IFT6285 – Devoir 6

Étiquetage Morphosyntaxique avec NLTK

Maxime MONRAT

Juan Felipe DURAN

Université de Montréal

Automne 2021

Objectifs

Nous explorons dans ce devoir les capacités d'analyse morphosyntaxique (*POS*) de la plateforme NLTK. Nous utilisons dans ce but un extrait du *Penn Tree Bank*.

1. Modèle CRF

Le premier modèle d'analyse que nous avons exploré est un modèle à champs aléatoire conditionnel (*CRF, conditionnal random field*). La classe **CRFTagger** de NLTK rend l'entraînement extrêmement simple. À l'aide de celle-ci nous avons entraîné plusieurs modèles utilisant deux manières différentes d'extraire des caractéristiques :

- En utilisant la fonction par défaut de la classe CRFTagger, qui ne prend en considération que le mot en cours de prédiction,
- En utilisant une fonction personnalisée *CustomFeatureFunc* qui cherche les mêmes caractéristiques, mais en prenant en compte le contexte du mot en question (les 2 mots précédents, et les 2 mots suivants)
- En utilisant la fonction *word2features* telle que définie par Aiswarya Srinivas dans son tutoriel d'utilisation de CRFsuite avec sci-kit. Cette fonction extrait plusieurs caractéristiques supplémentaires telles que la racine, les préfixes, la taille du mot ou encore les indicateurs de début ou de fin de phrase.

On peut remarquer que la taille du contexte influe légèrement sur l'exactitude des résultats. Dans les deux cas, on observe une exactitude maximale de 95.56%, mais pas pour la même taille de contexte : la fonction *word2features* permet d'obtenir ce résultat sans prendre en compte le contexte (on remarque d'ailleurs que le contexte a peu d'influence lorsqu'on utilise cette fonction), mais pour obtenir la même valeur sur nos données avec notre fonction personnalisée, qui extrait les mêmes caractéristiques que la fonction par défaut *_get_features* définie par NLTK, il est nécessaire de considérer le mot précédent et le mot suivant.

Taille du Contexte	**0**	**1**	**2**	**3**	**4**
CustomFeatureFunc	94.74%	95.45%	**95.56%**	94.57%	94.31%
word2features	**95.56%**	95.53%	95.47%	95.40%	95.31%

Table 1: Exactitude des modèles en fonction de la taille du contexte et des fonctions d'extraction de caractéristiques.

2. Modèles transformationnels

Dans un second temps nous avons entraîné plusieurs modèles transformationnels à l'aide de la classe *BrillTagger* de NLTK. Ces modèles prennent plusieurs méta-paramètres pour l'entraînement, nous avons ici exploré l'impact du nombre maximal de règles et de types de *templates.*

On remarque que le nombre maximal de règles influe beaucoup sur la performance, jusqu'à plafonner autour de 50 règles. Nous avons décidé d'étudier 4 templates de différentes tailles, en pensant que la complexité du template augmenterait la performance : on observe que ça n'est pas vraiment le cas. En effet notre 'petit Template' fait à la main ne comporte que trois règles, et est plus performant que le template demo de NLTK qui en comporte 18. Il en va de même pour le template d'Eric Brill issu de l'article original.

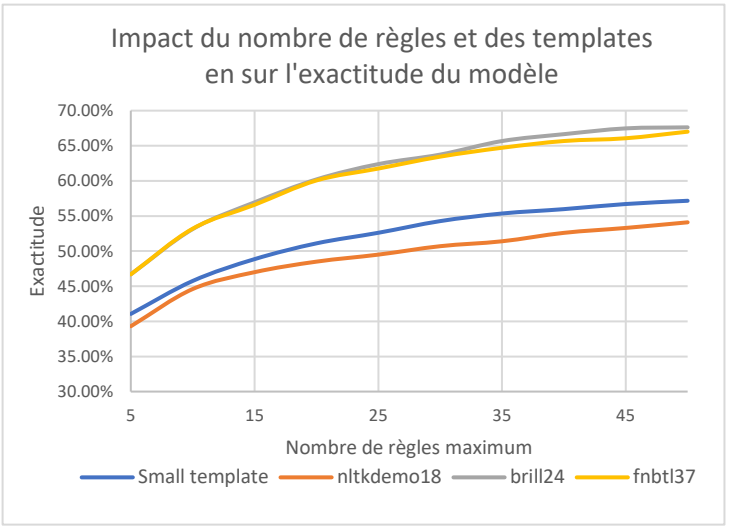

Figure 1: Impact des templates et du nombre de régles sur la performance d'un modèle TBL

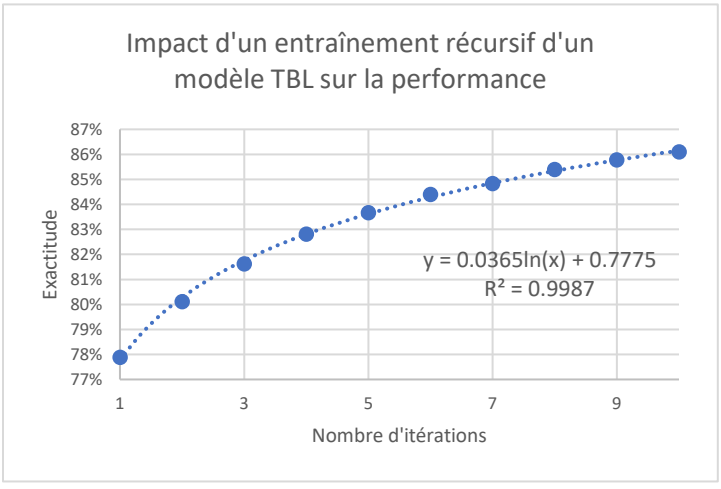

Figure 2: Impact d'un entraînement récursif sur la performance d'un modèle TBL

En entraînant notre modèle de manière récursive, c'est-à-dire en prenant le modèle entrainé comme étiqueteur de base pour un nouveau modèle, on observe que la performance générale augmente de manière logarithmique, et se stabilise au bout d'une douzaine d'itérations.

Les règles obtenues en basant notre modèle sur un étiqueteur Regex ou CRF sont assez similaires, la différence principale étant qu'elles n'apparaissent pas au même moment

Regexp	CRF
AT->DT if Pos:NN@[1,2,3]	NNP->, if Word:,@[0]
NN->, if Word:,@[0]	NNP->NN if Pos:NNP@[-2] & Pos:NNP@[-1]
NN->. if Word:.@[-1,0]	NN->. if Word:.@[-1,0]
NN->IN if Pos:DT@[1]	NN->DT if Word:the@[0]
NN->TO if Word:to@[0]	NN->IN if Word:of@[0]
NN->IN if Word:of@[0]	NN->TO if Word:to@[0]
NN->CC if Word:and@[0]	NN->DT if Word:a@[0]
NN->-NONE- if Word:*-1@[-1,0]	NN->IN if Pos:DT@[1]
NN->IN if Word:in@[0]	NN->-NONE- if Word:0@[-1,0]
CD->-NONE- if Word:0@[0,1]	NN->CC if Word:and@[0]
NN->-NONE- if Word:*@[0]	NN->IN if Word:in@[0]
NN->VB if Pos:TO@[-1]	NN->VB if Word:to@[-1]
NN->-NONE- if Word:*T*-1@[-1,0]	NN->-NONE- if Word:*-1@[-1,0]
NN->NNP if Word:Mr.@[-1,0]	NN->CD if Word:*U*@[1,2]
NNS->POS if Word:'s@[0,1]	NN->-NONE- if Word:*@[0]
NNS->VBZ if Word:is@[0]	NNP->DT if Word:the@[0,1]
NN->`` if Word:``@[0]	NNP->DT if Word:The@[0,1]
NN->IN if Word:for@[0]	NN->POS if Word:'s@[0]
NN->-NONE- if Word:*U*@[0]	NN->-NONE- if Word:*U*@[0,1]
NN->$ if Word:$@[-1,0]	NN->IN if Word:for@[0]

Table 2: Exemples de règles obtenues en basant notre modèle TBL sur un étiqueteur Regex ou CRF

3. Encapsulation de l'étiqueteur de SpaCy

Après avoir encapsulé le tagger de Spacy dans une classe héritée de TaggerI, nous avons utilisé la fonction « evaluate » pour vérifier les performances. Nous obtenons 89.1% pour le modèle *sm* et 89.2% pour le modèle *lg*. Cependant, dans le site https://spacy.io/models/en, la performance attendue est de 0.97 pour les deux. Il est possible que la fonction « evaluate » de NLTK utilise des métriques différentes pour évaluer un modèle que Spacy, et c'est pour cela qu'un modèle natif de Spacy pourrait avoir des moins bonnes performances dans NLTK.

Ensuite, nous avons utilisé le tagger encapsulé de Spacy pour entrainer un modèle transformationel de BrillTagger, et pour l'évaluer. Pour le modele « sm », nous obtenons 95.3%. Pour le modele « lg », nous avons eu 95.1%.