Intro : présentation de l'intitulé et de la composition du groupe

Notre projet consiste en l’implémentation de l’algorithme CYK probabilisé.

I. Qu'est ce que l'algorithme CYK ?

L’algorithme CYK tire son nom des trois individus qui l’on trouvé, Cocke, Younger et Kasami. Il s’agit d’un algorithme tabulaire d’analyse syntaxique ascendante pour les grammaires non-contextuelles. Celui-ci permet de déterminer si un mot est reconnu ou non par une grammaire, si oui, il fournit également un arbre syntaxique.

Cet algorithme demande à ce que la grammaire soit en forme normale de Chomsky. Le temps de calcul de l’algorithme est proportionnel à la taille de la grammaire et à la taille du mot à analyser.

Voici un exemple. Nous disposons d’une grammaire en forme normale de Chomsky :

S -> SN SV

SN -> D N

SV -> V SN

V -> mange

D -> la | une

N -> fille | pomme

{S, SN, SV, V, N, D} est l’ensemble des non-terminaux et {mange, la, une, fille, pomme} est l’ensemble des terminaux, c’est-à-dire nos lettres.

Nous disposons également d’un mot à parser : « la fille mange une pomme ».

Avec l’algorithme CYK, on obtiendra :

A REMPLIR AVEC LA MEME STRATEGIE QUE L’ALGO DE KORANTIN

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| La | Fille | Mange | Une | Pomme |

b) Présentation de la version probabiliste du CYK

Notre objectif était d’implémenter la version probabiliste de l’algorithme présentée précédemment.

- Quels intérêts ?

Cette version probabiliste est intéressante car elle permet de choisir le ou les meilleures analyses syntaxique (et arbres syntaxiques). En effet, lorsqu’on remplit la demi-matrice avec une grammaire un peu plus complexe que le petit exemple vu précédemment, on a souvent des cases que l’on peut remplir avec plusieurs règles différentes. Or chaque règles finira par engendrer des analyses et des arbres parfois très différents les uns des autres. Avec la version probabiliste du parseur, on peut choisir de ne garder que la (ou les) analyse les plus probables, en choisissant de remplir les cases de la demi-matrice avec la (ou les) règles qui ont une meilleure probabilité.

=> METTRE ICI un exemple avec une grammaire probabilisée et une demi matrice remplie pour comprendre le fonctionnement de la version probabiliste.

* Quels problemes éventuels avec cette version de CYK ?

Dans notre implémentation, nous devons choisir si nous gardons seulement une règle par case (la règle avec la plus forte probabilité), ou si nous en gardons plusieurs. En décidant de garder plusieurs règles, on autorise le parseur à produire potentiellement plusieurs analyses possibles pour une même phrase. En envisageant de couvrir plusieurs (voir toutes) les analyses possibles, nous sommes plus efficaces. Mais en contrepartie, la multiplication des règles dans les cases de la demi-matrice augmente de manière exponentielle la complexité de l’algorithme.

Nous verrons dans la suite de ce documents quels choix d’implémentation ont été retenus mais également ceux qui ont été abandonnés et pourquoi.

II. Notre implémentation

Maintenant que nous avons vu ce qu’est l’algorithme CYK et sa version probabiliste, nous allons nous pencher sur notre implémentation de cet algorithme. Comme nous l’avons vu précédemment, CYK nécessite pour son fonctionnement une grammaire en forme quadratique (ou forme normale de Chomsky). Ici, comme nous souhaitons implémenter la version probabiliste, il faut en plus que chaque règle de notre grammaire ait une probabilité, c’est-à-dire qu’il nous faut une PCFG (Probabilistic Context-Free Grammar).

a) Extraction de la grammaire en probabilisant chaque règle

- Présentation du corpus Sequoia fourni

Pour obtenir cette grammaire, nous disposons d’un corpus contenant plus de 3 000 phrases en français. Il s’agit du corpus Sequoia composé de diverses sources (Europarl, Le Monde, Wikipédia et EMA), c’est donc un corpus aux thèmes assez variés.

Un corpus à thèmes variés est meilleur qu’un corpus sur un seul thème ou domaine. Par exemple si l’on extrait une grammaire uniquement à partir d’un corpus médical, il y aura beaucoup de productions de la langue française qui nous échapperont totalement. En effet dans ce genre de corpus scientifique, il n’y a pas de dialogues, il n’y a pas ou peu de phrases à la première ou à la seconde personne et le vocabulaire est assez particulier et restreint. Des corpus très varié permettent lors de l’apprentissage de couvrir un maximum de phénomènes de la langue dans toute sa variété.

Le corpus Séquoia nous fournit un grand ensemble de phrases au format MRG. Ce format parenthésé est très pratique pour être parsé.

Voici un exemple de phrase que l’on peut trouver dans le corpus :

((SENT (NP-SUJ (DET La) (NC tâche)) (VN (ADV ne) (V sera)) (ADV pas) (AP-ATS (ADJ aisée)) (PONCT .)))

* Préparation des données avant extraction (ce qu’on ignore, ce qu’on garde et pourquoi)

Pour extraire une grammaire à partir de ce corpus, on doit d’abord définir quels seront nos terminaux et nos non-terminaux. Nous avons décidé que notre programme ne parsera pas des phrases de lexique mais uniquement des suites d’étiquettes syntaxiques.

Donc nos non-terminaux seront SENT, NP, VN, AP, ADV, DET, NC, V, ADJ… C’est-à-dire, toutes les étiquettes en majuscules que l’on trouve derrière une parenthèse.

Nos terminaux seront, non pas les mots de la phrase (en minuscules), mais nous transformerons ce lexique par la dernière étiquette juste avant lui. Par exemple pour « (V sera) » nous n’aurons pas de règle « V -> sera » mais plutôt une règle « V -> v ». Nos terminaux sont donc des catégories morphosyntaxiques notées en minuscules.

Sachant cela, nous pouvons dès à présent ignorer totalement les mots de lexique dans le corpus. Nous avons également décidé d’ignorer les fonctions grammaticales car elles ne sont pas primordiales pour un parseur syntaxique mais surtout parce qu’elles viendraient augmenter la taille de la grammaire et donc augmenter la complexité en temps de l’algorithme (rappelez-vous, nous avons vu dans la présentation du CYK que la complexité en temps de ce dernier est proportionnelle à la taille de sa grammaire).

- Expliquer comment fonctionne notre programme d’extraction

Notre programme d’extraction de grammaire prend en entré le fichier MRG.

Il nous permet de choisir quelle grammaire nous voulons en sortie. On peut choisir une CFG ou alors une PCFG, c’est cette dernière qui nous intéresse pour faire fonctionner notre CYK probabiliste !

- la partie qui extrait des structures « propres » en parcourant le corpus

La première étape consiste à parcourir le corpus à l’aide du parseur PLY. Celui si nous permet de…

EXPLIQUER LES ETAPES D’EXTRACTION EN DETAIL

Illustrer avec des exemples si possible

- Présentation des fonctions qui vont ensuite parcourir cette structure « propre » pour en extraite à chaque fois les règles et attribuer une probas

## COMMENT ON S’Y PREND POUR EXTRAIRE LES REGLES ? DANS QUELLE GENRE DE STRUCTURE DE DONNEES LES STOCKE T-ON ? ##

Les règles de productions ainsi extraites sont stockées sous forme d’objets. Nous avons créé une classe abstraite Grammaire et des classes filles CFG et PCFG. La classe PCFG a plusieurs attributs de classe, tous sont des listes, nous avons une liste de non-terminaux, une liste de terminaux, une liste de productions (ce sont les règles de réécriture, qui sont des tuples (lhs, rhs), « lhs » est la partie gauche de la règle et « rhs » est la partie droite.). L’axiome est par défaut SENT, car nous avons fait l’hypothèse qu’il n’y a pas de cycles dans la grammaire #IL FAUT PROUVER CAAAA#. Nous disposons de méthodes de classe pour attribuer une valeur (une probabilité) à chaque production. Dans un souci de précision, nous utilisons le module « fractions » pour Python pour les probabilités, et non pas des float.

* Probabiliser la grammaire

Comment s’y prendre pour probabiliser nos règles de réécriture ? C’est une question sur laquelle nous avons passé du temps. Nous avions deux stratégies et nous avons dû faire un choix. Tout d’abord, nous avions pensé à probabilisé chaque règle au fur et à mesure qu’elles soient extraites. Cela nous semblait plus simple au départ. Notre idée était la suivante, chaque fois qu’on rencontre une règle :

* Si la partie gauche n’a jamais été trouvé avant, la règle vaut 1.
* Si la partie gauche a déjà été trouvé avant mais que la partie droite est nouvelle, on divise 1 par le nombre de règles vu auparavant (+1 car on vient d’en trouver une autre) et on obtient la probabilité de chaque règle partageant cette même partie gauche. Il faut alors mettre à jour toutes ces règles-là.
* Si la partie gauche a déjà été trouvée et que la partie droite aussi (c’est-à-dire qu’on a déjà rencontré cette même règle) alors on divise 1 par le nombre de règles (comme précédemment) sauf qu’ensuite on additionne les probabilités des deux règles identiques et on en supprime une. Ainsi la règle qui a été vu deux fois (ou plus) a une probabilité plus forte que les autres qui n’ont été vues qu’une seule fois.

Finalement, nous avons décidé de probabiliser chaque règles après qu’elles soient toutes extraites. En effet, cela nous évite de faire sans cesse des mises à jour des règles vues auparavant. Donc la méthode est la suivante. Pendant l’extraction des règles, on dispose de compteurs. Certains nous permettent de compter +1 chaque fois qu’on rencontre une règle parfaitement identique à une ou plusieurs règles déjà extraites. D’autres nous permettent de compter lorsque nous rencontrons une partie gauche de règle qui a déjà été extraite. Après l’extraction, nous reprenons nos compteurs afin d’attribuer les probabilités à nos règles de productions. Pour mieux comprendre ce processus, voici une exemple avec une petite grammaire :

VP -> v

VP -> v NP

VP -> v ADJ

VP -> v

On dispose de 4 règles qui ont la même partie gauche « VP » donc notre compteur de VP est à 4. Nous disposons également de 2 règles VP identiques, donc notre compteur de règles « VP -> v » est à 2.

Nous divisons le nombre de règles identiques par le nombre de parties gauches identiques. Nous obtenons 2/4 = 1/2. C’est la probabilité attribuée aux deux règles identiques.

VP -> v = 1/2

VP -> v NP

VP -> v ADJ

VP -> v = 1/2

Pour attribuer une probabilité aux règles uniques, nous divisons 1 par le nombre de parties gauches identiques, nous obtenons 1/4.

VP -> v = 1/2

VP -> v NP = 1/4

VP -> v ADJ = 1/4

VP -> v = 1/2

b) Mettre la grammaire obtenue en forme normale de Chomsky

- Expliquer l'algorithme de mise en forme quadratique

- Expliquer comment nous avons implémenté ça en prenant en compte le recalcule des probas

c) On a enfin une PCFG, on peut faire fonctionner l'algorithme CYK probabiliste. Expliquer son fonctionnement. Expliquer nos choix d'implémentation (par exemple la forme de notre demi matrice, combien de règles par cases et pourquoi, qu’obtient-on en sortie ?, si on a essayé plusieurs versions il faut toutes les mettre et expliquer pourquoi on les a finalement abandonnées.

III. L'évaluation du programme

- Généralités sur l’évaluation d’un parseur, formules (précision, rappel, f-score)

Pour comparer nos arbres de sortie et les arbres gold, il faut remettre nos arbres en n-aire. Car les arbres gold ne sont pas binaires.

- Notre évaluation (les entrées, les sorties, que fait l’algo en détail)

- Nos résultats commentés

IV. Manuel d'utilisation

- Comment lancer l'extraction de la grammaire et comment parser une phrase

- listing des commandes dispo (un « help » ?)

Programme d’extraction de grammaire :

Options disponibles :

- g : type de grammaire

-- markov\_type : true/false

--- markov\_ordre : int

=> Ne pas oublier de bien illustrer avec des screenshoot genre tuto.

V. Annexes

- Listing des algorithmes des programmes en pseudo code (dans leur totalité) pour l'extraction, pour le CYK et pour l’évaluation.

ALGO INIT\_CYK (ARGUMENTS ?) : SORTIE ?

| span = 1

| for (I, letter) in enumerate (len(mot)) :

| | for nt in nonTerminals :

| | | for lexical in productions(nt) :

| | | | chart[spam][i][nt] = (lexical.proba, (span, i))

| | [ Fin for

| | Fin for

| Fin for

Fin ALGO

ALGO CYK (ARGUMENTS ?) : SORTIE ?

| Chart = defaultdict(lambda : defaultdict(int))

| for max in range (2, len(mot) +1) :

| | for min in range (max – 2, 0, -1) :

| | | for nt in nonTerminals :

| | | | best = 0

| | | | for binaire in prod-binaire-nonTerminals :

| | | | | for mid in range (min + 1, max -1) :

| | | | | | t1 = chart[min][mid][nt]

| | | | | | t2 = chart[min][max][nt]

| | | | | | candidat = t1 \* t2 \* binaire.proba

| | | | | | if candidat > best :

| | | | | | | best = (candidat, (t1.ind,t2.ind))

| | | | | | Fin if

| | | | | Fin for

| | | | | Chart[min][max][nt] = best

| | | | Fin for

| | | Fin for

| | Fin for

| Fin for

Fin ALGO