Manon Ramaroson
Dorine Fontaine
Virgil Rouquette–Campredon
Tristan Tribes

LIEN GITHUB : https://github.com/virgil-rouquettecampredon/TraitementSemantiqueTP2

Partie I. Architecture Système

1. Choix des outils

Pour réaliser ce projet, nous avons décidé d'utiliser le langage **Python 3**. Celui-ci nous permet d'utiliser la très riche librairie **rdflib** (https://github.com/RDFLib/rdflib). Cette dernière nous permet non seulement de lire facilement nos fichiers d'ontologies, mais également d'effectuer des requêtes à l'aide du langage **SPARQL**. Ainsi nous utilisons SPARQL afin de naviguer dans nos fichiers, et extraire les données nécessaires pour le programme.

On utilise également les nombreuses librairies de l'écosystème Python afin d'accélérer le développement. Notamment nous avons utilisé **Tkinter** (https://docs.python.org/3/library/tkinter.html) pour l'interface graphique, ainsi que **py-stringmatching** (https://github.com/anhaidgroup/py_stringmatching) pour ses fonctions de comparaisons de String. Enfin nous avons utilisé **spaCy** (https://spacy.io/) pour apporter plus de fonctionnalités, notamment le découpage des phrases, mais aussi la détection de synonyme et de similarité dans une phrase, de même que le filtrage des mots courants (stop words).

2. Principe de comparaison

Concernant la comparaison, nous récupérons les attributs jugés intéressants sur nos données, et les comparons avec tous les autres attributs du même type des autres œuvres musicales. Nous pouvons activer et désactiver les comparaisons, ainsi que pondérer chaque attributs. Les attributs sont les suivants : Titre(s), Genre(s), Note (une courte description de l'œuvre), Compositeur (mais uniquement l'URI), Gamme et l'Opus.

3. Implémentation

Notre implémentation se découpe en 3 fichiers Python :

- interface.py : Se charge de l'affichage de l'interface, et de la communication avec le reste du programme.
- main.py : Se charge de charger les ontologies, de les parser, de générer un graphe, ainsi que de récupérer les données et les transformer afin d'y accéder facilement pour l'exécution. Ce fichier se chargera également de gérer l'exécution multi-tâches.

- comparison.py : Contient toutes les fonctions de comparaison utilisées par l'algorithme, ainsi qu'une fonction "maître" se chargeant de la pondération.

4. Main :

Dans le fichier main.py on retrouve deux classes, ainsi que des fonctions annexes. Les deux classes sont les suivantes :

F22_Expression : Contient une œuvre musicale d'un fichier de l'ontologie, avec les attributs suivants : graph, expression, title, note, composer, key, opus, genre.

```
def __init__(self, graph, expression):
    self.graph = graph

    self.expression = expression
    self.title = [AnalysableText(text) for text in self.getAllTitles(expression)]
    self.note = [AnalysableText(text) for text in self.getAllNotes(expression)]
    self.composer = [AnalysableText(text) for text in self.getAllComposer(expression)]
    self.key = [AnalysableText(text) for text in self.getAllKey(expression)]
    self.opus = [AnalysableText(text) for text in self.getAllOpus(expression)]
    self.genre = [AnalysableText(text) for text in self.getAllGenres(expression)]
```

Ces attributs sont des tableaux de type `AnalysableText`. On a également des fonctions écrites en SPARQL pour parcourir notre graphe. Voyons par exemple comment récupérer la liste des titres :

```
def getAllTitles(self, expression):
    req = """
            PREFIX mus: <http://data.doremus.org/ontology#>
            PREFIX ecrm: <http://erlangen-crm.org/current/>
            PREFIX efrbroo: <http://erlangen-crm.org/efrbroo/>
            PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

            SELECT ?expression ?title
            WHERE {
                ?expression ecrm:P102_has_title ?title .
            }
        """

    qres = self.graph.query(req, initBindings={"expression":
expression})
    result = []

    for row in qres:
        result.append(str(row.title))
```

```
    return result
```

Grâce à cette fonction, on exécute une requête SPARQL qui retourne la liste des titres de type `ecrm:P102_has_title` d'une expression passée en paramètre.

AnalysableText : Cette classe - assez simple - se charge de tokeniser un string, pour pouvoir effectuer des traitements supplémentaires. Un AnalysableText comportera donc deux attributs, le `String` simple, et une variable de type `Doc` utilisée par spaCy.

Afin d'accélérer le traitement des données, nous tirons parties de l'architecture multi-cœur des processeurs récents. Chaque expression est comparée sur son propre processus, ce qui permet de lancer plusieurs comparaisons d'expressions à la fois. Pour cela nous utilisons la librairie `Concurrent` intégrée par défaut dans Python, ainsi que la méthode `ProcessPoolExecutor()` qui lancera autant de processus que le processeur possède de cœurs. Sur nos machines de test, on constate une amélioration quasiment linéaire du temps d'exécution. Par exemple, sur une machine 24 coeurs, nous sommes passés de 67 minutes à environ 3 minutes (22 fois plus rapide).

```
futures = []
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for exp1 in result:
            future = executor.submit(threadCompare, exp1, result2, ...)
            futures.append(future)
        for future in futures:
            if future.exception() is not None:
                raise future.exception()
```

5. Comparison :

Ici chaque fonction retourne un entier entre 0 et 1, où 1 indique une égalité parfaite et 0 que les deux String n'ont rien à voir entre eux. On utilisera la fonction compare() qui prend en entrée deux `AnalysableText` ainsi que des paramètres facultatifs pour activer ou désactiver des algorithmes de comparaison.

Partie II. Évaluation Effectué

Pour comparer deux chaînes de caractères, nous avons donc utilisé plusieurs algorithmes en l'occurrence ici : Jaro, Jaccard, Identity, Monge Elkan, Ngram et Levenshtein. Pour cela, nous avons utilisé les algorithmes de la librairie Py_StringMatching ou nous les avons codés nous même.

Afin de laisser le choix à l'utilisateur, nous avons donné la possibilité de mettre une pondération au algorithme pour que certains donnent plus d'importance lors du calcul de la similarité que d'autres. De plus, nous nous disons que si l'identity renvoie vrai, alors nous n'aurons pas besoin de regarder les autres algorithmes, car ils feront de même.

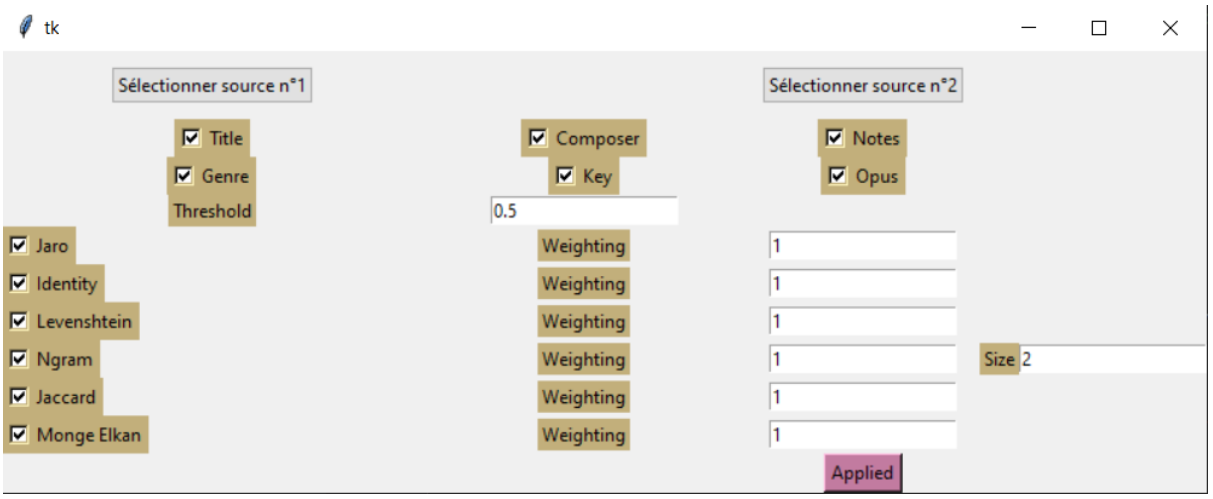
De plus, l'utilisateur à aussi le choix des algorithmes qui peut utiliser parmi la liste donnée auparavant, afin de pouvoir varier les valeurs retournées pour la comparaison.

Pour comparer deux entités F22_Self-Contained_Expression, nous utilisons 6 attributs qui nous paraissent le plus approprié, en l'occurrence : le titre, le compositeur (l'URI), les notes, l'opus, la game et le genre. Pour chacun des attributs, il est possible qu'ils contiennent une liste d'éléments. Pour cela nous prenons la valeur maximale retournée pour chaque algorithme lors de la comparaison avec un autre élément (du même type). Ensuite nous faisons la moyenne entre tous les algorithmes, puis entre tous les attributs pour comparer donc une entité F22_Self-Contained_Expression avec une autre.

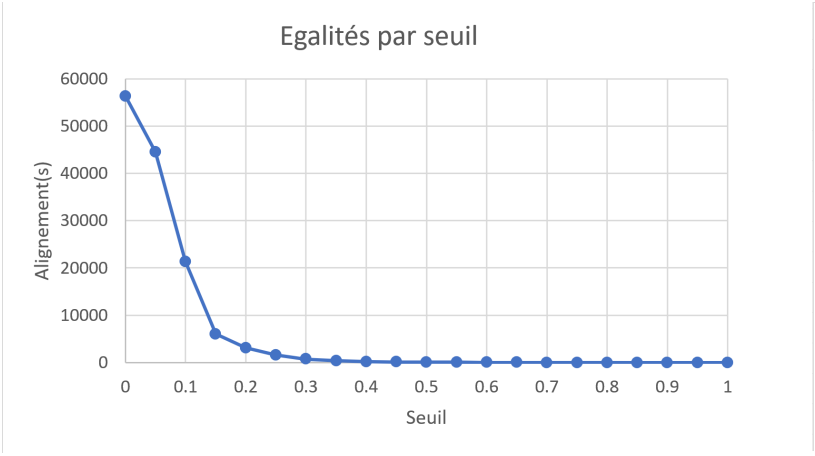
Tout cela nous retourne une valeur pour chaque paire d'entités qui sera ensuite comparée au seuil donné par l'utilisateur. Si cette valeur est supérieure ou égale à ce seuil alors nous considérons que les deux entités sont "similaires" et nous rajoutons donc une ligne <e1> owl:sameAs <e2> dans le fichier.

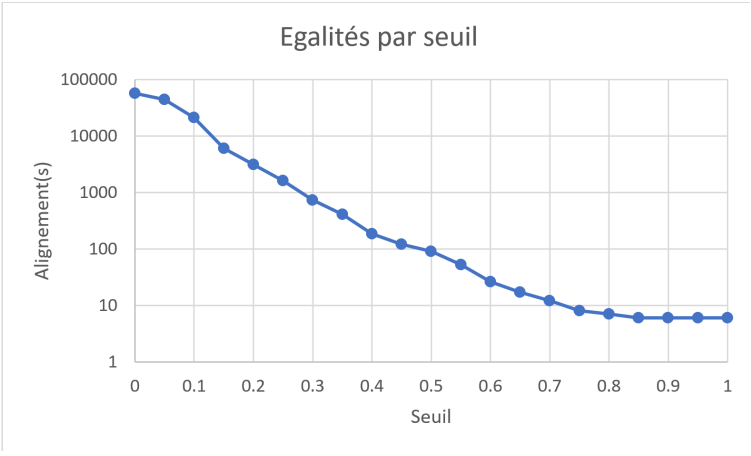
Partie III. Scénario de cas d'usage

Prenons par exemple l'exécution suivante:



Lorsque le programme se termine, nous obtenons 91 entités dites "similaires" selon notre méthode de comparaison. Pour obtenir cela, un fichier nommé "finalFile.ttl" contient toutes les entités qui sont similaires donc toutes les lignes au format suivant : <e1> owl:sameAs <e2>. Un autre fichier est créé à la fin, nommé "résultat.csv" où il contient toutes les valeurs d'égalité entre chaque entité seulement. Ce fichier nous permet de faire des statistiques sur les similarités entre les entités selon le seuil.





Ainsi en utilisant le fichier “résultat.csv”, on peut se rendre compte qu’avec les valeurs par défaut de l’interface, on obtient énormément d’entités similaires. Tout cela chute énormément jusqu’à 0,15 pour ensuite stagner jusqu’à la fin.

On peut voir sur la courbe logarithmique (courbe de droite) qu’à partir d’un seuil de 0,5 on obtient environ une 100 de “similarité” ce qui semble probable, étant donné que nous avons 238 œuvres à comparer.

Comparaison avec la vérité de terrain

Matrice Confusion (Seuil = 0,5)

	Positif	Négatif
Positif	12	225
Négatif	62	56 345

Accuracy : 99,5%

F1-précision (positif): 16,2%
F1-précision (négatif): 99,6%

F1-recall (positif): 5%
F1-recall (négatif): 99,9%

Conclusion

Ce programme nous a permis de découvrir le monde de l’alignement ontologique. Bien que basique dans son principe, nous comprenons mieux d’une part les fichiers d’ontologies, et d’autre part les outils mis à notre disposition pour les parcourir, comme SPARQL ou rdflib. Nous avons également pu utiliser des fonctions de comparaisons vu en cours.

Cependant, pour aller plus loin, nous aurions notamment pu utiliser les méthodes disponibles dans spaCy, par exemple pour extraire des mots clés, ou utiliser les fonctions de similarité ou synonymité fournies par défaut.

Enfin pour accélérer notre algorithme, il aurait pu être intéressant de segmenter nos données. On sait par exemple que les artistes d’une œuvre sont des IRIs, on peut donc faire une comparaison d’identité entre deux artistes, et puis uniquement comparer les œuvres entre elles dont on sait que les artistes sont les mêmes.