**TP2 – Construction d'un outil d'intégration de données**

# 1 Architecture

Nous avons structuré notre TP en 5 fichiers :

- utilities.py
- similarities.py
- graph.py
- parser.py
- main.py

## 1.1 utilities.py

Le fichier utilities.py est un fichier d'affichage. C'est-à-dire qu'il comporte une fonction qui permet de supprimer les éventuels textes déjà présents dans le terminal.

En effet, la fonction *cleanScreen* permet d'effacer le contenu du terminal selon le système d'exploitation. En fonction du système, on applique une commande : *cls* pour Windows; *clear* pour Unix/Linux.

## 1.2 parser.py

Ce fichier contient 6 fonctions qui permettent de parser les différents fichiers dont nous avons besoin.

– ***split_property(string)*** :

Cette fonction permet de transformer une chaine de caractères en une liste de chaines. Elle procède en testant si elle trouve dans la chaine des guillemets, des crochets, des points virgules ou simplement des caractères.

L'intérêt est de séparer la chaine donnée en sous-chaines en prenant les points virgules comme séparateur. Il peut y avoir des crochets contenant une autre chaine avec un point virgule à la fin mais qui ne sera pas pris en compte comme un point virgule de fin de sous-chaine. Et par conséquent ne fera pas changer de case la liste des sous-chaines.

Par exemple pour cette chaine : *"text1 ; text2 ; text3 [sous_text] ."*, on aura comme traitement : *["text1", "text2", "text3 [sous_text] ."].*

– ***formater(entity)*** :

La fonction prend une liste en entrée représentant une entité, et renvoie un dictionnaire qui contient les propriétés de cette entité qui seront formatées.

Plus précisément, à l'aide de la fonction split_property, il récupère et formate les propriétés de chaque entité donnée en paramètre sous forme d'une liste. Elles seront ensuite stockées dans un grand dictionnaire où les clés sont les noms des entités et les valeurs sont des sous-dictionnaires pour chaque propriété des entités : le nom de la propriété comme clé et la valeur associée.

Extrait d'un fichier :

```
<entite1>
    a    ex:prop1;
    ex:prop2
      [a   ex:prop21;
           ex:prop22 <entite2>
           ex:prop23 "val23"
      ];
    ex:prop3 "val3".
```

Entré de la fonction :

```
entite =
    ["<entite1>",
      "a",
      "ex:prop1",
      "ex:prop2",
      "[a ex:prop21;
       ex:prop22 <entite2>,
          ex:prop23 val23]",
      ";",
      "ex:prop3",
      "val3",
      "."
    ]
```

Qui retourne :

```
{
    "a": "ex:prop1",
    "ex:prop2" {
       "ex:prop22": "<entite2>",
        "ex:prop23": "val23"
    },
    "ex:prop3": "val3"
}
```

– ***parseur(file)*** :

Cette fonction permet de parser le fichier rdf fourni pour faire les comparaisons de nos données avec les vraies valeurs. Ainsi, cette fonction prend en entrer un fichier et retourne un dictionnaire comprenant la liste des entités et des propriétés associées comme tout à l'heure mais cette fois-ci pour un fichier rdf.

En effet, pour chaque ligne du fichier on fait voir, si il s'agit d'un préfixe, on le stocke dans le dictionnaire des préfixes. Si c'est une ligne vide, on passe au suivant. Sinon, on stocke la ligne dans l'entité courante sans formatage. Et si la ligne se termine par un point alors on la stocke dans le dictionnaire RDF_parse avec comme clé le nom de l'entité (souvent un lien URI) et comme valeur un dictionnaire de toutes ses propriétés.

Comme exemple, en prenant le fichier suivant à gauche et en appliquant *parseur(file)*, nous obtiendrons sur notre droite le dictionnaire des informations du fichier :

Entrée de la fonction :

```
<entite1>
      a    ex:prop11;
       ex:prop12  [a    ex:prop121;
                   ex:prop122  <entite2>
                   ex:prop123  "val123"
                  ] ;
       ex:prop13  "val13".

<entite2>
      a    ex:prop21  ;
      ex:prop22  "val22".
```

Retourne le dictionnaire des entités suivant:

```
{
<entite1> :
      {
            "a": "ex:prop11",
            "ex:prop12" {
                  "ex:prop122": "<entite2>",
                  "ex:prop123": "val23"
             },
             "ex:prop13": "val13"
       },
<entite2> :
      {
             "a": "ex:prop21",
             "ex:prop22": "val22"
       }
}
```

– ***spo_formater(parsed_graph)*** :

Elle prend en entrer un dictionnaire parsed_graph où chaque clé est une entité et chaque valeur est un dictionnaire de ses propriétés. Elle retournera une liste de triplet qui pour chaque propriété créera un nouveau triplet avec son sujet associé.

Pour plus de détails, elle récupère pour chaque clé valeur de l'entrée, les prédicats et les objets dans deux listes. Puis, elle crée une liste de clé qui contient la clé courante autant de fois qu'elle a de propriété. Et crée chaque triplet en prenant dans les listes key, prédicat, objet le premier élément, puis le deuxième etc...

Par exemple, prenons le graph sur la gauche et en appliquant *spo_formater(graph)* on obtient ce qu'il y a à droite :

Entrée de la fonction :

```
{
      <entite1> :
      {
            "a": "ex:prop11",
            "ex:prop12" {
                  "ex:prop122":  "<entite2>",
                  "ex:prop123": "val23"
             },
            "ex:prop13": "val13"
       }
}
```

Retourne la liste des triplets suivante:

```
[
   (<entite1>, "a", "ex:prop11"),
   (<entite1>, "ex:prop12",
       ["ex:prop122": "<entite2>",
       "ex:prop123": "val123"]),
   (<entite1>, "ex:prop13","val13")
]
```

– ***ground_truth_parser()*** :

Elle permet de parser le fichier "refDHT.rdf" pour en extraire les entités qui appartiennent au fichier "source" et au fichier "target" et de les convertir en triplets de la forme (s, p, o) où s est une entité de "source", o, une entité de "target" et p, l'ontologie "owl:sameAs" reliant les deux.

Pour cela, on parcourt le fichier redfDHT et on ajoute à chaque liste d'entité (entite1 et entite2) l'URI correspondant et on forme le triplet pour chaque entité des deux listes qu'on ajoute dans true_triplet.

Par exemple, prenons un extrait de rdfDHT.rdf à gauche et en appliquant *ground_truth_parser()* on obtient ce qu'il y a à droite :

Entrée de la fonction :

```
<map>
  <Cell>
    <entity1 rdf:res="http://test1"/>
    <entity2 rdf:res="http://test2"/>
    <measure rdf:dttype="xsd:float">1.0</measure>
    <relation>=</relation>
  </Cell>
</map>
```

Retourne la liste de triplet suivante:

```
[('http://test1', 'owl:sameAs', 'http://test2')]
```

– ***cutter(triplet_list)*** :

Et pour finir, cette fonction permet de nettoyer une liste de triplets en supprimant les caractères < et > inutiles qui peuvent se trouver dans les éléments s, p et o.

Pour cela on regarde simplement si à la fin de la chaine on voit qu'il y a un >. Ainsi, si c'est le cas, on supprime le premier et le dernier élément.

Pour donner un rapide exemple en prenant la chaine suivante : "<http://data.doremus.org/text>", on obtiendra alors la chaine "http://data.doremus.org/text".

## 1.3 similarities.py

Ce fichier contient toutes les fonctions de similarité que l'on utilise dans le code. Nous utilisons donc les suivantes :

> levenshtein_similarity > NGrams_similarity > jaccard_similarity

> jaro_similarity (importé) > synonymy_similarity > monge_elkan_symmetric

On a également une fonction de similarité qui correspond à la moyenne de toutes mais que l'on a pas mis dans le graph sachant que la génération de tous les triplets et le calcul des f-measures était trop long (plus de 1H) pour refaire les graphs avec.

## 1.4 main.py

Ce fichier est structuré en 7 parties permet de former les triplets ainsi que calculer les f-measures pour chaque seuil et chaque fonction de similarités :

### 1.4.1 Données :

Pour commencer, la première partie correspond aux données utiles comme la liste de toutes les fonctions de similarités utilisées et possibles. De plus, il y a la liste des seuils possibles qui vont de 0.55 à 1 avec un pas de 0.05.

### 1.4.2 Tests des similarités

Ensuite, il y a une partie sur le test des fonctions de similarités pour voir si elles marchent bien ou pas (on supprimera surement ou au moins commenté).

#### 1.4.3 Choix de l'utilisateur

On continue par la partie où l'utilisateur choisit la fonction de similarité à tester et le seuil de similarité qu'il veut utiliser. On lui propose donc la liste des fonctions et on lui indique de ne rentrer qu'un seuil entre 0.5 et 1.

#### 1.4.4 Parseur

Une fois le choix de l'utilisateur fait, on parse les deux fichiers source.ttl et target.ttl à l'aide des fonctions que l'on a expliqués plus haut. On les stockera dans $g1$ et $g2$. Puis, on transformera les dictionnaires en listes de triplet dans $l$iste_g1 et $l$iste_g2.

De plus, on récupère également les valeurs de vérité du fichier rdfDHT.rdf que l'on va stocker dans *true_triplets* sous forme également de triplet pour que les différents stockages aient la même structure.

#### 1.4.5 Comparaison des fichiers

Une fois qu'on a parser les différents fichiers, on continue par la comparaison des deux fichiers. On commence par parcourir la liste du fichier source *liste_g1*, on a donc *sujet1*, *predicat1* et *objet1*.

Ensuite on parcourt la deuxième liste qui correspond au fichier target *liste_g2*, on aura donc *sujet2*, *predicat2* et *objet2*. Nous vérifions ensuite si l'objet est un dictionnaire on le convertit en chaine de caractères.

Puis nous utilisons la fonction de similarité choisis et le seuil pour calculer la similarité entre les prédicats et les objets. Pour les prédicats on utilise 0.9 comme seuil car on veut que le prédicat soit presque identique. Mais pour l'objet on utilise le seuil choisi pour comparer.

Si jamais la similarité est plus grande que le seuil, les éléments sont proches et donc on ajoute à la liste des triplets le triplet courant.

#### 1.4.6 Ecriture des triplets dans results/triplets.txt

Une fois la comparaison des fichiers faite on ouvre le fichier *results/triplets.txt*.

On va donc remplir ce fichier avec les triplets contenus dans la variable *triplets* qui affichent donc l'URI du premier élément présent dans le fichier source.ttl, le prédicats os:SameAs, et l'URI du second élément présent dans le fichier target.ttl.

#### 1.4.7 Calcul de la précision, le rappel et la f-measure

Il nous reste à calculer la précision, le rappel et la f-measure, et pour cela nous devons d'abord calculer le nombre de triplets trouvés ainsi que le nombre de triplets qui sont vrais donc présents dans le fichier rdf.

Puis nous pouvons calculer la précision telle que :

- precision_valeur = true_matches_found / nbMatchesFound;
- rappel : recall_valeur = true_matches_found / nbTrueMatches;
- f-measure : f_measure_valeur = (2 * precision_valeur * recall_valeur) / (precision_valeur + recall_valeur).

#### 1.4.8 Ecriture de f-messure dans le fichier csv

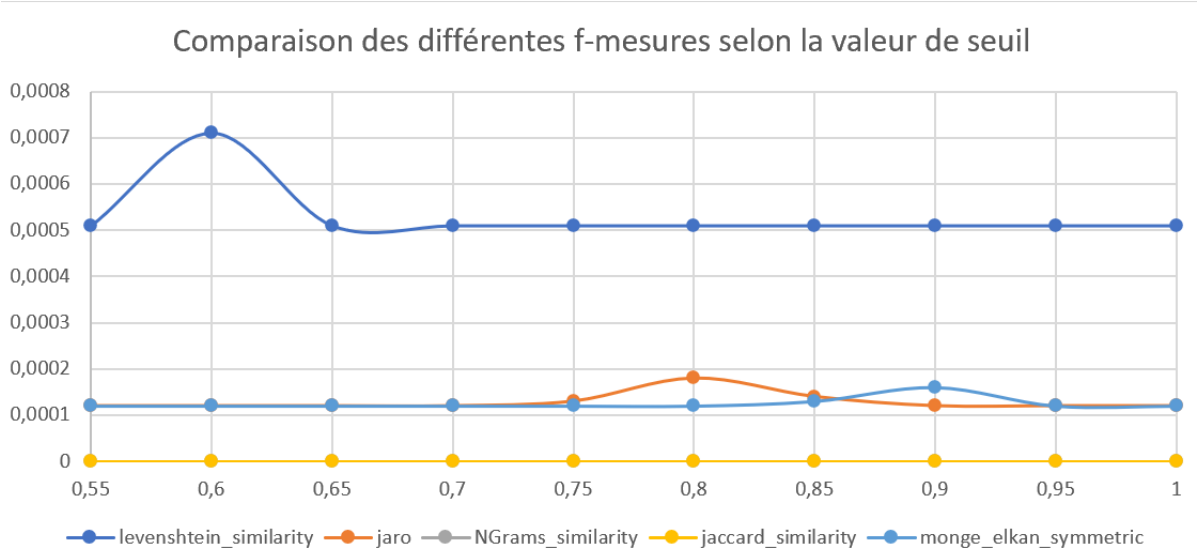
On ouvre alors ensuite un fichier csv nommé *results/measures.csv*. Puis nous mettons le nom de la focntion de similarité choisis ainsi que le seuil choisis. Et pour finir, nous ajoutons la precision, le rappel et la f-mesure calculée avant.

### 1.5 calcul-f-mesure.py

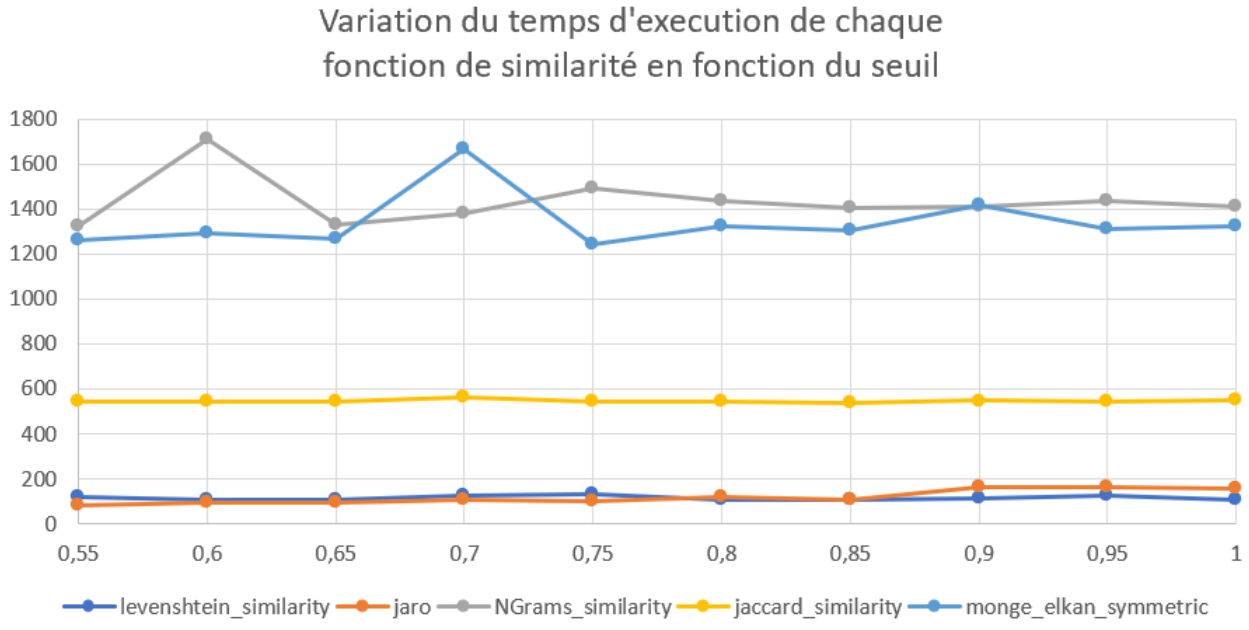
Il nous reste un fichier à décrire qui correspond à la même chose que *main.py* mais au lieu de faire choisir par l'utilisateur la fonction de similarité et le seuil, ce fichier va calculer pour chaque fonction de similarité et chaque seuil.

## 2 Evaluation

Nous avons voulu comparer les différentes fonctions pour connaître leurs efficacités respectives. Ces générations ont été faites dans les fichiers *f-mesureFull.csv*. Le graphe ci-dessous montre les f-mesures obtenues en fonction des valeurs de seuil.




Notre problème étant que cette génération nous a pris 18h, nous n'avons pas pu en refaire une nouvelle après amélioration de notre système de comparaison, où la f-mesure moyenne atteignait les 0.002. Nous avons également généré un graphe pour comparer les temps d'exécution des différentes fonctions dans le fichier *timeFull.csv*.



# 3 Scénario de cas d'usage

Montrons maintenant, un cas d'usage. Lorsqu'on lance l'application à l'aide de la commande *python3 main.py* nous obtenons l'affichage suivant :

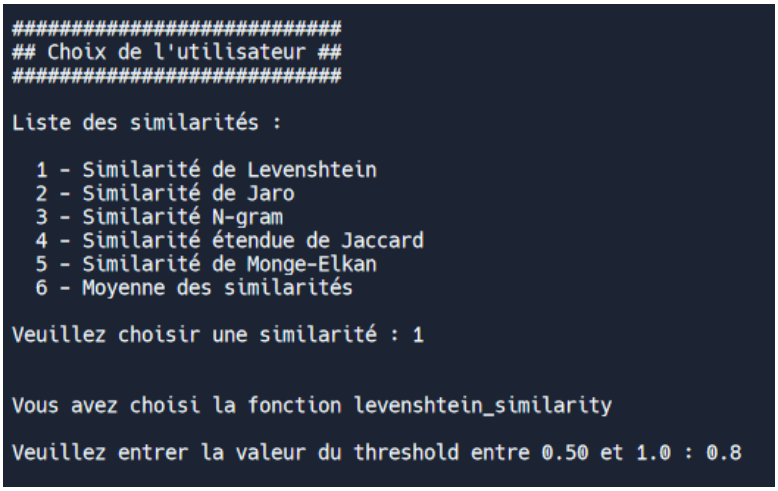



Figure 1: Choix de l'utilisateur

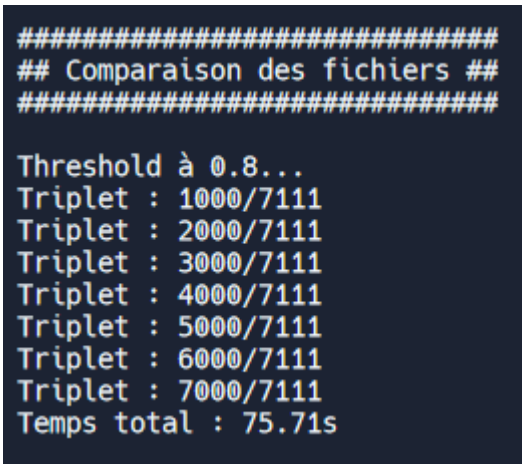



Figure 2: Comparaison des fichiers

Vous voyons alors qu'on choisis la fonction de similarité de Levenshtein, et un seuil de 0.8 sur la figure 1 à la page 6.

Et sur la figure 2 nous pouvons voir l'affichage lors de la comparaison des deux fichiers source et target. Nous choisisons d'afficher seulement le nombre de triplet par millier pour ne pas tout afficher.


Figure 3: Extrait du fichier *results/triplets.txt* obtenus

Nous pouvons donc voir sur la figure 3 à la page 6 un extrait du fichier *results/triplets.txt* obtenus après la compraison des deux fichiers.

Après la génération du fichier avec tous les triplets, on doit calculer la precision, le rappel et la f-measure qu'on stockera dans le fichier *results/measures.csv* ci-dessous.

| Nom fonction | threshold | precision | rappel | f-mesure |
|---|---|---|---|---|
| levenshtein_similarity | 0.8 | 0.00065 | 1.0 | 0.00129 |

Figure 4: Mesures calculés