Rapport de l'application (WIA)

Extraction des relations sociales entre les personnages d'un texte

Sofiane Kihal et Antoine De Scorraille

I - Introduction

L'objectif de l'application est d'**extraire les relations sociales entre les personnages d'un texte**. Trois problèmes principaux se présentent :

- La définition et l'implémentations des règles qui vont permettre d'extraire les relations entre les personnages du texte.

- Les pré-traitements de texte pour que l'analyse se fasse correctement.

- La définition des outils NLP à disposition pour mener à bien l'objectif.

Nous déclinerons les éléments de résolution utilisés dans notre application pour pallier ces problématiques et celles qui en découlent en parallèle de l'explication du code et des aspects purement techniques.

II - Architecture de l'application

L'application a, dans un premier temps, été développé sur un Note Book en langage Python. Nous avons ensuite transposé le code sur des fichiers python plus classiques. L'application se décompose en trois parties :

- Un fichier *main.py* : il contient les lignes qui permettent à l'application d'extraire les relations entre les personnages et de les afficher. Il contient également les lignes qui permettent d'évaluer le résultat et d'afficher cette évaluation (précision et rappel).

- Un fichier *functions.py* : ce fichier contient toutes les fonctions qui sont appelées dans main.py. Ces fonctions servent au traitement du texte, à l'initialisation et au remplissage des structures utiles à l'élaboration de l'application, à l'extraction des relations sociales entre les entités nommées du texte et à leur formalisation dans une structure adéquat, et enfin à l'évaluation des résultats fournis par l'application.

- Des fichiers de texte : chaque texte a 3 fichiers, un fichier contenant le récit de l'histoire 'Nemo.txt' par exemple, un fichier contenant les personnages présents dans le texte 'Nemo_CHARACTERS.txt' et un fichier contenant les relations, dans un seul sens, des personnages dans le texte 'Nemo_RELATIONSHIPS.txt'. Ces deux derniers servent à l'évaluation du résultat, ils n'interviennent pas dans l'exécution de l'application.

Remarque : Le fichier '*_RELATIONSHIPS.txt' doit être rempli par l'utilisateur en prenant en compte l'ordre d'apparition de la description des relations dans le texte. Il doit également prendre en compte le sens de la relation et le terme utilisé pour la décrire. Par exemple, s'il est écrit dans le texte 'Marlin est le père de Némo', il faudra écrire 'Marlin,father,Nemo' et non pas 'Marlin,parent,Nemo' , 'Nemo,son,Marlin' ou encore 'Nemo,child,Marlin' même si fondamentalement cela veut dire la même chose. Si cela n'est pas respecté, les scores affichés pour l'évaluation seront faussés.

III - Détail du code et des méthodes utilisées

Fichier main.py

Variables

Dans ce fichier nous déclarons différentes variables qui serviront dans l'utilisation des fonctions de l'application avant d'appeler ces dites fonctions dans l'ordre et afficher le résultat et l'évaluation de ce résultat. Parmi ces variables :

- f_text : contient le fichier texte dans lequel se trouve le récit de l'histoire.

- file_characters, file_relationships : ces variables accueillent les fichiers de texte relatifs aux personnages et aux relations présentes dans le texte d'origine.

- CHARACTERS : est une liste contenant les personnages extrait dans le texte, initialement vide.

- DEPENDENCIES : c'est un tableau qui va contenir les dépendances grammaticales présentes dans les phrases du texte. L'indice de chaque colonne du tableau correspond à l'indice de chaque phrase du texte. Le texte sera découpé par des fonctions que nous détaillerons plus bas, de façon à ce que les indices correspondent.

- TOKENS : c'est également un tableau dans lequel chaque colonne du tableau correspond à l'indice de chaque phrase du texte. Dans chaque sous-liste on retrouvera la phrase associée et découpée en 'token' par une fonction de la librairie nltk. Ce tableau va nous permettre de 'traduire' des indices de mot en mot. En effet, lors de nos traitements, les mots seront manipulés

par leur indice qui correspond à leur position dans la phrase. Afin d'avoir une sortie compréhensible, nous utilisons ce tableau dans lequel chaque indice de colonne correspond au numéro de la phrase et chaque indice de ligne correspond au numéro du mot que l'on souhaite connaitre.

- RELATIONSHIPS : c'est le dictionnaire qui contiendra la sortie. Toutes les relations entre les personnages seront listées dans ce dictionnaire. Ainsi, par exemple, pour Le Monde de Nemo, on aura une sortie de type : RELATIONSHIPS = { Nemo : (father,Marlin) , Marlin : (child,Nemo) }.

- INCOMPLETE_RELATIONSHIP : à l'origine, ce dictionnaire devait fonctionner sur le même principe que RELATIONSHIP, à l'exception près qu'il contient les relations dans lesquelles il manque une des deux entités. Nous avions jugé qu'il était pertinent d'avoir accès à des relations même si elles étaient incomplètes. Le code s'étant complexifié avec notre volonté de l'améliorer, nous avons perdu l'extraction de ces relations au profit de l'extraction de relations plus précises et complètes.

- LINKS : liste contenant tous les 'marqueurs de relation' que nous avons défini. Ils vont permettre d'identifier si, selon ce que notre méthode renvoie comme étant une relation (en se basant sur des constructions grammaticales), le résultat renvoyé est bien une relation valide entre deux personnages. Pour ce faire, elle identifie si le lien qui unie les deux supposées entités nommés dans la construction grammaticale est présent dans la liste LINKS.

- LINK_CORRESPONDANCE : ce dictionnaire contient tous les symétriques généralisés des marqueurs de relation. Par exemple, pour 'son' on va avoir le symétrique 'parent' (et non pas 'father' ou 'mother' car nous n'avons pas défini de méthode pour avoir une telle précision). Ce dictionnaire va nous permettre de créer les relations symétriques de celles que nous avons déjà identifié et les insérer dans notre résultat final.

Algorithme

L'algorithme est simple.

- Il y'a un appel à l'utilisateur qui doit entrer le nom du fichier d'origine.

- On extrait ensuite les personnages du texte (fonction *extract_NE*), on les stocke dans CHARACTERS et on initialise notre dictionnaire RELATIONSHIPS (fonction *init_relationships*).

- On procède à du traitement sur le texte original :

- on découpe le texte par mot et on remplace les occurrences faisant référence aux personnages telles que 'him/his/her/etc..' par le nom des personnages correspondant et on reconstitue le texte (fonction *replace_by_NE*).
- On découpe en token de phrase ce nouveau texte (fonction *sent_tokenize* de la librairies NLTK).

- Enfin, on tokénise par mot chaque phrase et on la stocke dans TOKENS (fonction *word_tokenize* de la librairie NLTK).

- On fait l'analyse des dépendances grammaticales et on les stocke dans DEPENDENCIES (fonction *nlp.dependency_parse* de la librairie StanfordCoreNLP et fonction *extract_dependencies*)

- On remplit RELATIONSHIPS par les relations qu'on a réussi à extraire à l'aide de nos méthodes (fonction *fill_relationships*)

- On crée le symétrique des relations qu'on a extraite (fonction *make_correspondance*).

- On fusionne RELATIONSHIPS et son symétrique pour obtenir la totalité des relations (fonction *merge_dictionnary*).

- On affiche le résultat et la mesure de l'évaluation de ce résultat (fonction *accuracy*).

Fichier functions.py

Le code étant abondamment commenté, nous détaillerons ici uniquement les fonctions qui méritent d'être expliquées et principalement notre méthode d'extraction de relation.

Fonction *extract_NE(text,CHARACTERS)* :

Extrait les entités nommées d'un texte (text) et les stocke dans une liste (CHARACTERS).
Pour ce faire, nous utilisons la fonction *nlp.ner()* de la librairie StanfordCoreNLP.

Remarque : Notre méthode d'extraction des relations permet néanmoins d'extraire des entités nommées qui n'auraient pas été détectées dans un premier temps par cette fonction.

Fonction *extract_dependencies(text_tokens, DEPENDENCIES)* :

Extrait les dépendances 'nmod:poss', 'appos', 'compound', et 'dep' au sein d'un texte découpé par phrases (text_tokens) et les insère dans DEPENDENCIES. Le choix de ces dépendances sera justifié dans la description de notre méthode (fonction suivante).

Fonction *make_relation(dep, dependencies, nb_line) :*

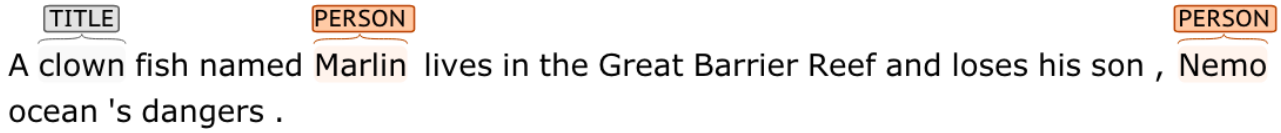
Renvoie une liste encoded_relationships contenant des quadruplés qui correspondent à des relations valides sous la forme (personne1,relation,personne2, ligne du texte où la relation a été identifiée).

Remarque : la ligne du texte est indiquée car elle va être nécessaire lors de la 'traduction' de ce quadruplé en une forme compréhensible. En effet, on va utiliser notre tableau TOKENS dans lequel on va devoir aller chercher dans la bonne colonne (donc dans la bonne phrase), les indices 'personne1' et 'personne2' pour en extraire les mots correspondant.

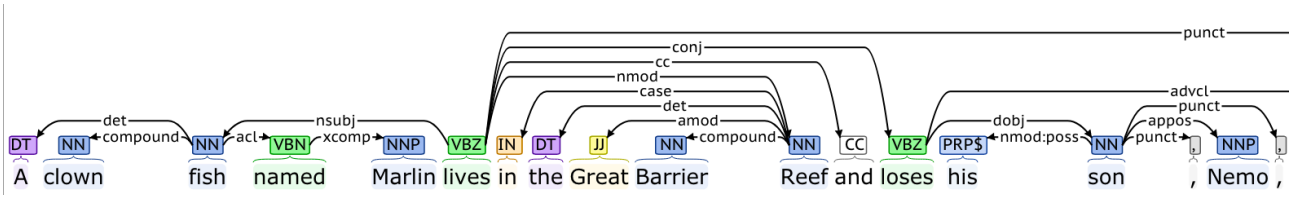
Méthode de detection de relation (présente dans make_relation())

Pour illustrer la méthode, nous allons prendre le morceau de phrase suivant : 'A clown fish named Marlin lives in the Great Barrier Reef and loses his son, Nemo.' qui illustre un cas simple.

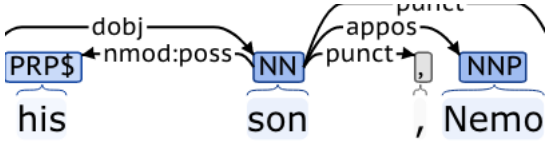
Dans un premier temps, nous avons identifié les entités nommées Marlin et Nemo en prenant soin de ne selectionner que ceux qui correspondent à des personnes. Pour ce faire, nous utilisons la fonction extract_NE et plus principalement *nlp.ner()* de la librairie StanfordCoreNLP.



Nous utilisons ensuite l'analyseur de dépendances grammaticales qui nous renvoie ceci :



A force d'observations sur différents corpus de textes, nous avons identifié une construction grammaticale qui se veut courante et semble être le modèle de base pour décrire une relation entre deux personnes. Il s'agit de la construction suivante :



Très souvent, il apparait que la combinaison d'un modificateur nominal possessif et d'une apposition permet de décrire une relation.

L'analyseur de dépendance renvoie des tuples pour formaliser ces dépendances, ils ont la forme suivante : (dépendance, indice du mot1, indice du mot2).

On vérifie alors, dans notre méthode, dans un premier temps, que nous avons bien sélectionné un 'nmod:poss', que l'élément qui le suit est de type 'appos', et que l'indice du mot1 de la dépendance 'nmod:poss' correspond à l'indice du mot1 de la dépendance 'appos'. Enfin, on vérifie que ce mot est bien un marqueur de relation, ici c'est le cas ('son'), et on ajoute la relation à notre dictionnaire RELATIONSHIPS.

Ici, si on se contente de suivre cette méthode, on aura dans le dictionnaire RELATIONSHIPS la relation 'His : (son,Nemo)'. Pour pallier ce défaut, nous avons écrit la fonction *replace_by_NE* qui permet de remplacer les occurrences telles que 'his' par le nom auxquelles elles font référence. Nous détaillerons cette fonctions plus loin dans le rapport.

Remarque : Ce qui vient d'être décrit est une règle. De la même manière, nous avons établi d'autres règles afin d'extraire le maximum de relation possible dans tous les styles d'écriture. En effet, il existe autant de règles qu'il existe de construction grammaticales pour décrire une relation. Nous détaillerons les avantages et inconvénients de ces règles dans la partie appropriée du rapport.

Fonction
fill_relationships(DEPENDENCIES,TOKENS,LINKS,RELATIONSHIPS,INCOMPLETE_RELATIONSHIP) :

Remplit le dictionnaire RELATIONSHIP avec les relations qui ont pu être extraites à l'aide de la fonction 'make_relation()'

Remarque : C'est dans cette fonction que l'on va également faire la 'traduction' des indices en mot à l'aide de la liste TOKENS.

Fonction
replace_by_NE(text_split, CHARACTERS) :

Remplace toutes les occurrences de la liste 'to_replace' par l'entité nommée à laquelle elles font référence dans le texte.

La liste d'occurrences est : ['he','she','his','him','her','He','She','His','Her','Him'].

Remarque : Pour les mot 'his' et 'her' on rajoute « 's » à la suite du nom du personnage pour conserver les dépendances grammaticales. On soulignera que cette fonction est simpliste (elle

remplace l'occurrence par la dernière entité nommée mentionnée) et commet souvent des erreurs sur les longs textes et styles d'écriture plus 'matures' en associant à ces occurrences les mauvaises entités nommées.

IV - Evaluation et analyse

Evaluation

Pour l'évaluation nous avons décidé d'utiliser la précision et le rappel comme indicateurs.

Précision : combien de candidats sélectionnés sont pertinents ?
Rappel : combien d'éléments pertinents sont sélectionnés ?

Nous les avons défini comme ceci:

P = (nombre d'entités nommées pertinentes + nombre de relations pertinentes)/nombre de candidats sélectionnés au total

R= (nombre d'entités nommées pertinentes + nombre de relations pertinentes)/nombre total de candidats pertinents

Le nombre d'entités nommées pertinentes est donné par le fichier '*_CHARACTERS.txt' comme précisé en début de rapport.

Le nombre de relations pertinentes est donné par le fichier '*_RELATIONSHIPS.txt' comme précisé en début de rapport.

Remarque : Pour l'étiquetage des relations pertinentes dans le fichier '*_RELATIONSHIPS.txt' nous avons fait le choix d'écrire uniquement les relations identifiables à l'aide de constructions grammaticales en faisant abstraction de la dimension sémantique puisque notre programme ne s'appuie que sur des règles grammaticales. De plus, même si nous pouvons dans un texte déduire de manière 'intelligente' que deux personnages qui s'entraident sont alors amis, si leur relation n'est pas clairement explicitée grammaticalement dans le texte (comme dans l'exemple de Némo), nous considérons qu'elle ne doit pas être étiquetée.

Ainsi, nous avons pu obtenir les résultats suivant :

Pour le synopsis de Némo (texte très court) :
Le texte contient 2 personnages et 2 relations.
Précision et rappel : 100%

Pour le synopsis du Roi Lion (texte court) :
Le texte contient 7 personnages et 8 relations.
Précision : 58%
Rappel : 54%

Pour le conte de Hamlet pour enfant (texte court-moyen) :
Le texte contient 7 personnages et 14 relations.
Précision : 93%
Rappel : 67%

Pour le résumé détaillé du Roi Lion (texte moyen-long) :
Le texte contient 13 personnages et 18 relations.
Précision : 66%
Rappel : 61%

Analyse

Pour le synopsis de Némo, l'analyse est simple : le texte est très court, il contient peu de personnages et et la construction des phrases est très basique.

Pour le synopsis du Roi Lion : le texte ici est tout aussi court et contient beaucoup plus de personnages et de relations. Les constructions de phrases sont donc plus élaborées pour pouvoir décrire une histoire avec plus d'éléments et autant de ligne, et donc les relations sont plus difficiles à identifier par notre méthode.

Pour le conte de Hamlet pour enfant : c'est le texte sur lequel nous avons travaillé principalement pour déduire nos règles. Le résultat est donc satisfaisant sur la précision. C'est un conte pour enfant, les constructions grammaticales sont donc très simplistes ce qui explique aussi des résultats meilleurs que ceux du synopsis du Roi Lion.

Pour le résumé détaillé du Roi Lion : on peut expliquer des résultats plus performant que son synopsis pour plusieurs raisons. D'abord, le texte est beaucoup plus long. Ainsi, si une relation n'a pas été identifiée à un endroit du texte, lorsqu'elle est mentionnée plus tard dans le texte elle peut être détectée par le programme. Nous avons ainsi plus de 'chances' de détecter une relation. De plus, le nom des personnages est mentionné beaucoup plus de fois, il paraît donc logique que le nom de ces personnages soient plus facilement detectable également.

V - Limites et améliorations

Nous avons rencontré nos premieres limites lors de l'élaboration des règles d'extraction de relation. En effet, il existe autant de règles à implémenter qu'il existe de construction grammaticales pour décrire une relation. De plus, ces constructions grammaticales doivent être complexifiées car il arrive parfois que les relations soient inversées ou incorrectes.

La fonction 'replace_by_NE' connait également des limites dans son fonctionnement. Elle remplace les occurrences 'he/she/his/her/him/etc' par les personnages auxquelles elles font références. Cependant, pour ce faire, elle remplace l'occurrence par le dernier personnage mentionné dans le texte, ce qui peut souvent s'avérer être une erreur et donc occasionner des erreurs au niveau de la description de relations car on aura remplacé l'occurrence par le mauvais personnage. Nous pourrions améliorer cette fonction en utilisant les co-références par exemple.

Nous nous sommes principalement concentré sur des constructions grammaticales pour extraire des relations car cela nous paraissait plus intuitif et plus simple. Il serait intéressant de s'interroger sur ce que les techniques relatives à la sémantique pourrait apporter à cette application.

On pourrait également imaginer une fonction qui déduirait des relations entre les personnages, comme le fait la fonction 'make_correspondance' mais de manière transversale et non symétrique comme celle-ci. C'est à dire une fonction qui déduit du fait que Mufasa soit le frère de Scar et le père de Simba, le fait que Simba soit le neveu de Scar.

Enfin, pour les fonctions d'évaluation, il serait intéressant de permettre à l'utilisateur d'écrire la relation tel qu'il le souhaite et non pas strictement comme elle est écrite dans le texte sans que le résultat soit faussé comme nous l'avions précisé précédemment dans le rapport.