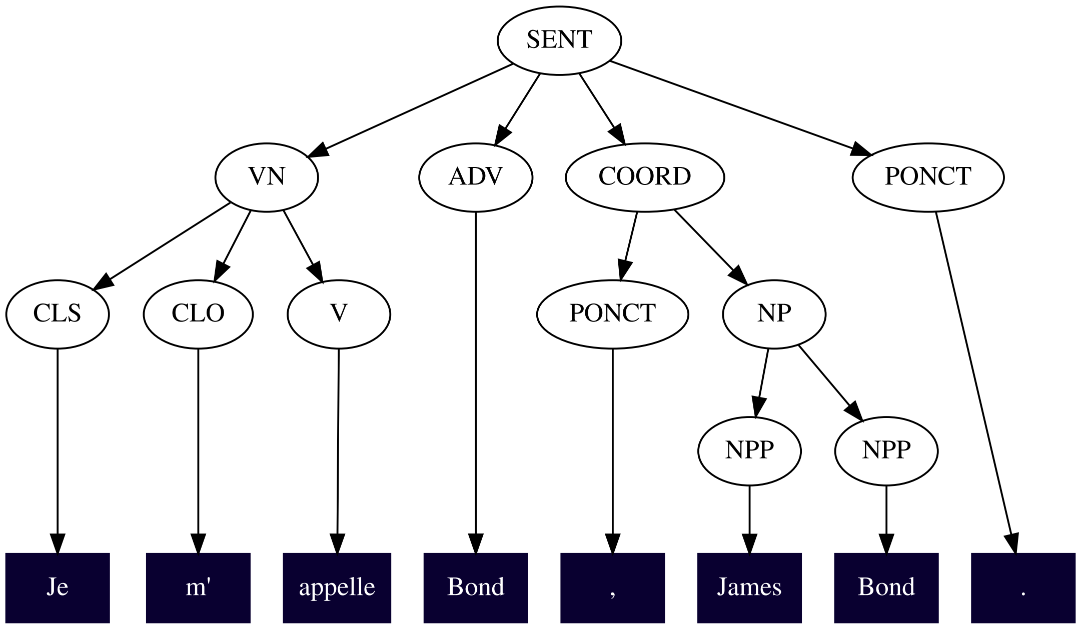

Figure 5: Parse tree for the sentence 5

B CYK using Sparse Tensor operations

Given M non terminal symbols $R_0 \dots R_{m-1}$, we can represent our grammar by the rules $R_i \to \cdot$ by a tensor M whose entry (i, j, k) represents the probability of the rule $R_i \to R_j\ R_k$. Of course a lot of rules are 0 and we can we can benefit from sparse tensor operations to speed up the computations.

As mentioned in *Jurafsky and Martin*, it is better to do a partial CNF normalization which keep unit production rules and handle these rules directly in the CYK algorithm. In fact we can handle them very efficiently by precomputing a unit production table.

B.1 Precomputing the unit production table

The matrix A that we compute by dynamic programming is of size $N \times N \times M$. $A_{i,j}$ is a vector of size M whose k^{th} entry represents the probability that the sequence of words $w_i \dots w_j$ has been generated by symbol k.

Now for each non null entry k of the vector $W_{i,j}$, we want to add all symbols that could have generated symbol R_k through a derivation using unit production rules. That is, if we note

$$\mathcal{C} = \{l \mid R_l \underset{UPR}{\Longrightarrow}^* R_k\}$$

We want to add

$$\hat{W}_{i,j}(l) = \max_{k \in \mathcal{C}} (p_{l,k} \cdot W_{i,j}(k))$$

where

$$p_{k,l} = \mathbb{P}(R_l \underset{UPR}{\Longrightarrow}^* R_k)$$

Notice that this is very similar to a matrix/vector multiplication, and a natural question would be whether it is possible to compute a matrix U such that we can handle production rule with only one matrix/vector operation. It turns out that it is possible :

$$\hat{W}_{i,j} = U \odot W_{i,j}$$

where we define for a matrix X and a vector V

$$[X \odot V]_i = \max_k X_{i,k} V_k$$

and for 2 matrices

$$[X \odot Y]_{i,j} = \max_k X_{i,k} Y_{k,j}$$

It turns out that if we calculate U^0 as

$$U^0_{i,j} = \mathbb{P}(R_i \Rightarrow R_j) + I_M$$

We can prove that this is a non decreasing sequence with respect to all the coefficients, and that every coefficient is bounded above by 1 which guarantees convergence (In fact the number of operations we need before reaching the fixed point is the length of the longuest non recursive derivation using only unit production rules) We can compute U as the limit of the recurence formula $U = \lim_{n\to\infty} U^n$ where

$$U^n = U^{n-1} \odot U^0$$

We also need to remember from which rule we came from to reconstruct the solution at the end. We can recover these indices by computing

$$T_i = \arg\max_k U_{i,k} W_k$$

In fact, in the CYK algorithm, it is better to work with log probabilities instead of probabilities directly to avoid underflow, so let us redefine the operation $\odot$ as

$$[X \odot Y]_{i,j} = \max_k X_{i,k} + Y_{k,j}$$

B.2 Sparse matrix view of the CYK algorithm

Let us define the function arg/max that returns a couple constitued of the max and the arg max coordinates of a matrix. It makes sense to do so because we can compute both simultaneously, and it greatly simplifies the expression of algorithm. The algorithm is given below (Algorithm 1). The algorithm starts by filling a tensor P as

$$P[i,j,k] = \mathbb{P}(R_i \to R_j R_k)$$

for the non unit rules, and computing the matrix U defined in the previous section. From there we can basically run the PCYK algorithm using only sparse tensor operations. The main operation involve adding a row vector to each rows of the matrix $P[p]$, then adding a column vector to each column of the matrix $P[p]$ and then taking the max over all the coefficient of the matrix (this gives a probability), as well as the arg max (this gives a couple (q,r) which corresponds to the highest probability that the sequence of word $w_i \dots w_j$ has been generated by the rule $R_p \to R_q\ R_r$ through a split at word k ($w_i, \dots w_k$ and $w_{k+1}, \dots w_j$). We stores the *argmax* in $B1$ and $B2$. $B1$ keeps track of the unit production rules $A \to B$ and $B2$ keeps track of the rules of the form $A \to B\ C$. Finaly S keeps tracks of the position of the split.

B.3 Implementation difficulties for this algorithm

Even if I think that this is a very good way to implement the CYK algorithm, I did not implemented it because sparse matrix libraries that I've found can't handle this efficiently. The reason behind this is that they assume that missing entries are 0, but in the proposed method, since all calculations are done in log probabilities, we really want them to be $-\infty$. In theory, this is not a problem, because we only want to have *product* replaced by *sum* and *sum* replaced by *max* in the matrix operations. By doing so, we see that the natural choice for missing values is then $-\infty$ ($-\infty$ is an absorbing element for the addition operation just as 0 is an absorbing element for the multiplication and taking the maximum between $-\infty$ and a value v yields v just like taking the sum betwwen 0 and v does).

However, I think that this use case is not that common, which is probably the reason why it's not implemented in libraries.

Algorithm 1: PCYK algorithm using sparse matrix operations

```
Function ComputeUnitTable($G$):
    for each unit production rule $R_p \to R_q$ do
        $U_0[p,q] \leftarrow \mathbb{P}(R_p \to R_q)$
    end
    $n \leftarrow 0$
    repeat
        $U_{n+1} \leftarrow U_n \odot U_0$
        $n \leftarrow n+1$
    until $U_{n+1} = U_n$
    return $U_n$

Function ComputeNonUnitTable($G$):
    for each non-unit production rule $R_p \to R_q\ R_r$ do
        $P[p,q,r] \leftarrow \mathbb{P}(R_p \to R_q\ R_r)$
    end
    return $P$

Function PCYK($w_0, \dots w_{N-1}$, $G$, $L$):
    $U \leftarrow$ ComputeUnitTable($G$)
    $P \leftarrow$ ComputeNonUnitTable($G$)
    for $j = 0$ to $N-1$ do
        $(A[j,j,p], B1[j,j,p]) \leftarrow \underset{q}{\text{arg/max}}\, U[p,q] + \mathbb{P}(R_q \to w_j) \quad \forall p$
        for $i = j-1$ down to $0$ do
            for $k = i$ to $j-1$ do
                $(A_t[p], B2_t[p]) \leftarrow \underset{q,r}{\text{arg/max}}\, P[p,q,r] + A[i,k,q] + A[k+1,j,r] \quad \forall p$
                $(A_t[p], B1_t[p]) \leftarrow \underset{q}{\text{arg/max}}\, U[p,q] + A[i,j,q] \quad \forall p$
                foreach $p$ such that $A_t[p] > A[i,j,p]$ do
                    $(S[i,j,p], A[i,j,p], B1[i,j,p], B2[i,j,p]) \leftarrow (k, S_t[p], A_t[p], B1_t[p], B2_t[p])$
                end
            end
        end
    end
return ReverseConstruction($w_0 \dots w_{N-1}$, $S$, $B1$, $B2$)
```