Rapport de projet d’Ingénierie Linguistique

Aurélien HERBIN, Morgane PAILLER, Nicolas PETITJEAN,

Laure POINT, Camille Rémi



**Table des matières**

[I. Introduction 3](#_Toc38381674)

[II. Extraction XML 3](#_Toc38381675)

[1. Présentation du langage XML 3](#_Toc38381676)

[2. Extraction des données du fichier XML 4](#_Toc38381677)

[III. Traitement des données 5](#_Toc38381678)

[1. Recherche de règles générales 5](#_Toc38381679)

[2. Présentation des différentes fonctions du code 6](#_Toc38381680)

[3. Implémentations des règles en python 6](#_Toc38381681)

[IV. Annexe 8](#_Toc38381682)

[1. La GUI 8](#_Toc38381683)

[2. Les différents scripts 8](#_Toc38381684)

[V. Conclusion 10](#_Toc38381685)

# Introduction

Dans ce projet d’Ingénierie linguistique, nous avons été amenés à créer un programme qui définit, à partir de l’export de pages Wikipédia en XML, une liste des tueurs ainsi que le nombre de leurs victimes et les personnes qu’ils ont tué. Pour cela, nous avons choisi une lettre, et nous avons répertorié une grande partie des tueurs dont le nom de famille ou le prénom commençait par cette lettre (nous n’avions pas assez de personnes dont le nom de famille uniquement commençait par notre lettre). C’est un exercice assez fréquent dans le domaine du Traitement Automatique des Langues, puisque nous récupérons des données brutes de fichiers (ici XML) afin de les étiqueter avec des outils comme la bibliothèque *nltk* sur Python.

Python est beaucoup utilisé dans ce genre de cas car c’est un langage de programmation impératif qui permet aussi de faire de la programmation orientée objet. Il est facile d’accès et a un grand nombre de bibliothèques déjà existantes pour pouvoir satisfaire ces traitements.

# Extraction XML

## Présentation du langage XML

Le langage est un langage de balisage[[1]](#footnote-1). Cela veut dire *eXtensible Markup Language* et son but était d’envoyer des pages sur le web et qu’elles puissent être traitées comme du code HTML, ce qui explique la similarité entre les deux langages. Le langage XML répond à une norme *unicode* c’est-à-dire qu’elle doit accepter les codages des caractères *UTF-8* et *UTF-16*. Pour simplifier un peu le propos on peut envisager un fichier XML comme un arbre avec une racine qui a généralement pour nom *mediawiki.*  Ensuite, on a plusieurs feuilles comme la feuille *siteinfo* et la feuille *page*. Il peut y avoir plusieurs feuilles *page* car cela va dépendre du nombre de pages que l’on veut exporter.

## Extraction des données du fichier XML

![A screenshot of a cell phone

Description automatically generated]()

III‑1 fonction qui permet de récupérer le texte des balises XML

La fonction *recuperationText()* est une fonction qui prend une chaîne de caractère en paramètre et renvoie une chaine de caractère en sortie. Décortiquons-la un peu plus : la variable qu’elle prend en entrée est le chemin vers le dossier XML. Une amélioration possible de cette fonction serait de, via une interface graphique, demander à l’utilisateur où se trouve le dossier XML avec les pages qu’il souhaite traiter. En fait, l’inconvénient d’enregistrer dans le code le chemin c’est qu’il ne peut pas être changé autrement qu’en le cherchant dans le code ou via un *input.*

Pour pouvoir extraire des données XML via python, il faut utiliser la librairie *Etree* qui permet d’aller chercher le dossier XML. Il faut ensuite l’exploiter via des fonctions comme *getRoot()*  qui renvoie un objet de type Root. Comme on le voit sur l’image on remplit une liste avec toutes les balises *text[[2]](#footnote-2)* dedans puis on transforme via la méthode *join()* la liste en string. Grâce à ça, on retourne un assez grand texte au format *string* avec dedans toutes nos pages exportées.

# Traitement des données

## Recherche de règles générales

L’extraction des noms et prénoms des tueurs est généralement assez simple parce que dans le premier dossier que l’on utilise et qui se nomme : *corpus.xml*. Les fichiers du corpus se présentent sous cette forme :

« *Martha Rendell: killed three stepchildren with chloridric acid in 1907-08; last woman to be hanged in Western Australia* »

Cela nous permet de faire un algorithme qui dit que derrière chaque puce, la première Entitée Nommée représente le tueur. Il faut ensuite réfléchir à comment trouver ses crimes et il existe autant de moyens d’aborder ce sujet que de verbes pour désigner un meurtre. En lisant plusieurs documents on a trouvé quelques verbes récurrents comme : *Poisoning, Assassinated, Demembred, Killed…* Une fois l’extraction de ces verbes est faite il faut voir si on peut déduire des règles générales.

On a donc notre mot cible qui est le verbe et l’on va donc s’intéresser aux mots qui se trouvent autour de ce verbe. Cela nous a permis de déduire des règles comme :

- Après un *who* on aura toujours le verbe *murdered* et un nombre de personne comme par exemple : *who murdered five people …*

- Aprèsun « of », on sait que cela concerne les victimes :

• Cela peut être un **chiffre**, en nombre ou en lettre comme par exemple : *poisoner of three individuals*

• Cela peut être un **mot-clé** : en XML, ça ressemble à [[mot-clé]] : si le mot-clé peut être un nom commun (comme le mot prince), on peut tomber sur un nom propre (qui nous intéresse, car ça va être celui de la victime)

- Au lieu de « of », on peut tomber sur un mot de « viser » comme par exemple *perpetrated* ***against***

## Présentation des différentes fonctions du code

**Tri(doc)**: est une fonction qui prend en entrée un texte et qui ressort une liste qui contient les noms propres associées à leurs positions dans le texte.

**Temporel(doc)**: est une fonction qui prend en entrée un texte et va associer chaque entité nommée à un indice temporel.

**Localisation(doc)**: est une fonction qui prend en entrée un texte et va associer chaque entité nommée à un indice de lieu.

**Association(doc)**: est une fonction qui prend en entrée un texte et va associer le nom du tueur à ses victimes.

**Deduction1(doc1,doc2)**: est une fonction qui prend en entrée 2 textes et va garder les noms propres associés à une date/lieu suivant la liste des victimes.

## Implémentations des règles en python

Le but du programme **tri(doc)** est de reconnaître avant tout, les personnes qui ont été tuées par le tueur : isoler seulement les noms propres ne suffit pas car certains peuvent ne pas être des victimes ou être des noms de bâtiments. C’est pourquoi, il faut commencer à partir des mots (noms communs et verbes) en relation avec le meurtre. On les mettra dans une liste (dans le programme, il est nommé « kill »).

On constate alors qu’il peut y en avoir beaucoup : avec un seul mot, on peut partir sur d’autres par « dérivation » (en ajoutant un suffixe par exemple) ; ce mot est nommé un lexème. Par exemple, avec le mot « kill », on peut obtenir « killing », « killer », « killed », etc. À noter que si le lexème peut être un verbe, on peut le dériver pour en faire un nom commun.

Afin de réduire efficacement la liste des mots en relation avec le meurtre, on y met seulement des mots basiques : lorsqu’on analyse le texte, on « réduit » si possible chaque mot lu en leur lexème (en supprimant les suffixes par exemple).

On remarque aussi que ce soit le lexème ou ses dérivations, ces derniers suivent les règles citées précédemment. De ce fait, se contenter des lexèmes simplifient le programme tout en gardent une certaine qualité de lecture.

De l’autre côté, il faut évidemment reconnaître les noms propres, mais aussi les regrouper : en rapport avec une des règles d’implémentation, après un mot en relation avec le meurtre, le mot « them » se rapporte à plus d’une personne.

Or, si avec « him » ou « her », il suffit de se rapporter au dernier nom propre enregistré, pour « them », il faut savoir jusqu’à combien de noms propres enregistrés, il faut prendre. De ce fait, il faut regrouper des noms propres proches, entre eux. Après avoir lu un nom propre, le programme compte la distance entre ce mot, le dernier nom propre enregistré : s’ils sont assez proches, le mot est mis avec le dernier nom propre enregistré (dans une liste) ; sinon, le programme met ce mot dans un nouveau groupe (en créant une nouvelle liste). À noter qu’en absence d’une virgule ou de « and », le programme rallonge artificiellement la distance afin d’éviter tout confusion : un point provoque la séparation de deux groupes de noms propres entre eux.

Par ailleurs, tout ce qui se trouve entre les noms propres est ignoré : on sait que sur un mot en relation avec le meurtre, l’énumération de noms propres est logiquement terminée et le programme force donc une nouvelle séparation avec les prochains qu’il va rencontrer.

En conséquent, la liste qui va contenir tous les mots du texte est une liste de liste : les noms propres proches entre eux sont regroupés dans une liste ; afin de rendre la lecture de cette liste plus fluide, tous les mots (incluent les noms propres isolés), seront mis dans une liste (qui contient qu’un seul élément). De ce fait, une fois ce regroupement terminé, il suffit de lire cette liste, d’identifier quel est le mot en relation avec le meurtre, puis déduire avec ce qui suit, quel(s) nom(s) propre(s) y est/sont associés (et donc les mettre dans une liste qui contient logiquement les noms des victimes du tueur). À noter que le nom du tueur sera ignoré pour des raisons évidentes…

# Annexe

## La GUI

Nous avons décidé de faire une partie graphique qui peut se lancer indépendamment de la partie en ligne de code pour pouvoir permettre à l’utilisateur d’avoir la possibilité d’importer des pages XML et d’ensuite de pouvoir les analyser via notre interface. Les fonctions restent les mêmes que pour le programme en ligne de code.

On a d’abord pensé à utiliser la librairie tkinter qui est directement disponible avec python 3, mais pour avoir passé une demie après midi dessus à écrire à la main les fonctions et la position de chaque élément graphique dans la fenetre. Nous nous sommes penchés sur pyqt5 avec qtDesigner qui permet de réaliser la partie graphique beaucoup plus vite. Le logiciel est assez complet et nous à éviter de chercher longtemps comment par exemple lier une scrollBare à un Label ou alors faire en sorte que l’utilisateur ne rentre qu’une lettre.

## Les différents scripts

Nous avons décidé de faire plusieurs scripts python pour rendre le code plus lisible :

* **File.py**: c’est le script qui s’occupe de tous les traitements qui n’ont aucun rapport avec l’analyse du corpus. Dans ce script on va trouver les fonctions pour extraire le texte des balise XML, ainsi que des fonctions qui aideront l’interface graphique.
* **WhoKillWho5.py**: c’est le script qui va s’occuper du traitement et de l’analyse du corpus avec des fonctions que l’on a expliquées plus haut. On l’a nommé ainsi car c’est la cinquième version de notre script pour l’analyse de corpus. Il apporte les fonctions qui permettent de déterminer où s’est fait le meurtre.
* **wkwGUI.py** : c’est le script qui permet de lancer l’interface graphique.
* **Launcher.py :** c’est le script final, celui qui lie la partie graphique et la partie ligne de commande.

## Améliorations possibles

Nous avons proposé une solution qui est principalement lié à un fichier xml et ce qui pourrait intéressant c’est faire que notre programme marche pour n’importe quel fichier xml qui aurait dedans un tueur en série. Une autre chose qui nous à beaucoup fait réfléchir c’est la localisation on à pas trouvé de manière convaincante de l’exprimer vu qu’encore une fois notre travail repose sur un seul corpus.

On pourrait imaginer que la liste de tueurs que l’on collecte à partir d’une lettre, s’écrive dans un fichier et que par la suite quand on ouvre un nouveau document xml avec un tueurs la liste de tueurs que l’on a stocké dans un fichier nous aide à approfondir nos connaissances sur ce tueur. C’est-à-dire que nous utiliserions dans un premier temps *corpus.xml*  pour chercher tous les tueurs en série existant et dans un second temps on ouvrirait un fichier comme *Thomas\_Jeffries.xml* et que notre programme affine ses connaissances sur ce Thomas Jeffries à partir des connaissances qu’il à déjà et aussi des nouvelles qu’il reçoit.

# Conclusion

Nous avons rencontré beaucoup de difficultés pendant la réalisation du projet. Il faut savoir que notre projet est construit sur le document *corpus.xml* donc il arrive que sur un document autre que celui-ci notre programme ne fonctionne pas. On a eu des difficultés sur des choses comme la localisation car par exemple *Bogota* c’est une ville et aussi un nom de victime. Dans ce cas précis le programme n’arrivera pas à traiter l’information correctement et n’en tiendra donc pas compte.

Dans les solutions que nous avons apportées il y’a la suivante qui peut poser un problème c’est celle des noms propres composés. Imaginons que nous avons comme lettre un « D » et que notre programme se retrouve face à Cayetano Domingo Grossi. Dans cette situation là le programme va reconnaitre comme prénom Cayetano et comme nom Domingo Grossi. On a pris le parti de dire que si dans un nom composé on a la lettre cible donc ici « D » alors on le met dans notre liste de tueurs en série. Parmi les autres solutions on a décidé de lier la victime à une date comme ça, cela nous permet par la suite d’afficher les résultats plus facilement.

Cet exercice était assez intéressant car il nous à permis d’utiliser plusieurs bibliothèques python et donc de nous familiariser avec les outils du traitement automatique des langues.

1. Source : <https://openclassrooms.com/fr/courses/1766341-structurez-vos-donnees-avec-xml/1766421-quest-ce-que-le-xml> [↑](#footnote-ref-1)
2. Le chemin *"{http://www.mediawiki.org/xml/export-0.10/}text"* fonctionne pour n’importe quel document il n’est pas spécifique au document *corpus.xml*. [↑](#footnote-ref-2)