Rapport PRIM TSIA

CHARIF Mahdi

PRIM TSIA 2020-2021

Encadrant : C. Paperman Référent : A. Amarilli

Table des matières

Introduction

Dans le cadre de ma dernière année d'études à Télécom Paris, j'ai décidé de compléter mon choix de cours avec un projet PRIM. Parmi les sujets proposés, celui encadré par M. Paperman a retenu mon attention. En effet, un précédent stage m'avait permis de traiter des données textuelles. Le projet sur lequel j'ai jeté mon dévolu me permettait de continuer sur ma lancée, traitant en partie de l'estimation de la similarité dans un grand corpus de documents.

Après une première phase de discussion autour du sujet et des différents axes que je pourrais explorer, mon choix s'est arrêté sur la base de données des articles de Wikipédia, sous format XML. L'approche était nouvelle pour moi, n'ayant eu à nettoyer des données textuelles qu'à partir de fichiers sous format HTML. Après des échanges constructifs avec M. Paperman, je me suis attelé à la préparation des données. Au cours de cette phase, j'ai dû combiner plusieurs techniques de parsing du code XML. Par le biais de M. Amarilli, j'ai pu accéder à une machine de Télécom Paris. Cette contribution fut précieuse car je devais extraire du fichier originel des millions d'articles. Effectuer cette tâche sur mon ordinateur personnel n'était pas envisageable car ne pouvant pas le « paralyser » pendant plus de 24h.

De même, j'avais décidé, après concertation avec mon encadrant, de considérer la similarité des documents sous deux angles différents. Premièrement, nous étions convenus de récupérer les liens internes de chaque article et de bâtir un graphe représentant toutes les connexions entre ces articles. Deuxièmement, j'avais proposé l'utilisation du modèle linguistique BERT. Contrairement à d'autres modèles, il permet de prendre en considération le contexte d'un mot donné dans une phrase.

La création du graphe formait une étape centrale dans ce projet PRIM. Etant donné la taille conséquente de la base de données des articles, j'ai dû me tourner vers des solutions plus adaptées au problème. Par chance, il s'est avéré que M. Paperman est un contributeur d'un projet de manipulation de graphes sur disque. J'ai donc pu utiliser cette approche et profiter de son aide continue. Une fois ce graphe obtenu, j'ai pu comparer les similarités textuelles des articles et leurs proximités dans le graphe.

Je présenterai dans ce rapport tous les points précédemment abordés, ainsi que les problèmes observés et les pistes d'amélioration. Le code utilisé dans ce projet est disponible sur ma page GitHub. [1]

1. Définition du besoin et choix des données

La première étape était celle de la définition claire du besoin. Une fois mis en contact avec M. Paperman, j'avais pu proposer mes premières idées pour lancer le projet. Je m'étais intéressé aux articles de Wikipédia traitant de films. En effet, ils présentaient l'avantage de présenter des motifs communs comme une section « *Summary* », par exemple. Cette structure m'aurait permis de récupérer les textes m'intéressant grâce à la détection des tags, via la librairie BeautifulSoup.

M. Paperman avait toutefois pointé du doigt que le fait de se fier au titre d'une section n'était pas une bonne solution, certaines pages pouvant proposer des variantes. Il fallait donc se pencher sur une solution plus générale et rigoureuse. Nous avions alors décidé de travailler sur un fichier XML brut des articles de Wikipédia. Ce fichier compressé, d'une taille de 18 Go, contenait un grand nombre d'informations, mais seules certaines d'entre elles nous étaient utiles. Le but était de récupérer le titre de l'article, son texte et ses liens internes. Le fichier a été téléchargé sur un site de référencement des dumps XML de Wikipédia.[2] Nous étions convenus d'exploiter ces derniers afin de produire un très grand graphe (via une solution adaptée). [3]

Un autre point dont nous avions discuté était celui de la similarité textuelle des articles. Une première approche consistait en l'utilisation de Word2vec pour générer des plongements de mots. Toutefois, j'avais pu utiliser des transformers lors d'une précédente expérience pour vectoriser des données textuelles. Cette solution m'a parue être la plus appropriée, les mécanismes d'attention prenant en compte le contexte d'un mot dans une phrase. Pour ce faire, j'avais décidé d'exploiter un modèle pré-entrainé et proposé par Google. [4]

Enfin, nous nous étions également penchés sur le sujet de la création du graphe. La bibliothèque Networkx, que j'avais déjà manipulée auparavant, me semblait être la solution la plus adaptée au problème. Cependant, cette précédente utilisation s'était faite sur un jeu de données relativement petit en comparaison avec celui choisi pour ce projet. Il n'était donc pas envisageable de charger en mémoire un graphe contenant plusieurs centaines de millions de liens. Il fallait donc trouver une solution permettant de gérer ce cas d'usage.

Une fois tous ces points éclaircis, j'ai pu me documenter plus en profondeur sur les technologies à utiliser, en commençant par la manipulation à distance d'une machine de Télécom Paris.

Figure 1 - Etapes définies du projet

2. Documentation et appropriation de nouvelles technologies/librairies

Compte tenu de la taille considérable des données, il n'était pas envisageable d'utiliser mon ordinateur personnel pour faire tourner mes scripts Python. Avec l'aide de M. Amarilli, j'ai pu avoir accès à des machines de Télécom Paris. Celles-ci ont une mémoire vive et une capacité de stockage assez importantes. N'ayant jamais utilisé le protocole SSH, j'ai dû rapidement m'y familiariser. Après un temps d'adaptation, j'ai finalement pu profiter de ces services.

Il fallait toutefois que je trouve un moyen de « déléguer » l'exécution de mes scripts, tout en coupant la connexion SSH. Il n'était pas réaliste d'effectuer des tâches aussi gourmandes d'un point de vue temporel en laissant ma connexion SSH vive. J'ai ainsi pu découvrir Tmux afin de mettre en arrière-plan des tâches assez lourdes. À titre d'exemple, une des manipulations les plus importantes de mon projet requiert 10 jours d'exécution. Il est donc clair que l'appropriation d'un outil comme Tmux paraissait indispensable. [5]

J'ai également dû me familiariser avec les différentes solutions de parsing de documents XML. J'ai pu consulter plusieurs sites et blogs traitant du sujet. Un premier passage sur le document brut est réalisable avec un seul parser mais je ne pouvais pas me contenter de cela. Mon but étant de récupérer les liens internes et le texte nettoyé, j'ai dû me renseigner sur les technologies les plus efficaces pour effectuer ces tâches. C'est ainsi que j'ai jeté mon dévolu sur la librairie mwparserfromhell et me suis renseigné sur un parser HTML, pour éliminer les tags résiduels.

Ayant déjà travaillé sur des données textuelles, j'avais pu utiliser des Transformers pour comparer des documents. J'ai toutefois pris le temps d'assimiler de nouveau ces méthodes, en lisant les papiers consacrés au sujet. Les différents tutoriels présentant des cas d'usage des modèles linguistiques basés sur BERT m'ont été utiles pour adapter cette solution à mon problème. Je me suis également renseigné sur une méthode de stockage et chargement rapide des embeddings. La bibliothèque Annoy de Spotify répond à ce besoin mais n'est pas adapté à des millions d'embeddings. Cette piste est donc restée au stade de l'étude.

Enfin, j'ai pu découvrir une librairie développée par une équipe de l'Inria concernant les graphes. Le besoin défini avec M. Paperman induisant la manipulation d'un graphe d'une taille conséquente, il me fallait trouver une solution différente de celles déjà rencontrées. Cette bibliothèque est nommée Networkdisk et permet de manipuler un graphe de très grandes dimensions, stocké sur disque. Etant basée sur networkx, elle est notamment utile pour la détermination des chemins les plus courts entre des nœuds.

Je me suis alors lancé dans l'écriture de mon code afin d'exploiter le contenu du dump XML de Wikipédia.

3. Processing du fichier XML

Afin de segmenter le fichier, j'ai utilisé la librairie *xml.sax*. Celle-ci m'a permis de détecter les tags indiquant le début et la fin d'une page. Une fois le contenu de l'article récupéré, j'ai dû nettoyer le texte. Tous les frameworks ne fournissant pas le même résultat, mon choix s'est arrêté sur la librairie *mwparserfromhell,* couplée à une courte fonction que j'ai rajoutée. Celle-ci rajoutait une couche de nettoyage du texte via l'utilisation de regex (certains tags HTML avaient échappé au parser).

Ce même parser présente non seulement l'avantage d'éliminer la majorité des tags et autres patterns au code XML qui subsistaient, mais aussi d'extraire les wikilinks. Ces derniers représentent les liens internes d'un article donné. Ils sont centraux dans ce projet : ils seront utilisés pour déterminer les plus courts chemins entre des nœuds au sein du corpus de documents.

J'appliquais ces processus à chaque ligne du document XML. J'insérais alors 4 éléments, identifiant de façon unique un article : le **nom de l'article**, son **texte**, ses **liens internes** et le **nom de l'article de redirection** (s'il existe). Dans le cas contraire, un booléen *False* y était placé. Il était converti en un 0 par SQLite. J'effectuais cette tâche en vérifiant la présence du mot « REDIRECT » au début de mon texte. Le cas échéant, je récupérais le nom de l'article en éliminant les termes propres à la redirection.

Ce dernier point est très important dans la mesure où plus de 40% des articles ne sont que des redirections vers d'autres articles. Cet aspect aura été déterminant pour la suite du projet, en particulier lors de l'encoding des textes. Afin de stocker les liens internes, je les ai convertis en objets json, format pris en charge par SQLite. J'ai utilisé une solution native de SQLite qui m'a permis de convertir les listes de façon implicite en les insérant dans la base de données. Cette solution sera présentée dans la partie suivante.

La création de la base de données aura duré un peu plus de 24 heures, rassemblant exactement 20.115.899 articles. Parmi cet important nombre se trouvent 9.332.553 articles de redirection. Le stockage préalable d'un 0 si l'article ne réorientait pas vers un autre était utile, m'évitant de vérifier cette condition lors de l'encoding des articles. Je n'avais alors qu'à requêter ma base de données en la conditionnant sur la valeur du champ concerné, appelé « redirect ».

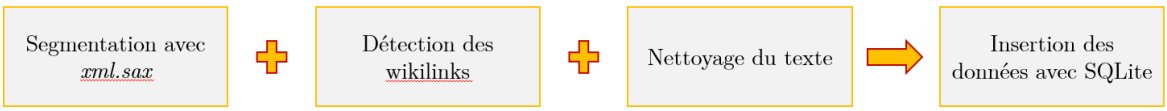

Figure 2 - Etapes du processing du fichier XML

4. Encoding des articles

Dès lors que j'ai pu récupérer les 10 783 346 articles n'étant pas des redirections, je pouvais me focaliser sur l'encoding des articles. Il convient toutefois de présenter la solution retenue mais aussi quelques problèmes rencontrés.

a. Détails du processus

Afin de mener à bien cette étape, j'ai comparé les différentes méthodes de vectorisation de phrases. Le modèle BERT m'a semblé être une solution satisfaisante. En effet, il utilise des *transformers*, qui permettent de prendre en compte le contexte dans une phrase, via des mécanismes d'attention (voir Annexe A). J'ai alors décidé d'encoder les articles et de stocker le résultat de cette manipulation dans une base de données, avec SQLite. Deux colonnes étaient alimentées : celle du titre de l'article Wikipédia, et le vecteur représentant le texte encodé. Compte tenu du fait que SQLite ne supporte pas le stockage de listes ou *arrays*, j'ai dû convertir les vecteurs en des objets *json*. Deux fonctions de sqlite3 présentent un intérêt pratique dans ce cas : register_adapter et register_converter. L'annexe B présente un code plus détaillé. Elles permettent de rendre la conversion des listes en objets Json en listes, et vice versa, sans avoir à utiliser une fonction avant chaque insertion. La figure suivante représente le code utilisé pour effectuer cette tâche :

```
def adapt_array(arr):

    return arr.tobytes()

def convert_array(blob):

    return np.frombuffer(blob)

sqlite3.register_adapter(np.ndarray, adapt_array)
sqlite3.register_converter('array', convert_array)
```

Figure 3 - Register_converter et Register_adapter

Après m'être connecté via SSH, j'ai lancé le processus sur une machine distante. L'encoding étant très long, j'ai utilisé tmux pour créer une session dédiée à cette tâche. L'exécution aura duré plus de 10 jours, produisant un fichier de 43 Go. Une fois cette longue étape passée, je n'aurai plus qu'à encoder un texte ou simplement récupérer l'encoding d'un

article et le comparer avec le corpus des articles. Cette comparaison se fera via l'utilisation de la similarité cosinus, cette dernière étant une métrique usuellement employée. Il est également possible que le graphe, dont je parlerai dans la prochaine partie, soit exploité pour enrichir la notion de similarité entre des articles.

b. Problèmes rencontrés et adaptation

Ayant dû manipuler à distance une machine pour la première fois, je pense qu'il est important de revenir sur les quelques difficultés rencontrées dans cette phase du projet et mon adaptation à ces problématiques.

Premièrement, le traitement de données très volumineuses représentait une approche nouvelle pour moi. L'utilisation d'un simple notebook sur mon ordinateur personnel était inconcevable, à moins de le « paralyser » pendant des jours entiers. Pour y remédier, j'ai utilisé une machine de Télécom Paris.

J'utilisais initialement mes répertoires personnels sur celle-ci. Sans en comprendre la raison, une fois ma session Tmux détachée et la connexion coupée, le lien entre la session et mon répertoire était supprimé. L'administrateur des machines m'a alors apporté une explication : après la fermeture de ma connexion via SSH, les montages NFS étaient supprimés. Il m'a alors proposé d'utiliser un montage permanent. Cette solution a résolu mon problème et m'a permis de lancer l'encoding sans avoir à garder ma connexion vivante.

Il arrivait également que la connexion SSH soit automatiquement avortée (malgré plusieurs tentatives de connexion). Une fois le problème remonté à l'administrateur des machines, j'ai dû attendre jusqu'à ce que tout soit résolu. Ces incidents n'étaient pas très handicapants car je pouvais toujours peaufiner mon code sur ma machine, mais je perdais tout de même du temps par rapport au temps d'exécution considérable de mes scripts. La réactivité de l'administrateur m'a toutefois permis d'avoir de nouveau accès aux machines assez rapidement.

Du point de vue de mon approche habituelle de vectorisation de textes, j'ai dû laisser de côté une solution assez utile pour des données de tailles petites/moyennes. La librairie *Annoy*, développée par Spotify, permet de réduire les dimensions du vecteur et de le stocker dans un fichier. Il suffit alors de le charger de nouveau et d'appeler une fonction propre à cette librairie afin de calculer les similarités. Malheureusement, cette solution n'est pas adaptée aux millions de textes que je cherche à traiter. J'ai donc dû m'adapter et utiliser un format n'ayant pas de contrainte par rapport au volume de données.

Une fois tous ces problèmes détectés et résolus, j'ai pu exporter la base de données des embeddings et l'ai transférée à M. Paperman (afin de lui permettre d'avoir la main sur mes données). Je pouvais donc passer à l'étape suivante du projet.

5. Insertion des edges avec Networkdisk

Après l'obtention de la base de données des articles et celle des encodings, il restait une étape cruciale : la création d'un graphe. Celui-ci permettrait de considérer les chemins entre des nœuds (articles, ici) et donc de rajouter une autre dimension à la notion de similarité, seulement textuelle jusqu'ici.

De nombreuses librairies de manipulation de graphes existent. L'une d'entre elles est assez complète et permet de déterminer assez rapidement les plus courts chemins entre deux nœuds : networkx. Cependant, il fallait trouver une solution adaptée aux dimensions de nos données. Le stockage sur disque d'un graphe apparaissait alors comme optimal. Une fois chargé, le graphe nous permettrait d'accéder assez rapidement aux millions d'edges. Networkdisk répondait à ce besoin.

J'ai donc écrit un script me permettant de générer une nouvelle base de données contenant des centaines de millions de lignes. Pour chaque article, je récupérais la liste des liens internes et ajoutais à une nouvelle liste tous les couples (*article d'origine, lien interne*). Après avoir appliqué cela à la vingtaine de millions d'articles, je pouvais alimenter la base de données créée avec Networkdisk. Cette librairie présente l'avantage d'utiliser SQLite mais aussi de proposer l'affichage d'un logger, afin de vérifier le bon déroulement de l'insertion.

Après quelques heures, je me suis retrouvé avec une base de données de près de 300 millions d'articles. Je pouvais alors analyser les liens entre les pages Wikipédia en chargeant simplement le graphe Networkdisk et en lui fournissant le nom d'une page donnée. Logiquement, j'ai pu observer que les « grosses » pages (celles desquelles on attend logiquement qu'elles aient beaucoup de liens internes) étaient souvent connectées entre elles. Si cette connexion n'était pas directe, elle se faisait alors par l'intermédiaire de quelques pages. L'exemple suivante illustre cette observation :

```
D:\XML\data>py
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path.append("C:/Users/chari/networkdisk/")
>>> import networkdisk as nd
>>> import networkx as nx
>>> G = nd.sqlite.DiGraph(db = "digraph.db")
>>> node1 = "France"
>>> node2 = "New York"
>>> nx.shortest_path(G, node1, node2)
['France', 'American Revolutionary War', 'Onondaga people', 'New York']
```

Figure 4 - Shortest path entre France et Etats-Unis

On observe que le plus court chemin entre la page « France » et la page « New York » lie quasi-directement ces deux articles. Ceci est une façon de prouver la richesse du graphe mais aussi d'entrevoir de bonnes possibilités d'exploitation de celui-ci.

On peut toutefois noter un point intéressant. Le chemin, plutôt court, n'en parait pas moins légèrement long compte tenu du nombre de pages qui relient théoriquement la France à une grande ville comme New York. Etant donné la richesse de la base de données des articles de Wikipédia, il est naturel que certains d'entre eux aient des noms très proches. Ils peuvent même référer à des éléments proches voire-même identiques. Ceci a une explication simple : certaines pages listent les différents articles Wikipédia qui lui sont rattachées (incluant la mention « NomDeLaPage most commonly refers to : »).

Reprenons l'exemple de la ville de New York. La page en anglais n'est pas nommée New York mais New York City. Ceci implique que le chemin diffère entre France et New York et entre France et New York City :

```
D:\XML\data>py
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path.append("C:/Users/chari/networkdisk/")
>>> import networkdisk as nd
>>> import networkx as nx
>>> G = nd.sqlite.DiGraph(db = "digraph.db")
>>> node1 = "France"
>>> node2 = "New York City"
>>> nx.shortest_path(G, node1, node2)
['France', 'Auguste and Louis Lumière', 'New York City']
```

Figure 5 - Shortest path entre France et New York City

Ce résultat semble encore plus convainquant que le précédent. En effet, il paraît plus logique d'avoir un seul nœud entre ces deux pages. Un rapide tour sur la page Wikipédia des Frères Lumière permet de vérifier leur lien avec New York City. Il faut toutefois noter que ceci est différent des pages de redirection, qui pointent automatiquement vers l'adresse cible. Voici un exemple basique (L'annexe C contient le sous-graphe généré) :

```
>>> nx.shortest_path(G, "AccessibleComputing", "Computer")
['AccessibleComputing', 'Computer accessibility', 'Automated teller machine', 'Machine', 'Computer']
```

Figure 6 - Shortest path pour une page de redirection

La page « AccessibleComputing » étant une page de redirection, on doit impérativement passer par la page vers laquelle elle redirige l'utilisateur. Ceci nous permet de tout de même obtenir un « path » pour ce genre de pages. Lorsque l'on veut combiner les résultats du graphe et la similarité textuelle des documents, il suffit alors de récupérer le texte de l'article cible (qui est ici « Computer accessibility »). En somme, ce graphe semble assez riche et sa comparaison avec les embeddings pourrait fournir des résultats probants.

6. Comparaison du graphe et des embeddings

Une fois dotés de toutes les bases de données requises, je pouvais enfin étudier un éventuel lien entre la similarité textuelle des différents articles et leur proximité dans le graphe. Après quelques manipulations du graphe et de la base de données des embeddings, j'ai pu observer des résultats probants, en me basant sur l'observation du « chemin le plus court » (*shortest path* en anglais).

L'idée était de récupérer deux nœuds et de déterminer le chemin optimal les reliant. La fonction *shortest_path* de Networkx est assez pratique pour effectuer cette tâche. Une fois cet élément récupéré, je pouvais calculer la similarité textuelle entre ces deux pages/nœuds. A titre d'exemple, voici le résultat obtenu en appliquant cette demarche pour les pages « France » et « Anarchism » :

```
>>> node1 = "France"
>>> node2 = "Anarchism"
>>> text1 = c.execute("SELECT Embedding FROM Embeddings WHERE Name = 'France'").fetchone()[0]
>>> text2 = c.execute("SELECT Embedding FROM Embeddings WHERE Name = 'Anarchism'").fetchone()[0]
>>> cosine_similarity(text1.reshape(1, -1), text2.reshape(1, -1))
array([[0.15925398]])
>>> nx.shortest_path(G, node1, node2)
['France', 'Albert Camus', 'Anarchism']
```

Figure 7 - Similarité textuelle et plus court chemin entre deux noeuds

Le résultat semble assez logique : la similarité textuelle des deux pages est assez faible. En effet, le texte stocké traitant de la France présente le pays dans ses aspects les plus globaux. Quant à la page décrivant l'anarchie/anarchisme, elle présente le concept dans ses grandes lignes également. Il est peu probable qu'un lien textuel avec la France soit présenté dans le résumé global de la page, d'où la similarité assez faible.

Toutefois, on peut noter un plus court chemin assez logique. La page Wikipédia d'Albert Camus fait référence à son adhésion à ce courant de pensée, il est naturel que les pages « Anarchism » et « France » fassent partie des Wikilinks de la page de l'auteur français. Il est donc logique d'observer un *shortest path* des plus courts, sans pour autant observer une similarité textuelle élevée. Bien évidemment, cette observation n'est pas absolue : on peut très bien avoir deux pages aux contenus textuels proches et présentant un chemin minimal entre leurs nœuds. L'exemple des pages « France » et « Paris » illustrent parfaitement cet exemple. Leur similarité textuelle est relativement élevée (près de 0.7) et ils sont directement reliés dans le graphe.

Cependant, la répétition des tests m'a fait détecter une erreur assez handicapante : certaines pages ont un embedding ne représentant pas leurs textes. Prenons l'exemple de la page traitant de la Chine et comparons-la à la France :

```
>>> node1 = "France"
>>> node2 = "China"
>>> text1 = c.execute("SELECT Embedding FROM Embeddings WHERE Name = 'France'").fetchone()[0]
>>> text2 = c.execute("SELECT Embedding FROM Embeddings WHERE Name = 'China'").fetchone()[0]
>>> cosine_similarity(text1.reshape(1, -1), text2.reshape(1, -1))
array([[0.00073424]])
>>> nx.shortest_path(G, node1, node2)
['France', '1992 Winter Olympics', 'China']
```

Figure 8 - Exemple de pages dépourvues de texte

On observe, certes, que leur proximité dans le graphe est avérée, mais aussi et surtout que leur similarité textuelle est extrêmement faible. Ce dernier point est contre-intuitif dans le sens où les embeddings de deux pays sont censés être proches. Aux premiers abords, je croyais que ce problème était dû à une mauvaise utilisation de BERT mais assez rapidement, je me suis rendu compte que le problème résidait dans le mauvais traitement du document XML, dès le début du projet.

J'avais décidé de ne récupérer que le premier paragraphe de chaque page, partant du principe qu'il suffisait afin d'obtenir des informations textuelles suffisantes sur un article donné. Un obstacle à cette hypothèse est l'opacité du parsing. En effet, devant traiter une vingtaine de millions d'articles, je ne pouvais pas l'adapter à chaque cas de figure. Il s'est avéré que les 2 parsers utilisés pour le nettoyage des données fournissaient certes un résultat satisfaisant, mais laissaient parfois une ligne vierge au début du texte. Je récupérais donc parfois cette même ligne et la stockait comme texte de l'article considéré.

C'est donc pour cette raison que j'ai obtenu des embeddings erronés. A posteriori, je dirais qu'il aurait fallu vérifier que le paragraphe considéré (obtenu avec la fonction *split* des strings) n'était pas une simple ligne vierge. Ensuite, une itération sur le nombre de paragraphes aurait été nécessaire jusqu'à rencontrer un paragraphe plein. Cependant, d'un commun accord avec M. Paperman, nous avions décidé de ne pas retenir cette solution mais plutôt d'utiliser des expressions régulières (regexp).

Mis à part ce problème qui touche quelques centaines de milliers d'articles, j'ai donc tout de même pu comparer la similarité textuelle des articles et leur proximité dans le graphe généré par Networkdisk. Les résultats semblent assez logiques, n'établissant pas forcément une corrélation directe entre une similarité textuelle élevée et une proximité directe. Je pense toutefois qu'il pourrait être intéressant de rajouter cette dimension spatiale du graphe dans la notion de similarité. Pour une page wikipédia A ayant une similarité textuelle identique avec deux pages différentes B et C, on pourrait les distinguer en désignant comme plus similaire la page présentant le *shortest path* le plus petit avec la page A.

En somme, l'exploitation des résultats est intéressante et a même permis de mettre en lumière un problème engendré par le parsing. L'étape suivante consistait ensuite en une optimisation du code.

7. Optimisation du processus

Compte tenu des observations effectuées sur mes données, M. Paperman et moi avions décidé de revoir certaines méthodes employées dans les différentes phases du projet. La première concernait le parsing du dump XML.

a. Parsing XML + regexp

N'ayant pas récupéré le bon texte pour un certain nombre d'articles, je m'étais tourné vers une autre bibliothèque de traitement du code XML et l'utilisation de regexp. Cette nouvelle approche permettrait également de réduire le temps d'exécution du programme.

Il s'est avéré que l'utilisation du parser *mwparserfromhell* ralentissait considérablement le parsing. La bibliothèque de substitution, *lxml.etree*, contient une fonction *iterparse* qui m'a permis de détecter chaque apparition d'un tag de début d'article (soit un tag « page »). M. Paperman a considéré que cette solution était plus pratique que celle précédemment utilisée, permettant notamment une réduction considérable du temps d'exécution. Le nettoyage du texte se faisait alors avec l'utilisation de plusieurs regexp pour détecter les tags à éliminer.

Cette nouvelle approche présente l'avantage d'avoir en quelque sorte le contrôle sur le parsing. Le nettoyage ne se fait plus aveuglément mais avec des outils codés par nos soins. De cette façon, on évite de reproduire l'erreur constatée sur des millions d'articles. On récupère un nombre limite n de mots dans un article afin d'obtenir ultérieurement les embeddings.

b. Stockage de toutes les données avec Networkdisk

Un des autres points abordés était celui du stockage des données. Jusqu'ici, les différentes manipulations étaient effectuées dans les fichiers suivantes :

- *Create_db.py* : l'exécution de ce fichier permettait, à partir d'un fichier XML brut, d'obtenir la base de données SQLite des articles (nom, texte, liens internes et article de redirection s'il existe).
- *Encode.py* : celui-ci rassemblait le code propre à l'encoding des articles Wikipédia, à partir de la base de données obtenue dans la première étape.
- *Graph.py* : ce dernier fichier permet, comme son nom l'indique, de générer le graphe Networkdisk en récupérant les liens internes (wikilinks) après avoir requêté sur la base de données générée par l'exécution de *create_db.py*.

Dans une optique d'optimisation du processus de stockage mais aussi de manipulation des données, nous voulions tout rassembler dans le graphe Networkdisk. De cette façon, il suffirait de fournir au graphe le nom de l'article pour récupérer ses liens internes, son texte et son embedding.

Ce dernier point restait toutefois plus difficile à réaliser. Le texte et les liens internes sont directement récupérables lors du parsing du fichier XML. Nous avions donc rassemblé dans le fichier *create_db.py* les informations récupérées dans l'ancienne version du fichier et l'insertion des nœuds et edges. Toutes les données d'un article donné étaient stockées dans un graphe networkdisk, ce qui rendait la manipulation des données bien plus rapide. Quant aux embeddings, ils demandent à être généré à partir du texte. Cette opération est particulièrement chronophage et retarderait considérablement le parsing, si on l'effectuait à la volée. Il n'était donc pas possible de rassembler strictement toutes les manipulations dans un même fichier. Il fallait séparer l'encoding du reste. Je discuterai dans la prochaine partie consacrée aux axes d'amélioration du projet d'une autre possibilité pour l'insertion des embeddings.

c. Création d'un script Bash

Afin de proposer un exemple d'utilisation de Networkdisk aux futurs utilisateurs de cette bibliothèque, M. Paperman m'avait proposé d'écrire un script Bash regroupant toutes les manipulations faites sur les données. L'idée était de fournir un script exécutable, qui se chargerait d'appeler les fichiers pythons évoqués précédemment, produisant ainsi plusieurs bases de données (ou une seule, sous la forme d'un graphe Networkdisk).

La première étape était celle du téléchargement du dump XML. L'idée était de déléguer la totalité du processus à ce script, afin que l'utilisateur qui découvre Networkdisk n'ait pas à effectuer une seule tache manuellement. Une fois le document téléchargé, les fonctions sont appelées à tour de rôle ; différents messages étant affichés à l'écran pour informer l'utilisateur de l'avancée du processus. L'idée était d'abord de lancer le parsing du document XML, de récupérer toutes les données textuelles et les liens qui nous intéressaient et d'insérer dans une base de données SQLite, générée avec Networkdisk, toutes ces données. La dernière version du script inclut l'encoding des articles mais il n'est pas envisageable de l'effectuer dans la continuité des autres actions. Cette tache dure beaucoup trop longtemps : elle avait duré près de 10 jours. J'ai envisagé une solution à ce problème dans la prochaine partie de ce rapport.

Cette étape a marqué la fin du projet. Il me reste toutefois à revenir sur les acquis de ce projet mais aussi à discuter des axes d'amélioration de celui-ci.

8. Bilan des acquis et axes d'amélioration

Ce projet m'aura demandé de puiser dans diverses connaissances acquises lors de mes stages et études mais aussi d'explorer de nouvelles pistes.

a. Bilan des acquis

La première technique que j'ai découverte lors du projet était celle de la connexion SSH.

Je ne connaissais cela que de nom, j'ai donc rapidement dû me familiariser avec ce concept et ses subtilités. Je considère avoir compris l'utilité de l'utilisation d'une machine distante. Jusqu'ici, je me cantonnais à un usage plutôt « académique » des ressources informatiques : un notebook, utilisé sur mon ordinateur personnel, suffisait souvent à réaliser les tâches que j'avais dû réaliser. Les contraintes de ce projet rendant cela impossible, mon initiation au protocole SSH était certes obligatoire mais elle m'a été très bénéfique. Un exemple concret illustre cela : lors de mes entretiens pour mon stage de fin d'études, nombreuses sont les entreprises ayant été intéressés par cette nouvelle compétence. J'ai également pu découvrir l'outil Tmux, qui s'est révélé être particulièrement précieux quand je devais exécuter un programme pendant plusieurs jours.

Au niveau du code Python, j'ai pu réutiliser certaines connaissances, telles que celles concernant les Transformers BERT. J'avais toutefois peu d'expérience concernant le parsing XML et l'utilisation de Networkdisk. La considération d'un fichier XML aussi volumineux m'était étrangère et j'ai donc dû me renseigner sur les techniques usuelles et m'en imprégner. Ayant seulement utilisé Networkx auparavant, j'avais une connaissance assez superficielle des techniques de manipulation des graphes. J'ai donc pu acquérir plus de connaissances dans ce domaine, ayant le privilège de disposer de l'expertise de M. Paperman sur ce point.

Enfin, ma contribution, certes maigre, au test de Networkdisk était assez intéressante. Pour la première fois, j'intervenais dans le processus de test d'une bibliothèque Python, faisant des retours directement à un de ses contributeurs. Les deux parties sont gagnantes dans ce cas de figure : j'en profite pour tester de nouvelles fonctionnalités et développer ma capacité de test et M. Paperman et ses collègues obtiennent un regard critique sur leur travail.

b. Axes d'amélioration – Pistes à explorer

Concernant les axes d'amélioration de mon travail, j'ai pu cibler plusieurs points, le premier étant celui du choix des regexp.

Les regexp utilisées semblent fournir un résultat satisfaisant (c'est-à-dire un texte bien nettoyé) mais nous ne sommes pas à l'abri de nous retrouver avec quelques imperfections. Des tags pourraient subsister, ou des textes vides pourraient être stockés. Cette phase de parsing est finalement celle qui requiert le plus de minutie afin d'obtenir les résultats les plus fidèles à la réalité possible.

Un deuxième point qui devrait être amélioré est celui de la création du graphe. Comme précisé précédemment, j'avais inséré la quasi-totalité des données dans un graphe Networkdisk après avoir tenté ma première approche. Les seules données qui restaient à être insérées étaient les embeddings. Je pense qu'il serait judicieux de consacrer un script Bash entier à cette opération. De cette façon, on attendrait que le parsing XML soit entier fait, impliquant la création d'un graphe via Networkdisk. Ceci éviterait d'avoir un seul bloc d'opérations durant plus d'une dizaine de jours, sans la certitude d'avoir un graphe fonctionnel. Je préconise donc l'emploi de cette approche afin de proposer aux utilisateurs de Networkdisk un exemple simple et valide.

Je pense également que l'encoding des articles pourrait être amélioré du point de vue du temps d'exécution. Ce dernier est très important et pourrait être diminué par l'emploi d'une autre technique d'encoding. Aujourd'hui, BERT fournit des résultats probants mais certains chercheurs tentent de trouver des modèles, toujours basés sur BERT, mais proposant un encoding plus rapide. Je dirais donc qu'il faudrait constamment se documenter pour choisir la meilleure solution.

Enfin, développer une API (Interface de programmation) serait judicieux. Ceci permettrait de proposer à un utilisateur une interface simple et concise, où il aurait simplement à renseigner le nom des articles dont il veut considérer la similarité textuelle et dans le graphe. Ceci s'inscrit presque dans une logique d'industrialisation du projet, dans le sens où je suis parti d'une idée, l'ai définie avec mon tuteur et aurait pu arriver à un produit final, exploitable par des personnes n'ayant pas participé au projet.

Conclusion

Travailler sur ce projet aura donc été l'occasion de renforcer mes acquis mais surtout de découvrir de nouvelles notions. En ce sens, j'ai trouvé cette expérience enrichissante et formatrice.

Après avoir reçu l'approbation de M. Paperman, j'avais pu définir les contours du projet avec lui. Il avait été décidé, d'un commun accord, de travailler sur la base de données des articles de Wikipédia. Ces derniers présentaient l'avantage d'avoir des données textuelles assez riches mais aussi, et surtout, des liens internes. Ces derniers nous auront permis d'exploiter une bibliothèque en plein développement et sur laquelle M. Paperman travaille. J'avais rapidement pu me familiariser avec les divers outils requis pour remplir les taches à accomplir. Ceci allait de l'apprentissage du protocole SSH à la documentation sur des bibliothèques Python telles que Networkdisk. Une fois cette première phase d'acclimatation effectuée, j'avais pu rentrer dans le vif du sujet, produisant un code Python permettant d'exploiter le document XML.

Une fois le parsing du code effectué, j'ai pu produire un code propre à l'encoding des articles, stockant ainsi les embeddings dans une base de données. Le but était de comparer les similarités textuelles des articles avec leurs proximités dans un graphe. Ce dernier a été produit suite à l'exécution d'un code Python, utilisant la bibliothèque Networkdisk. Ce graphe, stocké sur disque, présente l'avantage de permettre un accès très rapide aux nœuds adjacents d'un nœud (article) donné. Ainsi, je me retrouvais avec mes plongements topologiques et mon graphe, prêt à les exploiter.

Logiquement, j'ai pu observer que la corrélation entre ces deux derniers éléments n'était pas toujours observée. Cependant, un élément problématique apparut : nous n'avions pas pu récupérer les textes de certains articles. Nous avions alors décidé de retravailler le code, délaissant certains parsers au profit d'une approche plus manuelle, via des regexp, notamment. Nous avions également décidé de centraliser toutes les données dans un graphe généré via Networkdisk. La production d'un script Bash vint clôturer ce projet.

Les acquis qui résultent de ce projet sont nombreux. Du travail sur une nouvelle bibliothèque en plein développement (Networkdisk) à l'utilisation du protocole SSH, j'ai pu développer de nouvelles compétences, tout en consolidant mes connaissances. Ce projet peut bien sûr être amélioré, que ce soit au niveau de l'affinement potentiel des regexp, ou via la recherche d'une méthode optimale pour l'encoding des articles.

En somme, ce projet aura été l'occasion de découvrir de nouvelles technologies et approches. La manipulation de données volumineuses requiert beaucoup de rigueur. Je pense que cette expérience me sera bénéfique dans ma future carrière de Data Scientist.

Bibliographie

[1] GitHub (Page consultée le 09/02/2020). Processus parsing, encoding et création du graphe, [En ligne] ; https://github.com/mahdicharif/PRIM/tree/master/XML/processus

[2] Wikimedia (Page consultée le 09/02/2020). Wikipedia dump service, [En ligne] ; https://dumps.wikimedia.org/enwiki/20210201/

[3] GitLab (Page consultée le 09/02/2020). networkdisk, [En ligne] ; https://gitlab.inria.fr/guillonb/networkdisk

[4] GitHub (Page consultée le 09/02/2020). Sentence Transformers : Multilingual Sentence Embeddings using BERT / RoBERTa / XLM-RoBERTa & Co. with PyTorch, [En ligne] ; https://github.com/UKPLab/sentence-transformers

[5] Wiki ubuntu-fr (Page consultée le 09/02/2020). Tmux (terminal multiplexer), [En ligne] ; https://doc.ubuntu-fr.org/tmux

[6] BERT Encoder (Page consultée le 09/02/2020). Peltarion, [En ligne] ; https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/bert-encoder

Annexes

Annexe A

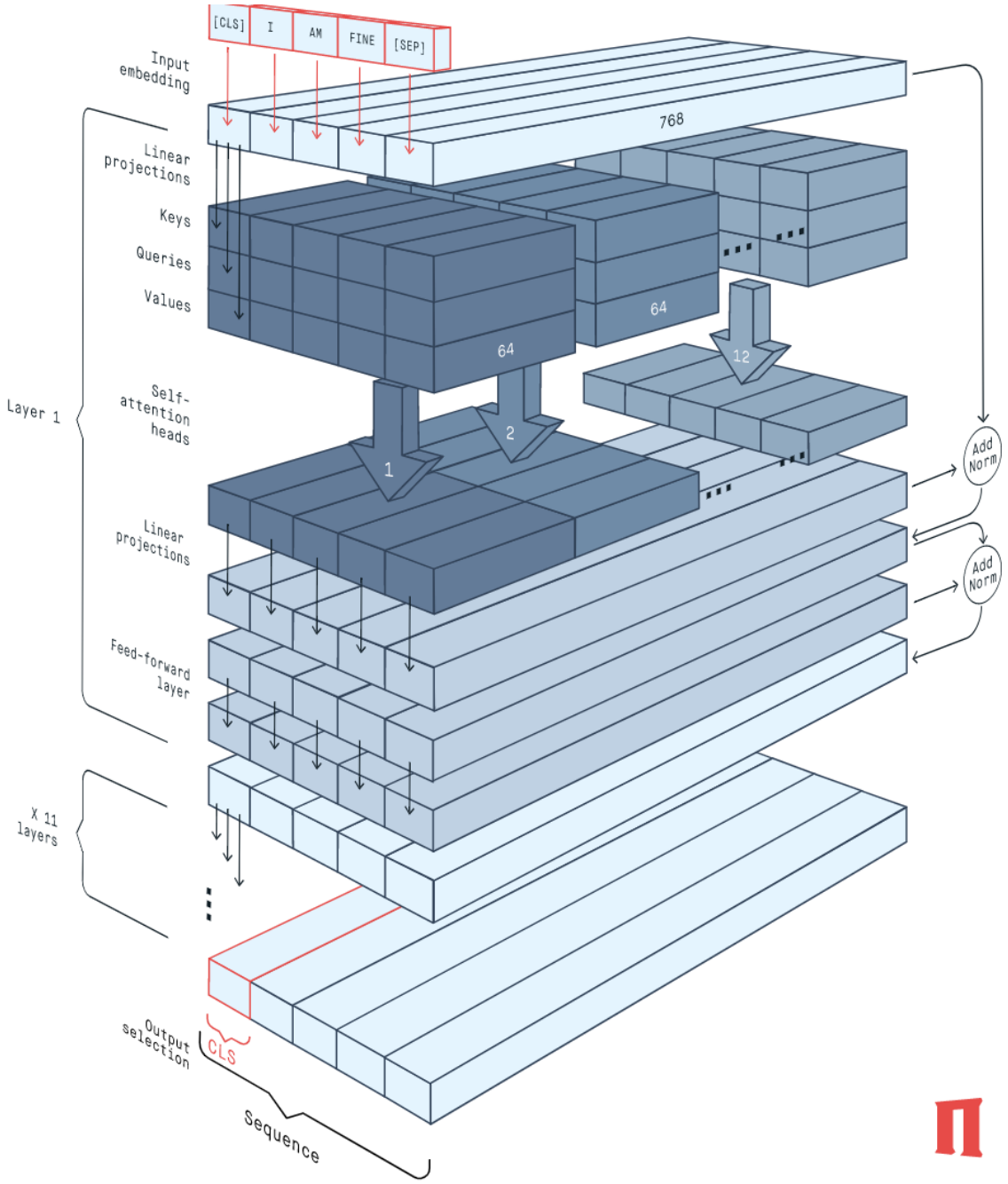

Figure 9 - Structure de BERT [6]

Annexe B

```
import sqlite3
import numpy as np

def adapt_array(arr):
    return arr.tobytes()

def convert_array(blob):
    return np.frombuffer(blob)

# Register the new adapters

con = sqlite3.connect("articles.db")
cu = con.cursor()
qu = "SELECT Name_article, Text_article FROM Article WHERE Redirect = 0"
res = cu.execute(qu).fetchall()
con.close()

print("data loaded")
print("the length of the data is {}".format(len(res)))
conn = sqlite3.connect(path + "embeddings.db", detect_types=sqlite3.PARSE_DECLTYPES)
c = conn.cursor()
c.execute("CREATE TABLE Embeddings(Name text, Embedding array)")
sqlite3.register_adapter(np.ndarray, adapt_array)
sqlite3.register_converter('array', convert_array)
conn.commit()
conn.close()
```

Figure 10 - Utilisation de register_converter et register_adapter dans mon code

Annexe C

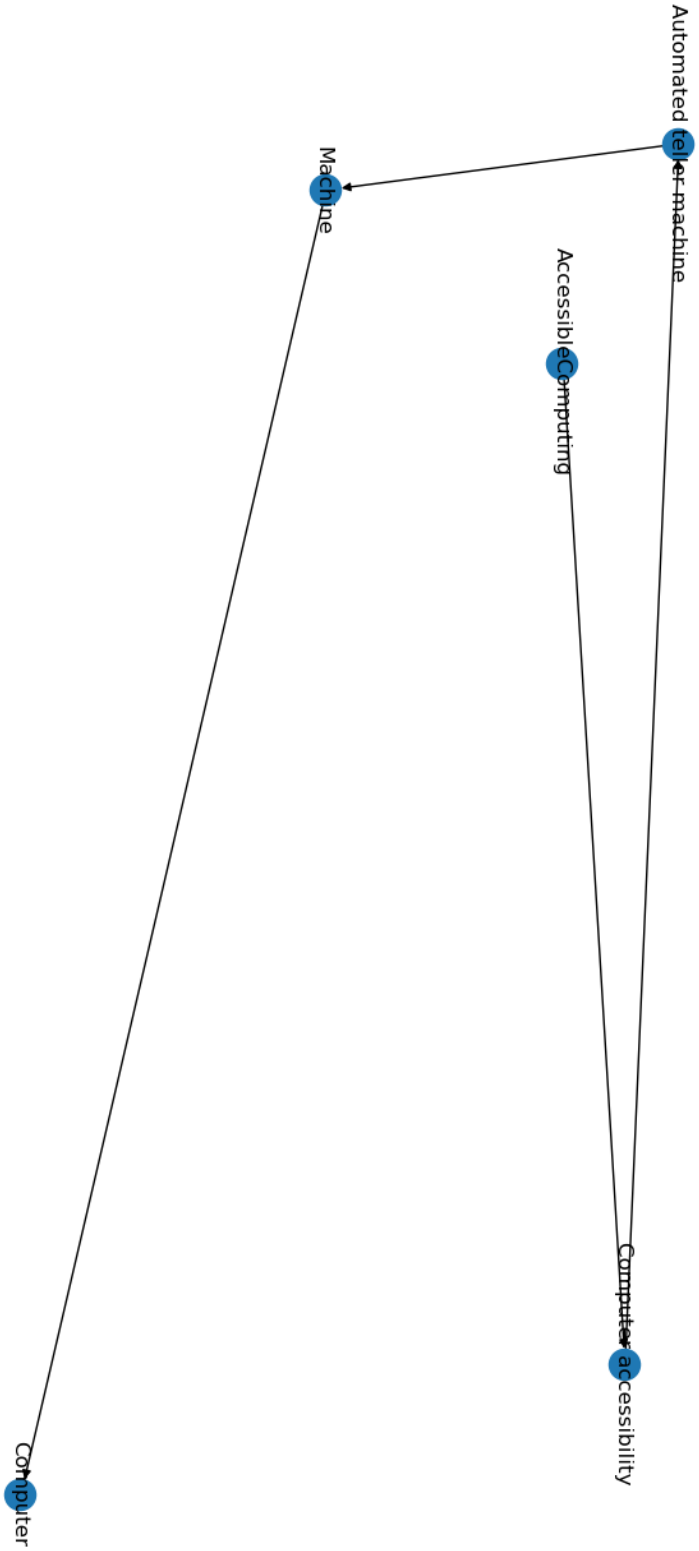

Figure 11 - Sous graphe reliant AccessibleComputing à Computer