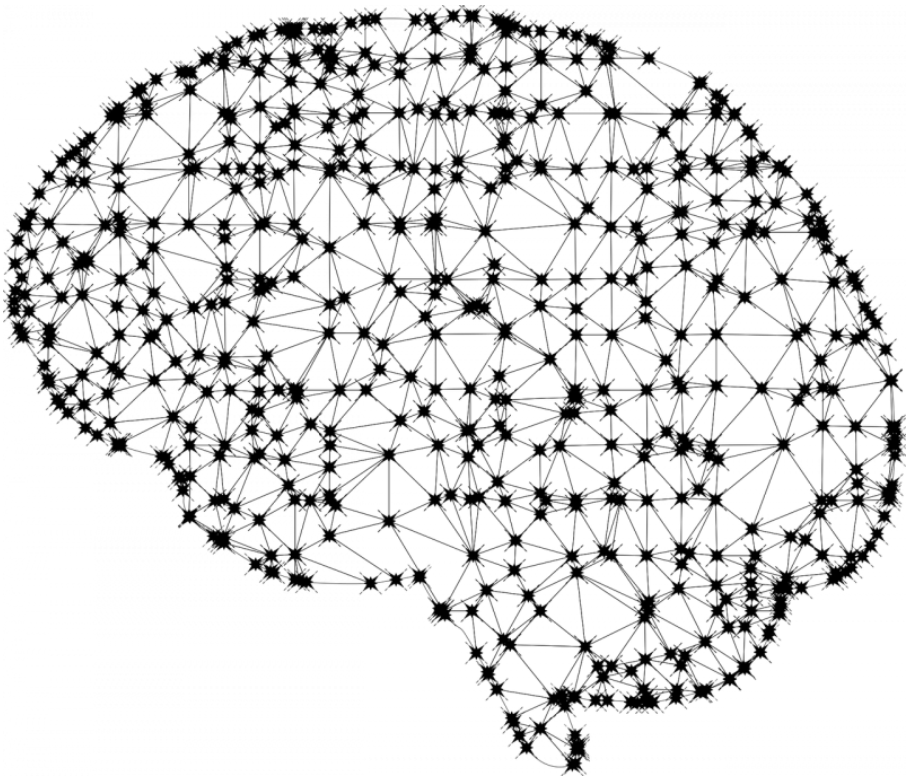
LABORATOIRE
D'INFORMATIQUE
& SYSTÈMES

Extraction de relations en Deep Learning par modèle hybride

Valentin Macé

Polytech – 2018



Extraction de relations en Deep Learning par modèle hybride

Rapport de stage

Valentin Macé

Tuteurs de stage

M. Bernard Espinasse & M. Adrian Chifu

Remerciements

Je tiens à remercier Bernard Espinasse et Adrian Chifu, qui m'ont permis d'effectuer mon stage au sein du LIS sur une problématique qui m'intéressait particulièrement en m'intégrant et me faisant participer activement à la vie du laboratoire.

Je remercie également Sébastien Fournier, Gaël Guibon, René Azcurra Irigoitia et tous les doctorants du LIS pour leur accueil chaleureux, avec qui j'ai souvent discuté et qui m'ont permis de découvrir le métier de chercheur.

Résumé

Ce stage s'inscrit dans le cadre des travaux menés par l'équipe DIMAG du LIS sur l'extraction d'information automatique dans un document textuel par apprentissage profond (deep learning).

Différents travaux portent sur l'usage du deep learning dans l'extraction et la classification de relations, notamment les travaux de (Nguyen and Grishman, 2015) utilisant des réseaux neuronaux convolutifs après une étape de word embedding comme Word2Vec (Mikolov et al., 2013), et sans prise en compte de caractéristiques linguistiques particulières.

D'autres travaux essaient de prendre en compte de telles caractéristiques linguistiques, soit en amont au niveau d'un word embedding tenant compte des dépendances syntaxiques (Levy & Goldberg, 2014), soit en utilisant un modèle hybride, comme dans les travaux de (Gormley et al., 2015) avec le modèle FCM pour Feature-Rich Compositional Embedding Model.

L'objectif de mon stage fut, dans un premier temps, de m'initier au traitement automatique du langage (TAL), de comprendre en détail le FCM sous son aspect théorique afin de pouvoir le présenter aux autres chercheurs, puis de maitriser son implémentation via les outils développés par ses concepteurs.

J'ai eu l'occasion d'expérimenter ce modèle en l'utilisant sur des corpus avec lesquels il n'avait pas encore été testé. C'est cette dernière tâche qui a constitué la plus grande partie de mon travail, pour laquelle j'ai développé des scripts en Python pour le prétraitement des données, l'automatisation du lancement du FCM sur des corpus avec différents paramètres, et la génération des mesures de performances.

Mon travail[1], couplé à celui d'autres chercheurs, permettra d'utiliser le FCM sur un ensemble de corpus plus large, de prétraiter de nouveaux corpus pour les utiliser avec le FCM et ainsi de pouvoir le comparer avec d'autres modèles plus facilement.

[1] https://github.com/valentinmace/fcm

Table des matières

1 Introduction

Tout au long de ce rapport, j'utiliserai de nombreux termes anglais. Il serait difficile de traduire ces termes techniques, parfois très spécifiques voire même propres à la publication que j'ai étudiée. Enfin, il serait encore plus compliqué pour le lecteur d'effectuer une recherche sur des termes qui ne donnent aucun résultat en français. J'essaierai toutefois de rester francophone autant que possible.

1.1 Contexte

Pour ce stage de quatrième année, j'avais à cœur de trouver une mission dans le domaine qui m'intéresse le plus, celui du machine learning[2] et plus précisément de l'apprentissage profond par réseaux de neurones artificiels. Je trouvais particulièrement pertinent de travailler dans cette voie pour compléter la formation reçue à Polytech Marseille, où nous avons eu un module sur l'apprentissage automatique dans lequel nous n'avons pas (ou très peu) traité des réseaux de neurones.

Avant de démarrer le stage, j'avais déjà eu l'occasion de m'initier à l'utilisation de réseaux de neurones profonds pour effectuer des tâches de classification, d'abord dans le domaine de la reconnaissance d'images avec la classification de chiffres manuscrits de la base de données MNIST[3], puis dans le domaine du traitement automatique du langage pour la classification de phrases anglaises selon qu'elles évoquent un sentiment positif ou négatif. Pour cela j'avais utilisé le framework TensorFlow comme une boite noire sans me soucier de la théorie ou des détails du fonctionnement des réseaux de neurones.

Pour le sujet du stage, il m'a été proposé d'étudier et de tester un modèle dit hybride, car il présente les caractéristiques d'un réseau de neurones artificiels et tient compte de règles linguistiques classiques, c'est-à-dire explicites et codées à la main, nous y reviendrons.

Mon rôle fut dans un premier temps de me documenter, de me former sur de nombreux aspects, notamment sur le traitement automatique du langage dans lequel je n'avais que peu d'expérience, et également de consolider mes connaissances concernant le deep learning. J'ai souvent échangé avec les chercheurs du laboratoire pour avancer dans ma compréhension du sujet et j'ai également assisté à des présentations données par ceux-ci. Dans un second temps, j'ai dû à mon tour effectuer la présentation du modèle qu'on m'avait demandé d'étudier à

[2] « Apprentissage automatique » en français

[3] Célèbre base de données distribuée par Yann LeCun contenant 70 000 images de chiffres manuscrits

l'équipe qui allait m'accompagner au long du stage et qui travaillait sur des problématiques similaires. Enfin, j'ai travaillé à l'élaboration de scripts permettant d'atteindre les objectifs fixés en début de stage, c'est-à-dire de tester le modèle FCM sur de nouvelles données[4].

Le travail était rythmé par une réunion hebdomadaire, souvent très importante, lors de laquelle nous faisions le point sur mon avancée, rectifions les objectifs à atteindre en fonction des éléments apportés et discutions de la stratégie à adopter pour la semaine suivante.

1.2 Présentation du LIS

Le LIS Laboratoire d'Informatique et Systèmes[5] est une nouvelle structure issue de la fusion de deux UMR : le Laboratoire d'Informatique Fondamentale de Marseille (LIF) UMR 7279 et le Laboratoire des Sciences de l'Information et des Systèmes (LSIS) UMR 7296. C'est une Unité Mixte de Recherche (UMR) sous tutelles du Centre National de la Recherche Scientifique (CNRS) et de l'Université Aix-Marseille (AMU).

Ses locaux sont situés sur les campus de Saint-Jérôme (c'est sur celui-ci que j'ai effectué mon stage) et de Luminy à Marseille et sur le campus de l'Université de Toulon. Ce laboratoire regroupe les activités de recherche relevant principalement des sections 06 et 07 du CNRS et des sections 27 et 61 du CNU. Le LIS fédère plus de 375 membres ; 190 permanents chercheurs et enseignants chercheurs, plus de 125 doctorants, plus de 40 post-doctorants et 20 IT/IATSS.

Le LIS est structuré autour de 4 pôles que sont les Sciences des données, le Calcul, l'Image et l'Analyse des Systèmes permettant la mise en évidence de groupements d'équipes, avec quelques sujets partagés qui sont étudiés soit d'un point de vue fondamental, soit sur des plans plus applicatifs.

Une des caractéristiques notables du LIS se situe dans la multidisciplinarité des compétences qu'il regroupe, ce qui en fait un laboratoire de référence au niveau régional. Les chercheurs et enseignants-chercheurs sont impliqués dans les différentes formations de l'Université d'Aix-Marseille, notamment à Polytech mais également à l'Ecole Centrale de Marseille.

C'est dans le pôle des sciences des données et plus particulièrement dans l'équipe DIMAG (Data, Information & content MAnagement Group) que j'ai été accueilli pour ce stage. Cette équipe se consacre au développement de modèles et d'algorithmes au cœur des systèmes d'information (SI d'entreprise, Web, bibliothèques numériques etc.)

[4] Sur les corpus reAce 2005 et Semeval 2018

[5] http://www.lis-lab.fr

1.3 Le Deep Learning

Avant de rentrer dans le détail des activités du stage, il convient d'introduire les notions importantes qui reviendront fréquemment dans le rapport.

L'apprentissage profond via réseaux de neurones artificiels ou « deep learning » est un ensemble de méthodes d'apprentissage automatique s'articulant autour de l'idée qu'un réseau constitué de structures atomiques relativement simples (les neurones artificiels) peut abstraire et modéliser des données complexes de manière efficace.

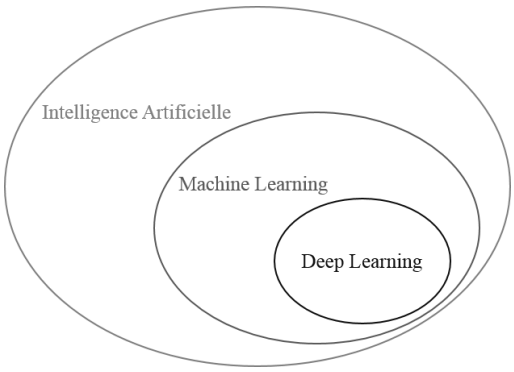


Figure 1 - La place du Deep Learning dans l'Intelligence Artificielle

Ces méthodes de deep learning forment un sous-ensemble du machine learning, qui vient lui-même s'inscrire dans l'ensemble plus vaste de l'intelligence artificielle (cf. Figure 1). Il est essentiel de garder à l'esprit que même si l'apprentissage profond est aujourd'hui très en vogue, il n'est qu'un sous-domaine particulier des techniques utilisées en intelligence artificielle et ne doit pas être employé comme synonyme de celle-ci.

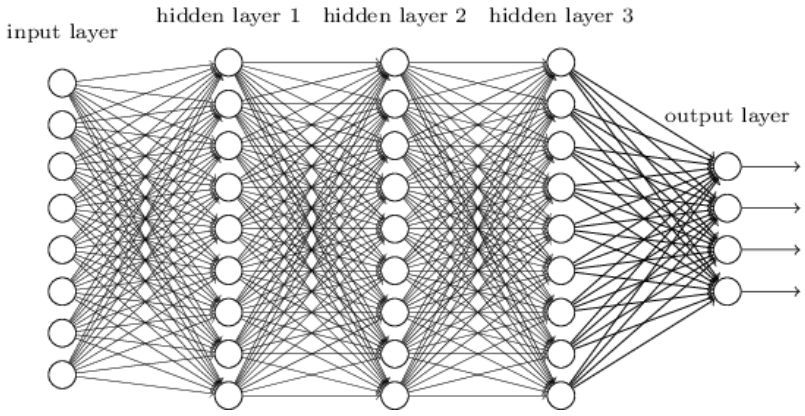


Figure 2 - Réseau de neurones profond

La Figure 2 est une représentation schématique d'un réseau de neurones profond classique. On y distingue de nombreux cercles représentant les neurones artificiels, reliés entre eux par des arcs formant des connexions orientées. Ce réseau contient, de gauche à droite, une couche d'entrée, trois couches cachées et une couche de sortie présentant quatre neurones.

Des réseaux de neurones similaires à celui-ci sont employés dans de nombreux programmes informatiques afin de répondre à des problèmes précis. Ce sont dans les domaines de la reconnaissance d'image, du traitement du langage ou encore de la reconnaissance de la parole que le deep learning a pour l'instant fourni les résultats les plus spectaculaires, permettant souvent de dépasser les meilleures performances des méthodes de machine learning classiques.

Pour illustrer le fonctionnement d'un réseau de neurones comme celui présenté ci-dessus, nous prendrons l'exemple de la reconnaissance d'images représentant des chiffres manuscrits en noir et blanc dans le dataset du MNIST.

Figure 3 - Echantillon de données du MNIST

Ces images ont une taille de 28x28 soit 784 pixels. On peut raisonnablement imaginer que l'on traduira chaque pixel en un nombre plus ou moins grand selon sa teinte (cf. Annexe 1). Pour une image donnée, chacun de ses pixels va être fourni en entrée au réseau de neurones, ces inputs (au nombre de 784 donc) vont se propager à travers les connexions et neurones du réseau qui, s'il a été préalablement bien entraîné, devrait prédire avec une bonne probabilité de réussite le chiffre contenu dans l'image.

Toute la difficulté et la prouesse des réseaux de neurones résident dans la phase d'entrainement. Il n'y a aucune raison à priori pour qu'un ensemble de neurones artificiels connectés entre eux donne de meilleurs résultats que du pur hasard. En effet, lorsqu'un réseau n'a pas encore été entrainé, ses prédictions sont hasardeuses et ne présentent aucun intérêt.

Cependant, lorsque l'on commence à entrainer le réseau et que celui-ci se trompe, on peut mesurer son erreur et la quantifier. En reprenant l'exemple du MNIST, il est intuitif de construire un réseau avec dix neurones en sortie (un pour chaque chiffre) et avec quelques outils mathématiques, faire en sorte que chacun de ces neurones de sortie donne une probabilité que le chiffre qu'il représente soit le chiffre donné en entrée du réseau.

Ainsi, si l'on donne une image du chiffre 4 à notre réseau et que celui-ci prédit 100% de chances que ce soit effectivement un 4 (donc avec 0% pour tous les autres chiffres), c'est une prédiction parfaite et l'erreur[6] mesurée est nulle. A contrario si la distribution de probabilités est différente, le réseau commet une erreur plus ou moins grande et sa mesure grandit en conséquence.

Lors de la phase d'entrainement, on met à l'épreuve le réseau en lui soumettant un grand nombre d'exemples à classifier et on mesure à chaque fois l'erreur qu'il commet. On vient ensuite inspecter chaque connexion du réseau pour la renforcer ou l'affaiblir, ces modifications sur les connexions[7] du réseau, si elles sont faites intelligemment, vont changer sa prédiction pour tendre vers un meilleur résultat.

Ici l'intelligence réside dans les outils utilisés pour effectuer ces modifications, citons la méthode de rétropropagation du gradient, qui utilise des dérivées partielles pour comprendre dans quelle mesure chaque connexion est impliquée dans l'erreur du réseau, souvent couplée à l'algorithme de descente du gradient stochastique qui sert à effectuer les modifications en tenant compte des nombreux exemples d'entrainement avec un temps de calcul raisonnable.

Il faut toutefois noter que la non-transversalité des réseaux de neurones artificiels n'en fait pas une technologie miraculeuse à la lisière de l'intelligence artificielle forte[8]. Une instance d'un réseau de neurones ne peut traiter que d'un problème, c'est-à-dire qu'un réseau entraîné à effectuer une tâche ne peut pas apprendre à traiter une autre tâche supplémentaire, on ne peut donc pas obtenir un seul réseau de neurones qui serait capable de répondre à de nombreuses problématiques.

Il est également tentant de faire un parallèle direct avec le fonctionnement du cerveau humain, et même si la structure d'un réseau de neurones artificiels rappelle fortement celle des neurones et synapses biologiques, il est plus raisonnable de les considérer comme une inspiration abstraite et de ne pas (encore?) parler de bio-mimétisme.

Enfin, on parle de réseau de neurones profond lorsque le nombre de couches cachées du réseau est supérieur à 1, sinon on parle simplement de réseau de neurones. Notons tout de même que ces termes sont très souvent utilisés pour désigner la même chose, cela peut s'expliquer par le fait qu'aujourd'hui en pratique, quasiment tous les réseaux utilisés comportent de nombreuses couches cachées, parfois même jusqu'à plusieurs centaines.

---

[6] Les techniques pour mesurer l'erreur sont nombreuses et variées

[7] En réalité, on ne modifie pas uniquement les connexions mais aussi un nombre que l'on appelle biais, différent pour chaque neurone

[8] Idée d'une machine capable de produire un comportement intelligent et une conscience de soi

1.4 Le Traitement Automatique du Langage

Le traitement automatique du langage (TAL) ou plus précisément traitement automatique du langage naturel (TALN) est un domaine multidisciplinaire impliquant la linguistique, l'informatique et l'intelligence artificielle. Ses applications principales sont la reconnaissance automatique de la parole et du langage naturel, ainsi que la génération de ce dernier.

Le TAL diffère assez largement de la théorie des langages que j'ai eu l'occasion d'étudier à Polytech mais présente parfois des concepts similaires. Il m'a fallu me former et apprendre au jour le jour de nombreuses notions relatives au TAL, en voici un bref condensé qui sera utile tout au long du rapport.

Le champ du traitement automatique du langage naturel couvre de très nombreuses disciplines et tâches, certaines d'entre elles ont des applications directes dans le monde réel tandis que d'autres sont des sous-tâches servant de fondations pour les précédentes.

1.4.1 Syntaxe

Les outils utilisés au niveau de la syntaxe permettent le plus souvent d'agir à l'échelle des mots, sur la manière dont ils sont agencés et sur les relations entre eux.

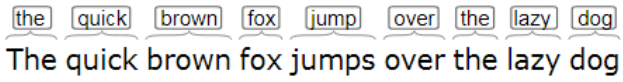


Figure 4 – Lemmatisation

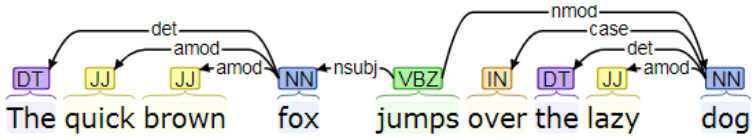


Figure 5 - Parsing de dépendances et étiquetage morphosyntaxique

Des tâches comme la segmentation de mots dans un texte, l'étiquetage morphosyntaxique, le parsing de dépendances ou encore la lemmatisation sont très communes et faciles à mettre en place avec des outils comme le Stanford CoreNLP[9] (Manning et al., 2014), ou des bibliothèques Python comme Spacy ou NLTK.

[9] Dans ce chapitre, tous les diagrammes ont été obtenus avec la version en ligne du CoreNLP de Stanford

La Figure 4 représente une phrase lemmatisée, c'est-à-dire que l'on extrait le lemme pour chaque mot. Le lemme représente en quelque sorte le mot dans sa forme la plus neutre, sans conjugaison par exemple, c'est pour cela qu'ici seul le mot « jumps » se trouve modifié.

La Figure 5 représente une phrase avec son étiquetage morphosyntaxique (Part-of-Speech tagging ou POS en anglais) en couleurs, ce sont les rôles que jouent chaque mot dans la phrase, on voit donc que le mot « fox » est tagué NN (noun) qui signifie « nom » en français ou encore le mot « jumps » tagué VBZ qui signifie un verbe à la troisième personne du singulier. Elle inclut également des arcs qui vont d'un mot à l'autre, ces arcs représentent le chemin de dépendances de la phrase qui lie les mots les uns aux autres en partant généralement du verbe.

1.4.2 Sémantique

La sémantique est l'étude de la signification du langage, de ce que l'on veut énoncer. De nombreuses tâches complexes dans le TAL existent au niveau sémantique. On y trouve par exemple la traduction automatique, la reconnaissance d'entités nommées, l'extraction de relation entre entités nommées, l'analyse de sentiment, la désambiguïsation lexicale ou encore la génération du langage naturel.

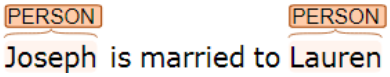


Figure 6 - Reconnaissance d'entités nommées

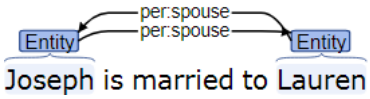


Figure 7 - Extraction de relation entre entités

Les Figures 6 et 7 illustrent deux tâches citées dans le paragraphe précédent. Dans la première, les deux entités nommées de la phrase sont reconnues et annotées, ici ce sont deux personnes mais cela aurait pu être un lieu, une organisation, une quantité ou encore un produit, on parle de reconnaissance ou d'extraction d'entités nommées. Dans la seconde, la relation « spouse » qui signifie époux en français, est extraite entre ces deux entités, on parle d'extraction de relation.

1.4.3 Autres

Il existe encore de nombreux autres niveaux d'abstraction et domaines auxquels s'applique le TAL, comme la reconnaissance de la parole, la génération de parole à partir de texte ou encore le résumé automatique. Dans ce rapport nous n'évoquerons pas ou peu de notions qui n'ont pas été présentées ici.

2 Détail des activités

Le sujet du stage étant centré sur une publication scientifique[10], il m'a fallu étudier en détail ces travaux pendant approximativement deux semaines lors d'une phase de documentation où j'ai également découvert le monde du traitement automatique du langage. Dans un second temps j'ai pu passer à la pratique en me familiarisant avec les outils développés par les auteurs du FCM pour les utiliser, le but final étant de tester le modèle sur de nouveaux corpus disponibles au LIS.

Tout au long de la phase d'implémentation (donc après la documentation) il y a eu un va-et-vient constant entre le besoin de planifier stratégiquement une approche de travail concrète et le besoin d'acquérir des connaissances théoriques plus importantes sur le domaine du TAL ou du deep learning afin de pouvoir mieux cerner mon sujet et échanger avec les chercheurs.

De très nombreuses pistes se sont montrées infructueuses ou trop complexes à mettre en place dans un lapse de temps aussi court et il a été nécessaire de contacter deux des auteurs du FCM pour leur demander conseils quant à la démarche à suivre.

A travers ce travail j'ai également découvert l'utilisation du langage Python sur des problématiques de TAL, qui m'a impressionné par sa simplicité et sa rapidité.

2.1 Présentation du FCM

Ici sera présenté le travail de documentation fourni essentiellement durant les deux premières semaines du stage.

Le FCM est un modèle de réseau de neurones profond dans le domaine du TAL ayant pour but procéder à l'extraction et la classification[11] de relations sur des entités nommées dans un texte. Sa particularité réside dans le fait que celui-ci construit une représentation des structures linguistiques du texte en utilisant à la fois les caractéristiques lexicales des mots (via word embedding) et les caractéristiques non lexicales en capturant le contexte autour d'un mot (via hand-crafted features).

Pour comprendre comment fonctionne le FCM, il convient de présenter ce que sont le « word embedding » et les « hand-crafted features ».

[10] http://www.cs.cmu.edu/~mgormley/papers/gormley+yu+dredze.emnlp.2015.pdf

[11] On parle d'extraction lorsque l'on doit d'abord trouver les entités dans le texte puis déterminer leur relation, on parle de classification lorsque les entités sont déjà données et que l'on cherche uniquement la relation

2.1.1 Word Embedding

Le sens des mots est quelque chose qui échappe totalement à un ordinateur, qui les représente comme une suite d'octets arbitraire sans pouvoir en tirer d'information. Le mot « chat » est aussi proche de « félin » que de « lave-vaisselle » et cela n'aide pas à réduire la complexité de certaines tâches en TAL qui ont besoin d'un apport sémantique pour fonctionner.

Pour résoudre ce problème, on utilise des méthodes de word embedding (ou « plongement de mots » en français) afin de construire des représentations lexicales ayant un sens pour l'humain, c'est-à-dire des représentations où les mots similaires (respectivement différents) ont une représentation très proche (respectivement très éloignée).

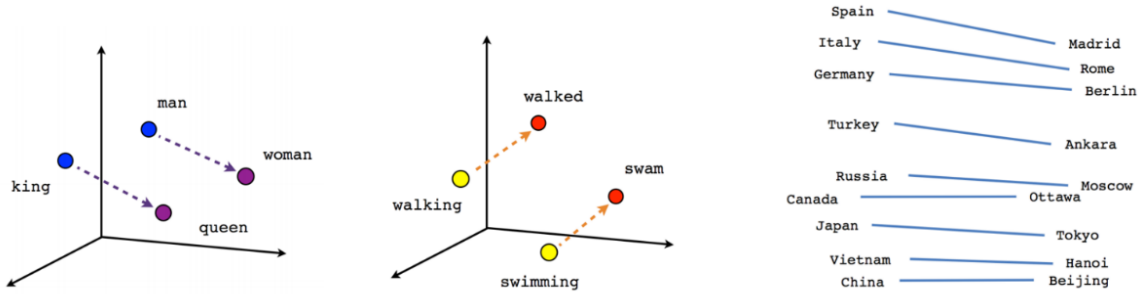


Figure 8 - Word Embedding

Le word embedding permet de représenter chaque mot d'un dictionnaire par un vecteur de nombres réels d'une dimension plus ou moins grande[12]. Il est construit grâce à des techniques de deep learning[13] et permet d'obtenir des propriétés lexicales et sémantiques intéressantes. Ainsi on peut obtenir des représentations de mots facilitant certaines analogies du type « woman est à man ce que queen est à king » (cf. Figure 8).

Enfin, le word embedding permet d'avoir une représentation compacte d'un lexique, là où souvent il était nécessaire de représenter chaque mot par un vecteur de la dimension du dictionnaire entier, constitué uniquement de 0 et d'un seul 1 pour désigner de quel mot il s'agit.

En résumé, le word embedding permet de traduire un mot en une suite de nombres.

[12] Le nombre de dimensions pour un word embedding est typiquement compris entre 100 et 500

[13] Une excellente démonstration de la construction d'un word embedding : https://ronxin.github.io/wevi/

2.1.2 Hand-crafted Features

Les hand-crafted features (ou tout simplement features) sont des caractéristiques extraites pour chaque mot d'une phrase. Ces caractéristiques peuvent être perçues comme de simples questions adressées au mot et à son contexte, on demande par exemple si le mot est un adjectif, s'il est précédé d'un verbe ou encore si c'est une entité dans la phrase. En fait il existe un très grand nombre de features plus ou moins complexes permettant de capturer des informations différentes.

Une feature est facilement représentable sous la forme d'une fonction binaire du type :

$$f_1(x, y) = \begin{cases} 1 \; si \; y \; est \; un \; adjectif \; ET \; le \; mot \; précédent \; est \; un \; nom \\ 0 \; autrement \end{cases}$$

Où *y* est le mot étudié et *x* son contexte (i.e. les mots qui l'entourent).

Dans le cadre du FCM, on choisit un ensemble de quelques features (cf. Annexe 2) que l'on va appliquer sur les mots d'un corpus, on se demande par exemple si le mot étudié w_i fait partie d'une entité, s'il est juxtaposé à une entité, s'il se trouve quelque part entre les deux entités ou encore s'il fait partie du chemin de dépendances. Chaque mot se verra ainsi attribuer un vecteur binaire (donc différent de celui généré par le word embedding) contenant le résultat des features.

2.1.3 Fonctionnement du FCM

Lorsque l'on évoque la particularité du FCM en disant qu'il construit une représentation des structures du texte en utilisant à la fois des caractéristiques lexicales et des caractéristiques non lexicales, on dit en fait qu'il crée une abstraction des phrases du texte en s'aidant des deux outils introduits en amont (cf. Word Embedding et Hand-crafted Features).

Nous allons à présent voir en détail comment cette représentation est construite et pourquoi peut-on raisonnablement estimer qu'elle capture des informations pertinentes sur le texte. Le FCM prend en entrée une phrase et ses annotations pour chaque mot (générées par exemple grâce à un étiquetage morphosyntaxique) et se construit autour de trois étapes principales.

Etape 1 il décompose la phrase annotée en substructures (« sous-structures » en français), chaque substructure est en fait un mot avec ses annotations.

Etape 2 le FCM extrait les features (cf. Hand-crafted features) pour chaque substructure (mot), et obtient ainsi un vecteur binaire pour chaque mot de la phrase. Sur la Figure 9, un vecteur (ou colonne) est lié à un mot dans la phrase, f_1 représente donc les features du premier mot. L'intitulé des features se trouve sur la gauche, on trouve par exemple la première feature qui indique pour chaque mot si celui-ci se trouve sur le chemin de dépendance entre les entités M_1 et M_2.

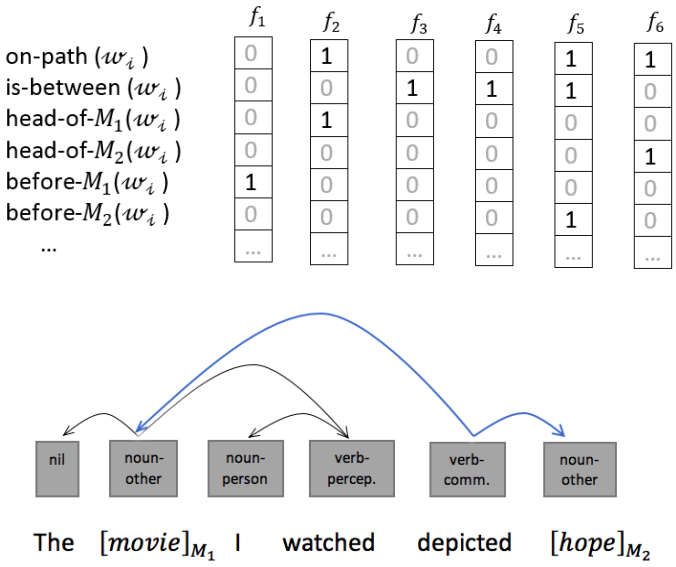


Figure 9 - Feature vectors, chaque vecteur (colonne) est attaché à un mot de la phrase. La phrase annotée se trouve en bas de l'image, ses entités sont marquées par *M1* et *M2* et le chemin de dépendance est indiqué par des arcs (bleutés lorsqu'ils se trouvent entre les entités)

Toujours dans cette deuxième étape, le FCM combine ces vecteurs de features avec le word embedding de chaque mot, le word embedding étant également un vecteur, contenant des nombres réels cette fois-ci. Cette combinaison est effectuée en prenant le produit cartésien des deux vecteurs (cf. Figure 10), chaque mot de la phrase est ainsi associé à une matrice de nombres réels. Cette matrice constitue une représentation abstraite du mot et est appelée **substructure embedding.**

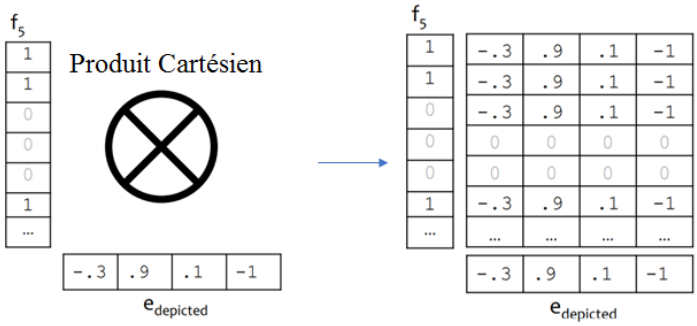


Figure 10 - Substructure embedding

On la note aussi $h_{w_i} = f_{w_i} \otimes e_{w_i}$ avec f_{w_i} le feature vector du mot *i* et e_{w_i} le word embedding du mot *i* et $\otimes$ le produit cartésien.

Etape 3 le FCM fait la somme de toutes les substructures embeddings (matrices) de la phrase pour obtenir une matrice finale appelée **annotated sentence embedding**.

On la note également $e_x = \sum_{i=1}^{n} f_{w_i} \otimes e_{w_i}$

En résumé, le FCM construit une abstraction de la phrase qui lui est donnée, d'abord en découpant chaque mot avec ses annotations (substructure), ensuite en créant pour chaque substructure une matrice de nombres obtenue à partir du word embedding et des features du mots (substructure embedding), enfin en faisant la somme de ces matrices pour obtenir une matrice finale qui constitue sa représentation de la phrase d'entrée (annotated sentence embedding).

Le FCM est un réseau de neurones profond, cependant sa structure est un peu particulière (cf. Figure 11), j'ai d'ailleurs mis longtemps à comprendre en quoi il en était bien un. Pour cela il m'a fallu étudier en détail la théorie derrière les réseaux de neurones, ce qui a également compté dans ma phase de documentation. Les paragraphes qui suivent deviennent plus techniques et s'adressent à un public initié.

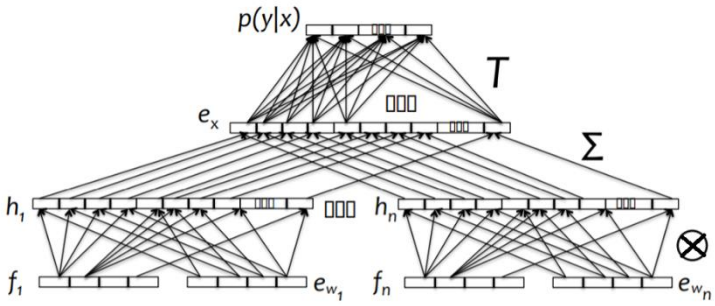


Figure 11 - Le FCM représenté comme un réseau de neurones

Nous avons vu dans l'introduction au deep learning qu'un réseau de neurones prenait des données en entrée (ici une phrase) et effectuait une prédiction (ici le réseau trouve quel est le type de la relation entre les entités de la phrase). Nous avons également vu que pendant la phase d'entrainement, on demandait au réseau d'effectuer des prédictions sur des exemples puis l'on modifiait ses connexions internes (paramètres) si celui-ci commettait une erreur plus ou moins grande.

Dans le FCM, lorsqu'une phrase a été fournie en entrée et quelle a été transformée en annotated sentence embedding (matrice de nombre), le réseau va effectuer une prédiction sur la relation *y* entre les entités de la phrase. On dit qu'il cherche à déterminer *y* sachant $x = (M_1, M_2, S, A)$[14].

[14] Avec *M* les entités, *S* la phrase et *A* ses annotations

Dans le FCM on utilise un tenseur *T* formé de matrices de la même taille que celle de l'annotated sentence embedding, *T* va jouer le rôle de paramètre (équivalent aux connexions du réseau) et servir à établir un score pour chaque relation possible[15]. Il y a donc autant de matrices dans *T* que de relations dans l'ensemble des relations *L*. T_y désigne ainsi la matrice de score pour la relation *y*.

Le score pour la relation *y* est obtenu par $s_y = T_y \odot (f_{w_i} \otimes e_{w_i})$ où $\odot$ est le Frobenious inner product[16] des deux matrices.

L'équation 1 montre comment on obtient une probabilité pour la relation *y* sachant le contexte $\boldsymbol{x}$ et les paramètres *T* et **e** (**e** est l'ensemble des word embeddings et pourra également être traité comme un paramètre du modèle). Cette méthode pour obtenir des probabilités à partir de scores bruts est appelée fonction softmax et est couramment utilisée en deep learning.

La constante de normalisation est donnée par Z et somme sur tous les labels possibles $y' \in L$

$$P(y|\boldsymbol{x}; \mathrm{T}, \mathbf{e}) = \frac{\exp(\sum_{i=1}^{n} T_y \odot (f_{w_i} \otimes e_{w_i}))}{Z(x)}$$

$$\text{Avec } Z = \sum_{y' \in Y} \exp(\sum_{i=1}^{n} T_{y'} \odot (f_{w_i} \otimes e_{w_i}))$$

Équation 1 - Prédiction en FCM

Maintenant que l'on sait comment le modèle effectue des prédictions pour chaque label de relation possible, nous allons voir comment entrainer celui-ci afin de minimiser l'erreur.

Tout d'abord la fonction de coût, permettant de mesurer l'erreur du FCM lorsqu'il effectue une prédiction (cf. Equation 2).

$$\ell(D; T, \mathrm{e}) = \sum_{(\mathrm{x},y) \in D} \log P(y|\mathrm{x}; T, \mathrm{e})$$

Équation 2 - Fonction de coût du FCM (cross-entropy loss function)

Avec *D* l'ensemble des données d'entrainement.

[15] En extraction de relation, il y a un nombre fini de relations possibles qui constitue l'ensemble *L*

[16] Prend deux matrices et renvoie un nombre (multiplie élément par élément et somme le tout)

Le calcul des gradients du FCM se fait par rétropropagation (i.e. application répétée de la règle de dérivation sur les fonctions composée).

On définit le vecteur *s* contenant les scores de tous les labels (relation) pour une annotated sentence embedding $s = \left[\sum_i T_y \odot \left(f_{w_i} \otimes e_{w_i}\right)\right]_{1 \le y \le L}$

L'Equation 3 indique comment le résultat de la fonction de coût (l'erreur) varie en fonction d'un changement du score.

$$\frac{\partial \ell}{\partial s} = \left[\left(I[\, y' = y] - P(y'|\mathrm{x}; T, \mathrm{e})\right)_{1 \le y \le L}\right]^T$$

Équation 3 - Dérivée de la fonction de coût par rapport au vecteur de scores

Où I[*x*] vaut 1 si *x* est vrai et 0 sinon.

L'Equation 4 représente les gradients pour le paramètre *T*, c'est-à-dire la direction et la proportion dans laquelle il faut modifier *T* pour obtenir de meilleurs résultats et réduire l'erreur.

Intuitivement on constate que si la prédiction est exacte ($y = y'$) le terme de gauche va s'approcher de 0 (car la probabilité *P* sera proche de 1). Comme ce nombre va multiplier la matrice de droite, le résultat sera une matrice dont les éléments sont proches de 0, ce qui est désirable car si la prédiction est exacte, on ne veut pas trop modifier les paramètres de $T_{y'}$.

$$\frac{\partial \ell}{\partial T_{y'}} = \left(I[\, y = y'] - P(\, y'|\mathrm{x}; T, \mathrm{e})\right) \cdot \sum_{i=1}^{n} f_{w_i} \otimes e_{w_i}$$

Ou dans sa forme généralisée $\frac{\partial \ell}{\partial T} = \frac{\partial \ell}{\partial s} \otimes \sum_{i=1}^{n} f_{w_i} \otimes e_{w_i}$

Équation 4 - Gradients pour le paramètre T

Enfin, l'Equation 5 représente les gradients pour le paramètre **e** (les word embeddings)

$$\frac{\partial \ell}{\partial e_w} = \sum_{i=1}^{n} \left(\left(\sum_y \frac{\partial \ell}{\partial s_y} T_y \right) \cdot f_i \right) \cdot I[w_i = w]$$

Équation 5 - Gradients pour le paramètre **e**

2.1.4 Intérêt du FCM

Grâce à sa manière d'abstraire, le FCM donne aux mots similaires une représentation matricielle proche. Ces mots qui ont à la fois la même fonction dans une phrase (donc un vecteur de features similaire) et un word embedding proche vont être interprétés comme des structures équivalentes.

Ce qui n'est pas le cas des mots dont la fonction ou le word embedding diffèrent, leur représentation matricielle va nécessairement s'éloigner car lors du produit cartésien entre le vecteur de features et le vecteur de word embedding, la modification de l'un des deux vecteurs entrainera la construction d'une matrice totalement différente (c'est facilement visible lorsque l'on change le vecteur de features, les lignes à 0 ne seront pas les mêmes).

La complexité du FCM est moindre que celle des modèles de réseaux de neurones classique (convolutifs par exemple) : $O(snd)$ avec s le nombre moyen de features à 1 par mot, n la longueur de la phrase et d la dimension du word embedding.

2.2 Travaux réalisés

Ici sera présenté le travail fourni lors de la suite du stage, après la phase de documentation et de compréhension du modèle FCM.

2.2.1 Pistes abandonnées

Nous avions convenu lors des premières réunions avec l'équipe du laboratoire, qu'afin d'utiliser le FCM sur de nouveaux corpus, ce qui constitue le but du stage, je devais me servir de la solution[17] développée par l'un des auteurs du FCM, Matthew Gormley. Cette implémentation a été faite en Java dans un framework très large et généraliste, développé pour répondre à de nombreuses problématiques, et dont seulement une partie spécialisée est dédiée au FCM.

Après de nombreuses heures à explorer le projet (ne possédant pas de notice d'utilisation), à fouiller le code source (souvent dépourvu de commentaires), et à tenter de le faire marcher, j'ai fini par contacter son concepteur qui m'a encouragé à utiliser cette implémentation du FCM en essayant de me familiariser avec les modules de test. J'ai suivi ses recommandations et malgré des résultats encourageants, j'ai conclu que l'utilisation de cette implémentation n'était peut-être pas adaptée et trop lourde pour la durée et le sujet de mon stage, sachant également qu'une autre implémentation du FCM possédait une notice d'utilisation et était facilement utilisable.

[17] Disponible sur https://github.com/mgormley/pacaya-nlp

Cette piste s'est avérée infructueuse, mais cela m'a permis de comprendre l'intérêt de produire un travail facilement réutilisable et documenté.

Nous avions également évoqué, avec mon responsable de stage, la possibilité de créer une implémentation nouvelle du FCM en se basant sur le framework de deep learning TensorFlow développé par Google, si le temps le permettait. Mais cela ne fut pas plus réaliste sur une durée aussi courte (deux mois) et bien que je sois particulièrement intéressé à l'idée d'acquérir de l'expérience avec TensorFlow, nous n'avons pas décidé de suivre cette piste.

2.2.2 Présentation des corpus

La communauté scientifique a régulièrement besoin de tenir à jour et comparer les travaux des chercheurs et organise fréquemment des compétitions internationales autour de problématiques précises. Le but de ces concours est d'avoir un cadre commun pour effectuer des comparaisons pertinentes qui permettront d'établir un état de l'art, c'est-à-dire un répertoire de ce qui se fait de mieux à une instant *t* dans un domaine précis. Cela permet également d'identifier les problèmes récurrents et d'orienter la recherche dans une direction désirable.

Dans le domaine du traitement automatique du langage, on trouve des corpus de référence qui permettent aux équipes scientifiques de tester leurs travaux sur un modèle commun. J'introduis ici deux de ces corpus sur lesquels j'ai eu l'occasion de travailler durant mon stage.

Le Semeval[18] est un concours tenu presque chaque année, constitué d'une vingtaine de tâches parmi lesquelles figurent l'extraction et la classification de relation. Chacune de ces tâches propose un corpus de texte sur lequel les chercheurs vont tester leur modèle (cf. Annexe 3 – Extrait du Semeval 2010). Lors du stage j'ai eu l'occasion d'utiliser le FCM sur les versions 2010 et 2018 de ce corpus en relation et classification de relation.

Le reAce[19] est un corpus spécifique à l'extraction de relations. C'est un corpus payant que le laboratoire du LIS possède et sur lequel j'ai eu l'occasion de faire marcher le FCM (reAce 2005), ce qui constituait un des objectifs annoncés du stage.

Un aspect majeur de mon travail durant ce stage fut de travailler avec ces corpus, qui ont des formes totalement différentes. Comme souvent en data science, le prétraitement des données demande beaucoup d'efforts et ralentit considérablement le processus de recherche.

[18] Dans sa version 2018 http://alt.qcri.org/semeval2018/index.php?id=tasks

[19] https://catalog.ldc.upenn.edu/LDC2011T08

2.2.3 Ma contribution

Une autre implémentation[20] du FCM (en C++) créée par Mo Yu, un de ses auteurs, est disponible en ligne et dispose d'une notice d'utilisation pour faire fonctionner le modèle. On y trouve également un exemple permettant d'utiliser le FCM sur le corpus Semeval 2010 relativement facile à mettre en œuvre et qui permet de reproduire les résultats publiés sur ce corpus.

Cette facilité de mise en œuvre et la capacité de reproduire rapidement les résultats attendus sur mon propre ordinateur furent des facteurs déterminants qui m'ont permis de choisir cette solution pour utiliser le FCM plutôt que celles mentionnées en amont.

Cette version plus légère présente néanmoins des défauts, sa notice d'utilisation est quelque peu erronée et certaines instructions du code en C++ sont spécifiques à Linux et m'ont posé des problèmes lorsque j'ai tenté de la faire marcher dans mon environnement Windows.

Après avoir effectué quelques modifications mineures au code source pour pouvoir le compiler et l'exécuter sur ma machine, tout a fini par fonctionner et j'ai pu aisément tester le FCM sur le corpus donné par défaut (Semeval 2010) en faisant varier les hyper-paramètres du modèle, comme le taux d'apprentissage ou encore le nombre d'epochs lors de l'entrainement.

La raison pour laquelle je n'ai pas uniquement utilisé un environnement Linux pendant le stage vient du fait que j'ai souvent travaillé en dehors des horaires du laboratoire, la plupart du temps chez moi afin d'avancer convenablement, je n'avais donc pas accès aux machines du LIS sur lesquelles Ubuntu est installé. J'étais également trop occupé par les problèmes techniques et théoriques du stage pour me lancer dans une installation complète de Linux sur mon ordinateur, et j'ignorais surtout que je risquais de rencontrer des problèmes à cause de Windows.

Afin d'apporter ma contribution aux recherches du laboratoire, je devais parvenir à faire fonctionner le FCM sur des corpus pour lesquels il n'avait pas encore donné de résultats. Nous avions fixé cet objectif au corpus reAce 2005 et avons par la suite ajouté le corpus Semeval 2018, toujours pour la tâche de classification de relations.

Pour d'atteindre cet objectif, le principal défi fut dans un premier temps de comprendre le format d'entrée du corpus utilisé par Mo Yu[21] dans son implémentation du FCM. Le modèle ne prend pas directement le corpus tel qu'il est présenté dans l'Annexe 3 et nécessite un prétraitement conséquent, comme nous l'avons vu dans la présentation du FCM, celui-ci a besoin d'une phrase et de ses annotations afin de construire sa représentation matricielle du texte.

---

[20] Disponible sur https://github.com/Gorov/FCM_nips_workshop

[21] Nous lui avions posé la question mais celui-ci n'a répondu qu'après que nous ayons compris le format

En résumé, nous avions le corpus Semeval 2010 brut (cf. Annexe 3), nous avions sa version prétraitée, annotée et prête à être utilisée par le FCM mais nous ne savions pas comment passer de l'un à l'autre. Ce qui posait un problème de taille car si je voulais effectuer des tests sur de nouveaux corpus, il était important que je comprenne cette conversion. Point positif, grâce à Gaël Guibon, un doctorant du LIS qui travaille sur des sujets similaires, nous avions déjà un script permettant de convertir le corpus reAce 2005 au format Semeval 2010, il ne manquait donc plus qu'à trouver un moyen d'effectuer la conversion du format Semeval 2010 vers le format d'entrée du FCM.

La Figure 12 schématise et rend plus lisible cette situation.

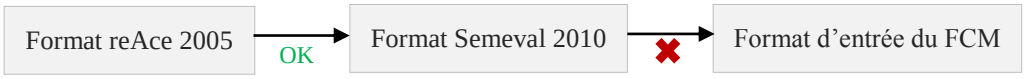


Figure 12 - Situation à mi-parcours

Pour remédier à cela nous avons d'abord contacté Mo Yu en espérant qu'il puisse nous fournir son script de prétraitement des données afin de l'appliquer à un corpus quelconque qui serait au format Semeval 2010. Cependant il nous a indiqué l'avoir perdu depuis quelques années et être dans l'incapacité de le retrouver, même si ce fut dans un premier temps une mauvaise nouvelle, ce fut également une excellente opportunité pour moi de travailler sur cette problématique de prétraitement des données en élaborant un script python permettant d'effectuer la conversion.

Dans la publication du FCM, les auteurs évoquent brièvement l'outil utilisé pour générer les annotations des phrases d'un corpus. Cet outil s'appelle le Super Sense Tagger (SST), il permet de générer des annotations à mi-chemin entre la reconnaissance d'entités nommées et la désambiguïsation de mots[22], il effectue également un étiquetage morphosyntaxique de la phrase.

Grâce au SST, chaque mot de la phrase est associé à trois annotations[23] qui seront utilisées par le FCM. Faire fonctionner cet annotateur fut un véritable défi, premièrement car le lien dans la publication est obsolète et qu'il m'a fallu le chercher longtemps sur internet (je n'ai d'ailleurs toujours pas trouvé la version exacte utilisée par les auteurs du FCM ce qui m'a causé des

[22] Processus par lequel on apporte une information sémantique en levant l'ambiguïté entre des mots identiques qui ont des rôles différents par exemple « un canard » qui peut désigner un animal, un journal, une fausse note etc.

[23] Dans l'ordre POS tag, SuperSense tag, et Named Entity tag

problèmes sur lesquels je reviendrai), mais également car son code source contenait de vieilles instructions C++ qu'il a fallu remplacer. Comme souvent, la notice d'utilisation n'est pas des plus claires.

La Figure 13 représente une phrase brute, puis sa version annotée par le SST. On constate que chaque mot est suivi de cinq annotations au lieu des trois nécessaires, c'est dans mon travail en Python que je supprime les annotations inutiles pour chaque mot.

```
                                The cat is dead

The DT the 0 0 0 cat NN cat B-noun.animal B-E:ANIMAL 0 is VBZ be B-verb.stative 0 0
```

Figure 13 - Phrase annotée par le SST (seulement jusqu'au mot « is » par manque de place)

Une fois un corpus annoté par le SST, il n'est toujours pas prêt à être utilisé par le FCM, aux trois annotations générées par le tagger vient s'ajouter une dernière information binaire pour chaque mot qui vaut donc soit 1 soit 0. Cette dernière annotation est restée longtemps mystérieuse, j'ai d'abord pensé qu'elle devait être générée par le SST mais après avoir essayé toutes les possibilités de celui-ci, nous avons dû contacter de nouveau Mo Yu qui nous a expliqué qu'elle indiquait si oui ou non, le mot faisait partie du chemin de dépendance entre les deux entités de la phrase (cf. Figure 9 où le chemin de dépendances est bleuté).

La Figure 14 représente en premier lieu une phrase au format Semeval 2010 sur les deux premières lignes, puis cette même phrase prétraitée et prête à être utilisée par le FCM sur les quatre dernières.

```
8003    "The school <e1>master</e1> teaches the lesson with a <e2>stick</e2>."
Instrument-Agency(e2,e1)


Instrument-Agency(e2,e1)    2   2   master  8   8   stick
The DT  0   0   0   school  NN  B-noun.group    0   0   master  NN  B-noun.person [...]
1   B-noun.person B-E:PER_DESC
1   B-noun.artifact 0
```

Figure 14 - Conversion du format Semeval 2010 vers le format d'entrée du FCM

On voit sur la Figure 14 que le format d'entrée du FCM (partie du bas) est constitué de quatre blocs ou lignes.

La première indique, dans l'ordre, le type de la relation entre les entités, l'indice de début et de fin de la première entité ainsi que son mot, puis la même chose pour la seconde entité.

La deuxième ligne est tout simplement la phrase annotée par le SST, on voit donc apparaitre un mot, suivi de trois annotations qui lui sont relatives (car on a supprimé deux d'entre elles[24]) auxquelles on ajoute l'information binaire sur le chemin de dépendance ainsi de suite pour tous les mots de la phrase.

Enfin, les deux dernières lignes contiennent, respectivement pour chaque entité de la phrase, sa taille, et deux annotations.

Obtenir ce résultat à partir du format Semeval 2010 m'a pris de nombreux jours de travail, car il a fallu d'abord le comprendre, me former à la manipulation de texte et de fichiers en Python, être très rigoureux dans les manipulations, constamment vérifier que j'obtenais des résultats cohérents grâce à des outils de comparaison de texte et surtout gérer les cas particuliers susceptibles de poser problème et de faire chuter les performances du FCM.

Pour cela j'ai souvent utilisé des expressions régulières plus ou moins complexes afin de chercher dans les corpus (qui sont beaucoup trop grands pour être inspectés à l'œil) des patterns ou des mots susceptibles de créer des soucis. Il y avait par exemple de nombreux problèmes avec les mots anglais du type « a.m. » ou « U.S. », que le SST n'interprétait pas correctement car comme je l'ai dit plus haut, ma version de l'annotateur n'est pas exactement celle utilisée par les auteurs du FCM qui ont publié un lien devenu obsolète. Le traitement de ces mots problématique se fait automatiquement dans mon script (cf. Annexe Script 1).

Enfin, j'ai dû trouver un moyen de compléter les annotations du SST et d'indiquer pour chaque mot d'une phrase s'il se trouve sur le chemin de dépendance entre les deux entités en greffant une dernière annotation à valeur dans {0,1}.

Dans le processus de transformation du corpus, lorsqu'une phrase est traitée, on la transforme en un graphe orienté où chaque mot constitue un sommet et chaque arc relie les mots en fonction du chemin de dépendances dans la phrase (cf. Figure 15). Cette opération est effectuée à l'aide de bibliothèques Python comme Spacy et NetworkX[25].

Une fois que le graphe est créé, il suffit de calculer le chemin le plus court entre les deux entités et d'annoter les mots avec 1 lorsqu'ils se trouvent sur ce chemin ou avec 0 s'ils n'y sont pas.

[24] On supprime le lemme et la reconnaissance des entités nommées

[25] Spacy est une bibliothèque orientée TAL et NetworkX permet de travailler sur des graphes facilement

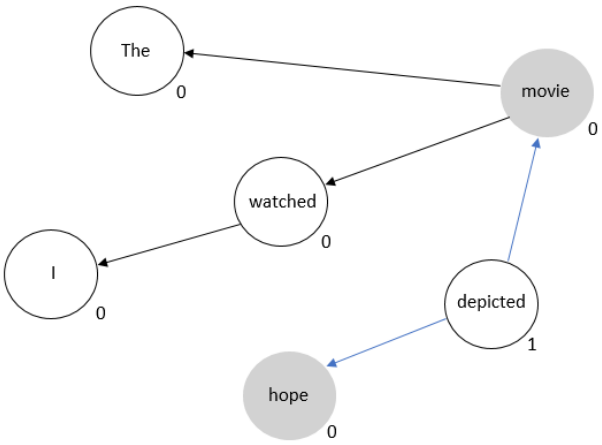


Figure 15 - Graphe orienté de la phrase "The movie I watched depicted hope"

Ce script de conversion (cf. Annexe Script 1) répond ainsi au problème énoncé dans la Figure 12, il agit comme un pipeline prenant en entrée le corpus au format Semeval 2010 et le transforme petit à petit pour arriver au résultat final, faisant appel au SST à mi-parcours. Pour plus de détails[26], voir les commentaires du code.

Grâce à ce travail, j'ai pu transformer les corpus que l'on possédait au format attendu en entrée du FCM, il m'a été possible de l'utiliser rapidement sur le reAce 2005 et d'obtenir des résultats très satisfaisants (les résultats sont reportés dans le chapitre du même nom).

J'ai également pu le tester sur le corpus Semeval 2018, grâce à quoi nous nous sommes rendus compte que les mesures de performance reportées par le FCM n'étaient pas exactement les mêmes que celles utilisées par la communauté des chercheurs. Pour être précis nous utilisions un F1-score pondéré, qui mesure un score par classe (dans notre cas les relations) et fait une moyenne totale de ces scores en accordant plus d'importance aux classes très représentées et moins à celles n'apparaissant que peu de fois.

Les scores reportés en ligne sont souvent sous la forme de Macro F1-score (une simple moyenne de tous les scores par classe) et sont ainsi plus punitifs, car une classe n'étant instanciée que très peu de fois fera fortement baisser la moyenne totale si le modèle ne la prédit pas correctement.

Pour remédier à ce problème, j'ai élaboré un script Python qui calcule divers scores lorsque l'on lance le FCM sur un corpus, dont les F1-scores susceptibles de nous intéresser ainsi qu'un détail des scores par classe (matrice de confusion) grâce à la bibliothèque Scikit-learn.

[26] Il est disponible avec sa notice d'utilisation sur https://github.com/valentinmace/fcm/

Ce script (cf. Annexes Script 2) permet également de lancer le FCM plus facilement, en utilisant plusieurs word embeddings à la suite et en reportant les résultats dans un fichier. Cela permet d'accélérer le processus de lancement du modèle ainsi que comparer les résultats en fonction du word embedding utilisé plus facilement. On peut considérer qu'il constitue une enveloppe du FCM.

La Figure 16 représente un extrait d'un fichier de résultat généré par le script mentionné plus haut.

```
|       Results for FCM on the semeval2018_ corpus       |
 -------------------------------------------------------
| Word Embedding:   wiki_w2vf_dim300
| Learning Rate:    0.005
| Number of Epochs: 30

                      precision    recall  f1-score   support

   PART_WHOLE(e1,e2)     0.3333    0.4762    0.3922        21
     COMPARE(e1,e2)      0.4828    0.5957    0.5333        47
   PART_WHOLE(e2,e1)     0.2308    0.3158    0.2667        19
MODEL-FEATURE(e2,e1)     0.6154    0.5106    0.5581        47
MODEL-FEATURE(e1,e2)     0.4667    0.3043    0.3684        23
        TOPIC(e1,e2)     0.8750    0.3889    0.5385        18
       RESULT(e1,e2)     0.0000    0.0000    0.0000         2
       RESULT(e2,e1)     0.0000    0.0000    0.0000         3
        USAGE(e1,e2)     0.7570    0.7168    0.7364       113
        USAGE(e2,e1)     0.6714    0.7581    0.7121        62

         avg / total     0.6103    0.5915    0.5919       355

| Macro F1:    0.4106
| Micro F1:    0.5915
| Weighted F1: 0.5919
```

Figure 16 - Extrait d'un fichier de résultat, avec la matrice de confusion et les mesures mentionnées

2.2.4 Méthodologie

Tout au long du stage j'ai tenté d'adopter une méthodologie rigoureuse afin de garantir des résultats fiables et reproductibles. Lorsque j'étais confronté à des problèmes nouveaux, j'essayais de m'efforcer à isoler les variables pouvant créer ces problèmes et mesurer leur impact sur mon travail. Je pense notamment au moment où il a fallu décider de générer ou non l'annotation concernant le chemin de dépendances de la phrase. Il a fallu faire des comparaisons à paramètres égaux et générer des corpus de test (avec un corpus témoin) pour finalement prendre la décision d'investir du temps dans son développement car son absence engendrait une baisse de performances.

Un problème qui survient fait toujours apparaitre une combinatoire plus ou moins grande de possibilités pour le traiter et il faut parfois user d'heuristiques pour décider de la voie à suivre.

3 Résultats

Ici sont présentés les résultats obtenus avec le FCM sur les trois corpus disponibles et tous les word embeddings à ma disposition. Pour tous les résultats présentés ci-dessous le taux d'apprentissage a été fixé à 0.005 et le nombre d'epochs à 30, il n'y a pas eu de mise en pratique d'un early-stopping[27] et je sais par expérience que de meilleurs résultats peuvent être obtenus en s'arrêtant plus tôt dans l'entrainement.

3.1 Résultats obtenus

Corpus / Word embedding	reAce 2005	Semeval 2010	Semeval 2018
Googlenews vectors negative 300	0.8023	0.7468	0.4315
Nyt portion gigaword 5	0.7654	0.7471	0.3976
Wiki en dim200 wdsize5	0.7975	0.7502	0.4082
Wiki en dim 300 wdsize5	0.7925	0.7477	0.4252
Wiki w2vf dim 300	0.8110	0.7543	0.4106

Figure 17 - Macro F1-Scores

Corpus / Word Embedding	reAce 2005	Semeval 2010	Semeval 2018
Googlenews vectors negative 300	0.8234	0.7895	0.6169
Nyt portion gigaword 5	0.7931	0.7909	0.5775
Wiki en dim200 wdsize5	0.8234	0.7891	0.5859
Wiki en dim 300 wdsize5	0.8166	0.7858	0.6000
Wiki w2vf dim 300	0.8317	0.7946	0.5915

Figure 18 - Micro F1-Scores

[27] Technique qui consiste à stopper l'entrainement d'un modèle pour éviter le sur-apprentissage

Corpus / Word Embedding	reAce 2005	Semeval 2010	Semeval 2018
Googlenews vectors negative 300	0.8210	0.7871	0.6115
Nyt portion gigaword 5	0.7902	0.7882	0.5769
Wiki en dim200 wdsize5	0.8212	0.7865	0.5856
Wiki en dim 300 wdsize5	0.8143	0.7826	0.6031
Wiki w2vf dim 300	0.8291	0.7927	0.5919

Figure 19 - Weighted F1-Scores

Classifier	**F1**
SMV (Rink and Harabagiu, 2010) (Best in SemEval2010)	82.2
RNN	74.8
RNN + linear	77.6
CNN (Zeng et al., 2014)	82.7
CR-CNN (log-loss)	82.7
CR-CNN (ranking-loss)	84.1
DepNN	82.8
DepNN + linear	83.6
(1) FMC (log-linear)	82.0
(2) FMC (log-bilinear)	83.0
(5) FMC (log-linear) + linear (Hybrid)	83.4

Figure 20 - Performances du FCM sur le Semeval 2010 comparé à d'autres réseaux de neurones et à un SVM

3.2 Interprétation

Les résultats concernant le Semeval 2018 sont à prendre avec précaution car nous n'avons pas eu suffisamment de temps pour nous assurer que sa conversion au format Semeval 2010 était sans défauts. Nous savons même que certains caractères posent problème et sont mal interprétés par le SST, ce qui peut être la cause des mauvaises performances.

Les résultats concernant le Semeval 2010 sont différents de ceux annoncés par les auteurs du FCM, on l'explique par le fait que le scorer officiel du Semeval 2010 ne prend pas en compte la relation « Other » du corpus. Cela ne constitue pas réellement un problème car j'inclus le scorer officiel dans le rendu de mon travail[28] et les résultats annoncés sont bel et bien reproductibles avec celui-ci. Encore mieux, le scorer officiel donne tout de même un score tenant compte de la relation « Other » et celui-ci est identique à celui calculé par mes soins.

[28] https://github.com/valentinmace/fcm/

Les résultats concernant le reAce 2005 sont très intéressants, j'ai eu l'occasion de consacrer suffisamment de temps à ce corpus pour être sûr que le FCM l'utiliserait correctement.

Une tendance se dessine dans les résultats exposés (cf. Figures 17, 18 et 19), si l'on exclut le Semeval 2018 dont les résultats ne sont pas nécessairement pertinents, on constate que pour chaque corpus c'est le dernier word embedding qui donne les meilleurs résultats. Ce word embedding (Wiki w2vf dim 300) est issu d'une publication (Levy and Goldberg, 2014) plus récente et se construit de manière à tenir compte des dépendances syntaxiques dans la phrase. Il présente également l'avantage d'être significativement moins volumineux (d'un facteur 10 avec celui de Google par exemple).

Enfin, les performances du FCM sont proches de l'état de l'art. La Figure 20 compare les résultats de celui-ci sur le Semeval 2010 aux autres modèles neuronaux (ici j'ai réutilisé les données reportées par les auteurs du FCM obtenues grâce au scorer officiel). Sur ces résultats le FCM se décline en trois modèles légèrement différents, premièrement selon que l'on utilise ou non les word embeddings comme paramètres du modèle, et enfin selon qu'il soit couplé à un autre modèle pour affiner les résultats.

4 Bilan

Ce stage m'a permis d'avancer dans la direction qui m'intéresse le plus en informatique : les sciences des données et l'intelligence artificielle. J'ai considérablement renforcé mes connaissances dans le domaine des réseaux de neurones, visité un immense champ de leur application qu'est le traitement automatique du langage et également découvert le monde de la recherche scientifique dans un grand laboratoire comme le LIS.

Je suis particulièrement reconnaissant envers Bernard Espinasse qui m'a permis d'avoir cette expérience dans ce domaine passionnant au moment où j'allais m'orienter vers un stage par défaut dans les technologies du Web, et qui m'a répété plusieurs fois qu'il serait intéressant de réfléchir à la possibilité d'effectuer une thèse, idée que je prends à présent très au sérieux.

Les sciences des données présentent l'intérêt de mettre en valeur la qualité de la formation reçue en école d'ingénieur, car elles demandent à la fois des capacités techniques de maitrise des langages et structures de données, et sont intrinsèquement liées à un aspect théorique et mathématique suffisamment complexe pour n'être accessibles qu'à un certain niveau de formation.

Ce stage m'a également permis de découvrir une méthode de travail souple et rigoureuse, où l'autonomie, la créativité et la capacité de communication avec les autres chercheurs sont des qualités essentielles à mettre en œuvre. Il faut également beaucoup d'opiniâtreté et un peu de chance pour parvenir à appréhender cet environnement complexe et compétitif qu'est le milieu scientifique.

Références

- (Gormley et al., 2015) Gormley M., Yu M., Dredze M., Improving Relation Extraction with Feature-Rich Compositional Embedding Models

- (Levy & Goldberg, 2014) Levy O., Goldberg Y. Dependency-Based Word Embeddings

- (Mikolov et al., 2013) Mikolov T., Chen K., Corrado G. and Dean J. (2013). Efficient Estimation of Word Representations in Vector Space

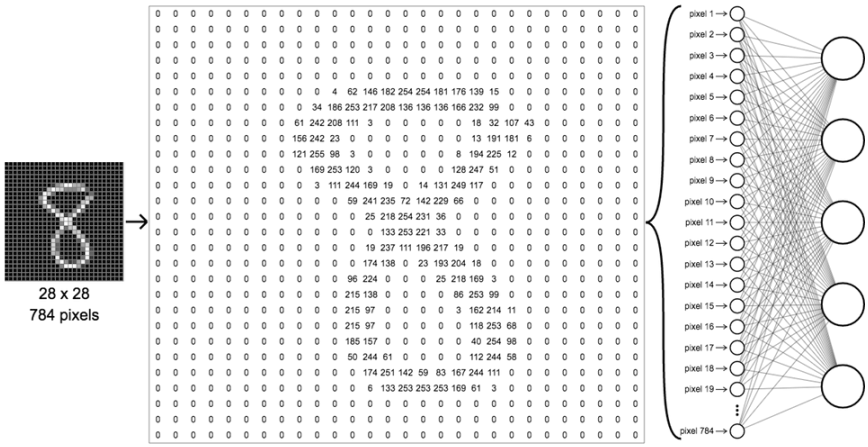
- (Nguyen and Grishman, 2015) Nguyen T.H. and Grishman R., Relation Extraction: Perspective from Convolutional Neural Networks

- (Manning et al., 2014) Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit

- (Levy and Goldberg, 2014) Omer Levy and Yoav Goldberg. Dependency-Based Word Embeddings

Annexes

Images



Annexe 1 - Une image du MNIST transformée en une matrice de nombres donnés en entrée d'un réseau de neurones

Set	Template
HeadEmb	$\{I[i = h_1], I[i = h_2]\}$ (w_i is head of M_1/M_2) $\times \{\emptyset, t_{h_1}, t_{h_2}, t_{h_1} \oplus t_{h_2}\}$
Context	$I[i = h_1 \pm 1]$ (left/right token of w_{h_1}) $I[i = h_2 \pm 1]$ (left/right token of w_{h_2})
In-between	$I[i > h_1] \& I[i < h_2]$ (in between) $\times\{\emptyset, t_{h_1}, t_{h_2}, t_{h_1} \oplus t_{h_2}\}$
On-path	$I[w_i \in P]$ (on path) $\times\{\emptyset, t_{h_1}, t_{h_2}, t_{h_1} \oplus t_{h_2}\}$

Annexe 2- Les features du FCM

```
8001    "The most common <e1>audits</e1> were about <e2>waste</e2> and recycling."
Message-Topic(e1,e2)

8002    "The <e1>company</e1> fabricates plastic <e2>chairs</e2>."
Product-Producer(e2,e1)

8003    "The school <e1>master</e1> teaches the lesson with a <e2>stick</e2>."
Instrument-Agency(e2,e1)
```

Annexe 3 -Extrait du Semeval 2010 pour la tâche d'extraction de relations, les lignes numérotées sont les phrases du corpus avec les entités indiquées par des balises et la ligne qui les suit contient le label de la relation entre ces entités

Code

Script 1 - Conversion

Ci-dessous se trouve le code de mon premier script, il permet de convertir un corpus étant au format Semeval 2010 vers le format d'entrée du FCM.

```
# Valentin Macé
# valentin.mace@kedgebs.com
# Developed for my internship at the LIS lab during my 4th year in the Computer Science
Engineer program at Polytech Marseille

# Feel free to use this code as you wish as long as you quote me as author
# There is a notebook version of this script in the folder 'notebook' wich is easier to read,
but it is in french

#  ----------------------------
# |          PURPOSE           |
#  ----------------------------

# Pupose of this script is to convert a file in Semeval 2010 format to the format used by Mo
Yu as input to FCM (Gormley, Yu 2015)
# FCM publication : https://www.cs.cmu.edu/~mgormley/papers/gormley+yu+dredze.emnlp.2015.pdf_

# Input format :
#
#     4  "A misty <e1>ridge</e1> uprises from the <e2>surge</e2>."
#     Other
#
# Output format :
#
#     Other    2       2       ridge   6       6       surge
#     A DT     0       0       0       misty   JJ      B-adj.all       0       0       ridge   NN
       B-noun.object  0       0       uprises NNS     I-noun.object   0       1       from    IN
       0       0       1       the     DT      0       0       0       surge   NN      B-noun.event
       0       0       .       .       0       0       0
#     1 B-noun.object 0
#     1 B-noun.event 0

# Input format is pretty simple, first line is a sentence with 2 entities between quotes (not
necessarily) and an index, second line is the relation type

# Ouput format contains 4 lines, first line is the relation type, index of begining for 1rst
entity, index of ending for 1rst entity and same for 2nd entity
# second line is the sentence tagged with the SST (SuperSenseTagger), note that the last
feature is not generated by the SST but by the present script
# third line is the size of the 1rst entity plus two of its tags
# fourth line is the same for 2nd entity

# Once this format is generated, it can be directly used by the FCM
# FCM implementation in C++ : https://github.com/Gorov/FCM_nips_workshop

#  ----------------------------
# |            USAGE           |
#  ----------------------------

# Using python 3 (I have not tested with version 2)

# Open a terminal, go to 'raw to formated_script' folder and just use :
# python raw_to_formated.py <input file>
# Example : python raw_to_formated.py semeval2010_train

# Note: The input file HAS to be in the 'data/corpus/raw' folder, and the output will have the
same name and be located in the 'formated' folder
```

```
# Of course the file has to be in the 2010 Semeval format for the script to work

# Setup
import sys
import copy
import numpy as np
import subprocess
import networkx as nx
import spacy
from spacy import displacy
nlp = spacy.load('en_core_web_sm')


file_input = '../raw/'+str(sys.argv[1])
file_output = '../formated/'+str(sys.argv[1])

# Checking corpus funciton
# cheks if each sentence contains 2 entities
def check_corpus(file_path):
    file = open(file_path, "r")
    lines = file.readlines()
    corpus_ok = True
    for i in range(0, len(lines), 3):
        if('<e1>' not in lines[i] or '<e2>' not in lines[i] or '</e2>' not in lines[i] or
'</e1>' not in lines[i]):
            print("There is a problem with the corpus at the line ",i+1)
            corpus_ok = False
    if(corpus_ok):
        print("The corpus seems ok")
print("\n\nChecking corpus integrity...")


# Shortest Dependency Path between 2 words function
# Input : sentence and the two words for which we want dep path
# Ouput : a tab containing words on the dep path
def shortest_dependency_path(doc, e1=None, e2=None):
    edges = []
    shortest_path = []
    for token in doc:
        for child in token.children:
            edges.append(('{0}'.format(token),
                          '{0}'.format(child)))
    graph = nx.Graph(edges)
    try:
        if(e1 in graph.nodes and e2 in graph.nodes):
            shortest_path = nx.shortest_path(graph, source=e1, target=e2)
    except nx.NetworkXNoPath:
        shortest_path = []
    return shortest_path



# Here starts the file processing
# Processing inconvenient words like 'a.m', '100,000' ..., spacing out entities and deleting
sentence indexes

# Cheking corpus
check_corpus(file_input)

file = open(file_input, "r")
lines = file.readlines()

# Note: This preprocessing is necessary because the SST tagger is a little dumber than the one
used by Mo Yu
# And I could not find it
print("Processing inconvenient words like 'a.m', 'U.S' ...")
for i in range(0, len(lines), 3):                          # All 3 lines ... so each sentence
    line_split = lines[i].split()
    for j in range(len(line_split)):
```

```
        if(line_split[j].find(',') > 0):                # if there's a comma inside a word
            line_split[j] = line_split[j].replace(',','')# we delete it
        if(line_split[j].find('.') > 0):
            line_split[j] = line_split[j].replace('.','')
        if(line_split[j] == '.' and j != len(line_split)-2):
            line_split[j] = line_split[j].replace('.','')


    lines[i] = ' '.join(line_split) + '\n'              # rebuild the line from the split

# Spacing out entities for split to recognize them
# Deleting external quotes
print("Spacing out entities and deleting useless quotes")
i = 0
while i < len(lines):
    #lines[i] = lines[i].replace('"', '')               # Old version
    line_split = lines[i].split()
    for j in range(len(line_split)):
        if(j == 1 and '"' in line_split[j]):
            line_split[j] = line_split[j].replace('"','')
        if(j == len(line_split)-1 and '"' in line_split[j]):
            line_split[j] = line_split[j].replace('"','')
    lines[i] = ' '.join(line_split) + '\n'

    lines[i] = lines[i].replace('<e1>', '<e1> ')
    lines[i] = lines[i].replace('<e2>', '<e2> ')
    lines[i] = lines[i].replace('</e1>', ' </e1>')
    lines[i] = lines[i].replace('</e2>', ' </e2>')
    i+=1

# Deleting indexes
for i in range(0, len(lines), 3):
    line_split = lines[i].split()
    if(len(lines[i]) > 0):
        del line_split[0]
    lines[i] = ' '.join(line_split) + '\n'
    i+=1


# Extracting first line of Yu format and adding cleaned sentence
lines_res_temp_1 = copy.deepcopy(lines)               # will contain the first temporary
result
i = 0
for i in range(0, len(lines), 3):                     # All 3 lines ... so each sentence
    line_split = lines[i].split()
    first_line = lines[i+1].replace('\n', '') + '   ' # first we put the relation in
first_line

    tabulation_fin = False                            # Bolean not to tabulate after e2
    j = 0
    while j < len(line_split):
        # Extracting indexes and deleting entity tags
        if('<e' in line_split[j]):
            tabulation_fin = not tabulation_fin
            del line_split[j]
            first_line += str(j) + ' '              # j = entity begining index
            k = j                                     # k is used to go over entity starting
at j
            ent = ''
            while "</e" not in line_split[k]:
                ent += line_split[k]                  # adding the word in the entity to ent
                if("</e" not in line_split[k+1]):     # if the entity is not over we put a
space
                    ent += ' '
                k+=1
            if(tabulation_fin == True):               # if it's the first entity we tabulate
                ent += '      '

        elif('</e' in line_split[j]):
```

```
            del line_split[j]
            j-=1
            first_line += str(j) + ' ' + ent       # j = end entity index

        # Building res_temp_1 (containing blocks of 2 lines)
        lines_res_temp_1[i] = first_line + '\n'
        lines_res_temp_1[i+1] = ' '.join(line_split) + '\n'
        j+=1


# Processing file that will go through SST
file = open("sst/DATA/to_sst.txt", "w")
i = 0
file.write("   ")     # tabulate once, otherwise the tagger will 'eat' the first word of the
file

# Only the sentences will go through SST, the first line will be used later
for i in range(1, len(lines_res_temp_1)+1, 3):
        file.write(lines_res_temp_1[i])
        file.write('\n')
        file.write('\n')
file.close()

# Command to launch SST, see its README for more details
command = ['sst', 'multitag-line', './DATA/to_sst.txt', '0', '0',
'DATA/GAZ/gazlistall_minussemcor',
          './MODELS/WSJPOSc_base_20', 'DATA/WSJPOSc.TAGSET',
           './MODELS/SEM07_base_12', './DATA/WNSS_07.TAGSET',
           './MODELS/WSJc_base_20', './DATA/WSJ.TAGSET',
           './MODELS/CONLL03_base_15', './DATA/CONLL03.TAGSET',
           '>', './DATA/res_sst.tags', '&', 'clean.bat']
print('We call the SST tagger\n')
process = subprocess.Popen(command, cwd="sst", shell=True, stdout=subprocess.PIPE)
process.wait()   # Waiting for the end of SST tagging, results in res_sst.tags
print("End of SST tagging")


# Generating last tag for each word (whether it's on the dep path between entities or not)
# Deleting lemmas

# reading annotated sentences
res_sst = open("sst/DATA/res_sst.tags", "r")
lines_res_sst = res_sst.readlines()
res_sst.close()

# Generating a dependencies graph for each sentence, and finding the shortest path between e1
and e2
# Then checking for each word of the sentence if it's in this path
print("Generating dependency path feature")
for i in range(0, len(lines_res_sst), 3):
    line_split_res_sst = lines_res_sst[i].split()         # Line to modify (adding the tag for
dep path)
    line_split_res_temp_1 = lines_res_temp_1[i].split()   # Sentence without annotation

    # finding e1, its size and starting index
    e1_size = int(line_split_res_temp_1[2]) - int(line_split_res_temp_1[1])+1
    e1_start = int(line_split_res_temp_1[1])
    e1 = lines_res_temp_1[i+1].split()[e1_start]
    # Same for e2
    e2_size = int(line_split_res_temp_1[4+e1_size]) - int(line_split_res_temp_1[3+e1_size])+1
    e2_start = int(line_split_res_temp_1[3+e1_size])
    e2 = lines_res_temp_1[i+1].split()[e2_start]

    # Loading the sentence to be processed with NetworkX
    doc = nlp(lines_res_temp_1[i+1])
    dep_path = shortest_dependency_path(doc, e1, e2)

    # For each word...
    for j in range(5, len(line_split_res_sst), 6):
```

```
        # if it's on the dep path between e1 and e2...
        if(line_split_res_sst[j-5] in dep_path
          and line_split_res_sst[j-5] != e1
          and line_split_res_sst[j-5] != e2):
            # putting it's last tag to 1
            line_split_res_sst[j] = '1'
        else:
            line_split_res_sst[j] = '0'
    lines_res_sst[i] = '      '.join(line_split_res_sst) + '\n'

# Deleting lemmas
print("Deleting lemmas")
i = 0
for i in range(0, len(lines_res_sst), 3):                # All 3 lines ... so each sentence
    line_split = lines_res_sst[i].split()

    j = 2
    while j < len(line_split):
        del line_split[j]                                # Deleting lemma
        j+=5                                             # going to the next one
    lines_res_sst[i] = '      '.join(line_split) + '\n'


# Replacing the sentence by its annotated version in temporary result 1, and adding the last 2
lines

print("Finalizing...")
file = open(file_output, "w")
# In the final result
for i in range(len(lines_res_temp_1)):
    if(i%3 == 0):
        # Writing the first line with the relation, entities and their indexes
        file.write(lines_res_temp_1[i])
        # Adding the next line (annotated sentence)
        file.write(lines_res_sst[i])

        # Splitting them in order get informations for building the last 2 lines
        line_split = lines_res_temp_1[i].split()
        sst_split = lines_res_sst[i].split()

        # findind e1 size and its starting index
        e1_size = int(line_split[2]) - int(line_split[1])+1
        e1_start = int(line_split[1])
        append1 = ''                          # append1 contains the third line of Yu format
(related to e1)
        for j in range(e1_size):
            if(j > 0):
                append1 +=' '
            # Getting tags for e1 (the 2nd and 3rd)
            append1 += sst_split[(e1_start+j)*5+2] + ' ' + sst_split[(e1_start+j)*5+3]
        file.write(str(e1_size) + '  ' + append1 + '\n')

        # Same for e2
        e2_size = int(line_split[4+e1_size]) - int(line_split[3+e1_size])+1
        e2_start = int(line_split[3+e1_size])
        append2 = ''                          # append2 contains the fourth line of Yu format
(related to e2)
        for j in range(e2_size):
            if(j > 0):
                append2 +=' '
            # Getting tags for e2 (the 2nd and 3rd)
            append2 += sst_split[(e2_start+j)*5+2] + ' ' + sst_split[(e2_start+j)*5+3]
        file.write(str(e2_size) + '  ' + append2 + '\n')

file.close()

print("Done")
```

Script 2 – Lancement du FCM

Ci-dessous se trouve le code de mon second script, il permet de lancer le FCM sur un corpus avec des word embeddings différents. Les résultats sont générés dans un fichier, ils contiennent des mesures de performance comme la Macro, Micro et Weighted F1-score.

```
# Valentin Macé
# valentin.mace@kedgebs.com
# Developed for my internship at the LIS lab during my 4th year in the Computer Science
Engineer program at Polytech Marseille

# Feel free to use this code as you wish as long as you quote me as author

#  ----------------------------
# |          PURPOSE           |
#  ----------------------------

# Pupose of this script is to run the FCM on various embeddings (contained in the 'word_emb'
folder)
# and to get results in a file ('macro_f1' folder) including macro f1, micro f1 and weigthed
f1 scores (plus precision and recall)
# Early stopping is not implemented yet (it might be necessary to modify the FCM directly)

#  ----------------------------
# |            USAGE           |
#  ----------------------------

# Using python 3 (I have not tested with version 2)

# Open a terminal, go to 'fcm global.py' folder and use :
# python fcm_global.py <training_file> <testing_file> <number of epochs> <learning rate> [word
embeddings]
# If you do NOT fill the argument for word embeddings, it will run on all the files availables
in the folder
# Example : python fcm_global.py semeval2018_train semeval2018_test 30 0.005

# Note test_file and train_file have to be in data/corpus/formated folder, word embeddings has
to be in data/corpus/word_emb folder

# Setup
import sys
import subprocess
import numpy as np
import os
from sklearn.metrics import classification_report
from sklearn.metrics import f1_score

# Function for running the FCM once
# Returns a string containing results
# Takes as parameters:
# train file and test file which must be in data/corpus/formated folder
# epochs and learning rate
# word emb the name of the file of embeddings which must be in data/word emb
def fcm_and_results(train_file=None, test_file=None, epochs=None, learning_rate=None,
word_emb=None):
        # Command to run the FCM, check its README if you need to modify
        command = ['RE_FCT', '../data/corpus/formated/'+train_file,
                               '../data/corpus/formated/'+test_file, '../results/predict',
                               '../data/word_emb/'+word_emb, epochs, learning_rate]
        print("Loading word embedding... (can take a while)")
        process = subprocess.run(command, cwd="fcm", shell=True, stderr=subprocess.PIPE) #
starting FCM and waiting for it

        file = open("results/predict", "r")          # reading results of FCM
        lines = file.readlines()
```

```
        file.close()

        # Getting predicted labels
        predicted = []                                  # will contain all predictions
        for i in range(len(lines)):
            line_split = lines[i].split()
            predicted.append(line_split[1])

        # Getting real labels
        file = open("results/macro_f1/keys/"+test_file+"_keys", "r")
        lines = file.readlines()
        file.close()
        keys = []                                               # will contain all keys
        for i in range(len(lines)):
            line_split = lines[i].split()
            keys.append(line_split[0])
        set_keys = set(keys)

        # Preparing string with results
        results = '\n\n -------------------------------------------------------\n'
        results += '|       Results for FCM on the '+test_file[:-5]+' corpus       |\n'
        results += ' -------------------------------------------------------\n'
        results += '| Word Embedding:   '+word_emb+'\n'
        results += '| Learning Rate:    '+learning_rate+'\n'
        results += '| Number of Epochs: '+epochs+'\n\n'
        results += classification_report(keys, predicted, target_names=set_keys, digits=4)
                        # Tab showing results
        results += '\n| Macro F1:    ' + "{0:.4f}".format(f1_score(keys, predicted,
average='macro'))      # Various F1 scores...
        results += '\n| Micro F1:    ' + "{0:.4f}".format(f1_score(keys, predicted,
average='micro'))
        results += '\n| Weighted F1: ' + "{0:.4f}".format(f1_score(keys, predicted,
average='weighted'))
        results += '\n-------------------------------------------------------\n'
        print(results)
        return results                  # returning string


#-------------------------------------------------------------------------------------------


# The script starts here

# Arguments for the script
if(len(sys.argv) == 5):
        train_file =    str(sys.argv[1])
        test_file =     str(sys.argv[2])
        epochs =                str(sys.argv[3])
        learning_rate = str(sys.argv[4])
        word_emb_arg =          None
elif(len(sys.argv) == 6):
        train_file =    str(sys.argv[1])
        test_file =     str(sys.argv[2])
        epochs =                str(sys.argv[3])
        learning_rate = str(sys.argv[4])
        word_emb_arg =          str(sys.argv[5])
else:
        sys.exit('\nArguments number invalid')

# Reading embeddings available (in folder data/corpus/word_emb)
embeddings = os.listdir("data/word_emb")

# Creating file containing keys (true labels for the data) using the corpus file
file = open("data/corpus/raw/"+test_file, "r")
lines = file.readlines()
file.close()
for i in range(0, len(lines), 3):                               # deleting useless lines
    lines[i] = ''
```

```
for i in range(2, len(lines), 3):                          # deleting useless lines
    lines[i] = ''
# Creating the key file
file = open("results/macro_f1/keys/"+test_file+"_keys", "w")
for line in lines:
    file.write(line)
file.close()

# Results will contain the string of FCM results
results = ''
# Case where word embedding argument is empty, run the FCM on all embeddings in word_emb
folder
if(word_emb_arg == None):
        for word_emb in embeddings:
                results += fcm_and_results(train_file=train_file, test_file=test_file,
epochs=epochs, learning_rate=learning_rate, word_emb=word_emb)
# Case where word embedding arg is provided, run the FCM with only this embedding
else:
        results = fcm_and_results(train_file=train_file, test_file=test_file, epochs=epochs,
learning_rate=learning_rate, word_emb=word_emb_arg)

# Writing results in a text file
file = open("results/macro_f1/"+test_file+'_results.txt', "w")
file.write(results)
file.close()

print("End, see the results in 'macro_f1' folder")
```