Table des matières

1. [Data Pipeline](#data-pipeline)

2. [Ad-hoc](#ad-hoc)

3. [Pour aller plus loin](# Pour aller plus loin)

3. [Sql Requests](#sql-requests)

1. Partie 1 : Data Pipeline

L'objectif de ce pipeline est de produire en sortie un fichier JSON qui représente un graphe de liaison entre les différents médicaments et leurs mentions respectives dans les différentes publications PubMed, les différentes publications scientifiques et enfin les journaux avec la date associée à chacune de ces mentions.

1.1 Inputs Data

File	Count	Description
data/drugs.csv	7	un fichier csv contenant les données des drugs avec deux colonnes : atccode => id de drug, drug => nom de drug
data/clinical_trials.csv	8	un fichier csv contenant les données des essais cliniques avec 4 colonnes : id => id de trial, scientific_title => le titre scientifique de trial, data: date de trial, journal: nom de journal de publication de trial
data/pubmed.csv	7	un fichier csv contenant les données des publications avec 4 colonnes : id => id de publication, title => le titre de la publication, data: date de la publication, journal: nom de journal de publication
data/pubmed.json	5	un fichier json contenantles données des publications avec 4 colonnes : id => id de publication, title => le titre de la publication, data: date de la publication, journal: nom de journal de publication

1.2 Pipeline :

Workflow	Etapes
Tous	Reading Inputs → Data Cleaning → Processing → Loading Result

1. *Reading Inputs :* consiste à lire la config et les données d'entrée en utilisant un connecteur; Dans notre cas, on a un seul type de connecteur : un connecteur local.

   A. *Lire la config :* utiliser la fonction load_conf_file pour le fichier de config en json.
   B. *Lire les données drugs :* en utilisant la fonction reader qui permet de charger des données de différents formats (JSON/CSV).

C. *Lire les données de liens (pubmed, clinical_trials):* en utilisant la fonction concat_data qui permet de charger les données de différents formats et de les concaténer dans un seul dataframe. => Le fichier pubmed.json est endommagé. J'ai rajouté un détecteur d'erreur et un logging qui décrit le nom du fichier et l'erreur correspondante.

2. **Data Cleaning :** consiste à :

A. *clean_date_column :* permet de normaliser les dates afin d'avoir le même format : '%d-%m-%Y'

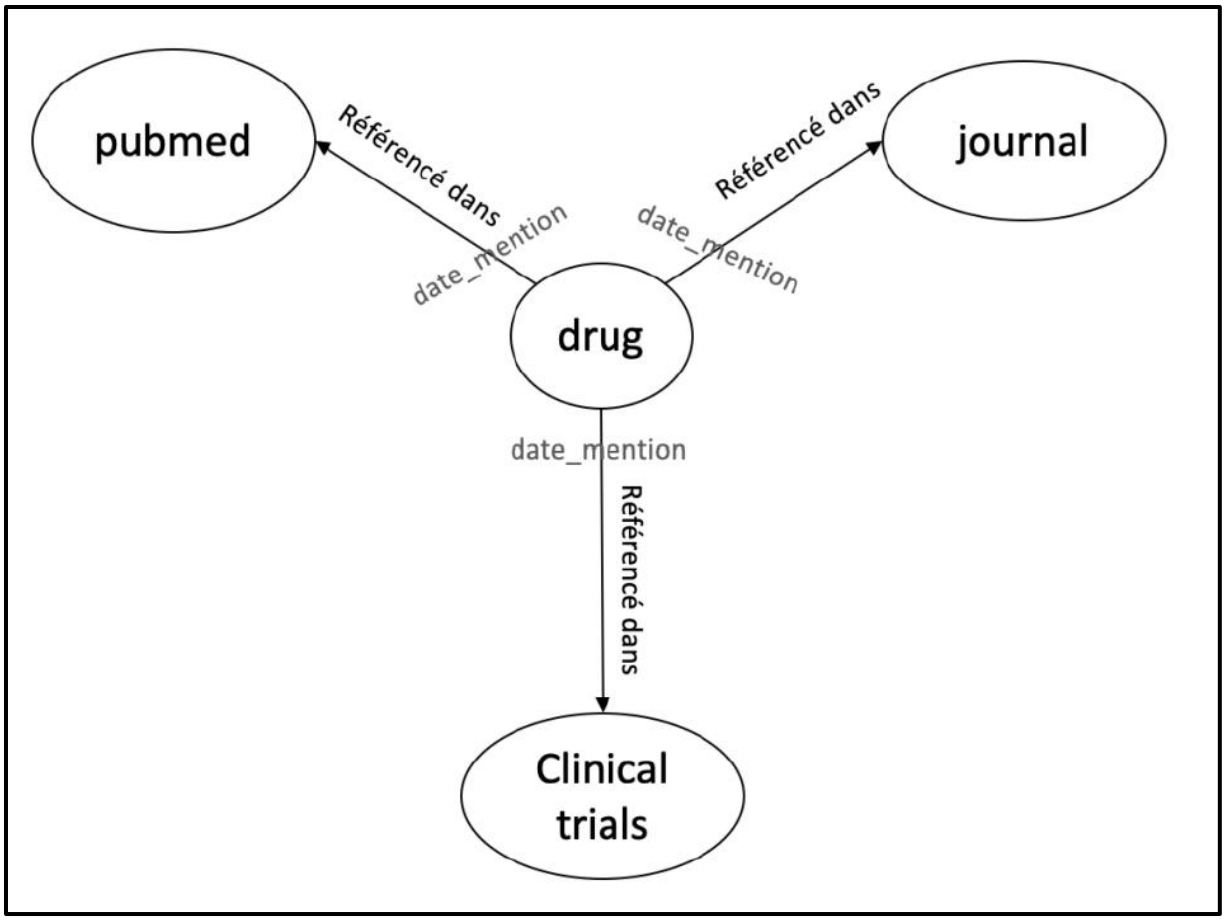
B. *clean_words_column :* permet de nettoyer les colonnes contenant des mots, d'enlever les caractères non ascii, les espaces et de lower case ces strings.

C. *deal_with_nan :* permet de supprimer les nulls.

Le data cleaning peut être complété et améliorée si nous avons plus d'information sur les inputs et les différentes colonnes, par exemple la colonne atccode dans le dataframe des drugs a un id, à la ligne 6, qui ressemble aux autres id : 6302001 …

3. Data Processing : Cette étape utilise la fonction compute_drug_links qui permet de calculer les liens entre les différents médicaments et leurs mentions dans les PubMed, les publications scientifiques et les journaux. .

Afin de représenter le graph attendu suivant :



Le calcul se déroule comme suit :

1. Définir une liste vide qui va stocker tous les drugs id avec leurs liens
2. Faire une Boucle sur le l'index de dataframe des drugs
3. Définir un dictionnaire vide qui va stocker un drug avec ses liens à chaque tour de boucle
4. Recupérer l'id et le nom de drug correspond à l'index d'une boucle donnée.
5. Mettre à jour le dictionnaire des links avec deux items : drug_id et links.
5. Faire une boucle sur les lignes de dataframe données mention
6. Si la colonne "title" contient le drug de la boucle => on ajoute dictionnaire à la clé links qui comprend les items suivant :
    'link_type' : peut prendre deux valeurs : « clinical_trial », « pubmed »
    'journal' : nom de journal qui mentionne le drug
    'date' : date à laquelle le drug a été mentionnée

Ce calcul permet de modéliser le graph des liens qui prend le schéma suivant :

```
[{"drug_id": drug_id_value,
        "links":[{
            "link_type": link_type_1, "journal_name" : journal_name_1,
            "date_mention": date_mention_1},  ....],
```

```
        }]
```

4. Loading Result: permet d'écrire l'output de la pipeline dans un fichier json en utilisant le laoding. Dans notre cas, on a un seul type de loading : loading_local.

5. Tests : nous pouvons rajouter plus de tests avec plus d'assert. Nous pouvons exécuter les tests dans un module avec : pytest test_mod.py.

- **Run Pipeline :**

Ce pipeline peut être lancée en utilisant :

```
main.py
```

Partie 2 : Ad-hoc :

Cette feature permet d'extraire depuis le json produit par la data pipeline le nom du journale qui mentionne le plus de médicaments différents. Le calcul de cette feature ….

- **Run feature :**

Cette pipeline peut être lancée en utilisant :

```
main_feature.py
```

Partie 3 : Pour aller plus loin :

Afin de gérer de grosses volumétries de données, nous avons plusieurs options :

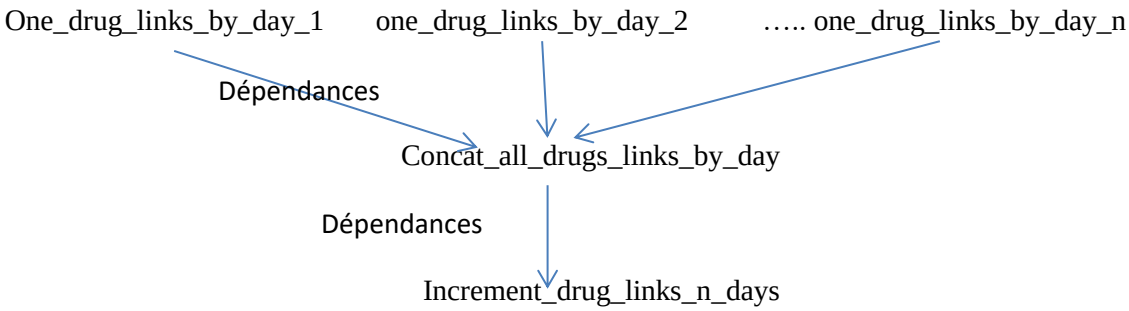
Pipeline	Moyen d'optimisation	Comment ?
Data reading & loading	Formats du fichier	Le format csv (et json) prend beaucoup de temps en lecture et en écriture de gros volumes de données et ne souvient pas des types de données sans indication explicite. En revanche, il est par l'homme. Pour améliorer le format de notre pipeline : • On peut utiliser les mêmes formats avec compression ce qui peut diviser le temps de lire et décrire en deux. • Il existe d'autres formats alternatives pour gérer des grands datasets : pickle, Feather, Parquet et HDFS….
	chunking	Diviser les données en morceaux : pandas read_csv, read_json etc. ont un parameter chunksize crée un objet itérateur qui peut être utilisé pour parcourir les différents chunks.
Data cleaning	Type de données	Lorsqu'on charge les données dans un dataframe pandas, les types de données par défaut attribués à chaque colonne ne sont pas efficaces en mémoire. Par exemple : int64 peut être converti en int8 ou int16 ou int32 en fonction de la valeur max et min de la colonne.

		Si on converti le data type, nous pouvons économiser beaucoup la mémoire.
Data processing/ modeling	Les alternatives de pandas	Il existe des librairies python pour gérer des ensembles de données en mémoire plus efficaces que Pandas : Dask(traitement parallel), Ray, Modin et Vaex … qui utilisent des fonctionnalités de multi-threading, et multi-traitement. Ils ne chargent pas les données directement en mémoire mais les divisent d'abord en partitions.

	Lazy Evaluation	Comme en spark, l'évaluation paresseuse évalue l'expression python lorsque le résultat est nécessaire. Ce qui peut améliorer l'efficacité de code et économiser beaucoup de ressources.
		
		A. Range() /zip() : Quelle que soit la taille de range, l'objet a toujours la même taille. Cela est dû au fait que le range ne stocke que les valeurs de début, de stop et de step et calcule chaque élément lorsque cela est nécessaire.
		
		B. Iterator > generator : l'iterator est un objet dont la classe a deux méthodes : __next__ et __iter__. Chaque fois qu'on appelle next() à l'objet itérateur, on obtient l'élément suivant dans la séquence jusqu'à ce que l'objet itérateur soit épuisé et déclenche StopIteration. Le générateur est une fonction qui renvoie un itérateur. Cela ressemble à une fonction normale sauf qu'elle utilise yield au lieu de return.
		
		Lorsque l'instruction yield est exécutée, le programme suspend l'exécution de la fonction en cours et renvoie la valeur renvoyée à l'appelant => La valeur est calculée et renvoyée lorsque l'appelant est nécessaire et la valeur suivante sera toujours silencieuse et ne fera rien dans le programme.
		C. Lambda : permet d'effectuer le calcul à un seul élément dans chaque boucle.
		
		Nous pouvons améliorer notre code de data modeling en utilisant un generator :
		
		def lazy_loading(items):
		for i in items:
		yield i ** 2
		
		for ind in lazy_loading (drugs.index) :
		….

	Luigi	Nous pouvons améliorer notre pipeline en utilisant les tâches luigi qui sont des blocs de construction à partir desquels nous créerons notre pipeline. Une task luigi est l'endroit où l'exécution de notre pipeline et la définition des dépendances d'entrée et de sortie ont lieu. Luigi permet de gérer des jobs s'étalant sur plusieurs semaines. C'est de python, nous savons déjà comment le coder Sa structure permet de récupérer les tâches ayant échoué sans ré exécuter l'ensemble de pipeline. Il a une interface graphique indiquant l'état de pipeline et des tâches. Le workflow luigi de notre pipeline va définir des dépendances entre ces trois grandes tâches: 1. Calculer les liens par jour par drug.

<table>
<tr>
<td>Data Pipeline</td>
<td></td>
<td>
2. Concaténer les liens de plusieurs drugs
3. Incrémenter les liens des drugs sur plusieurs jours (mois, année ….)



<u>Task luigi :</u>
import luigi
class TaskLuigi(luigi.Task):

def requires(self): => définit les dépendances
return drug_by_day()

def output(self): => définit l’output/target
return luigi.LocalTarget(‘links_outputs.json’)

def run(self): => contient le code qu’on veut executer à notre stade de pipeline
with self.output().open("w") as outfile:
outfile.write("drugs links!")
Nous pouvons rajouter à notre pipeline des paramètres de configurations pour personnaliser les drugs et le nombre de jours pour lesquels calculer les liens….

class GlobalParams(luigi.Config):
Drugs = luigi.ListParameter()
NumberDays = luigi.IntParameter(default=500)
</td>
</tr>
<tr>
<td></td>
<td>Spark</td>
<td>Spark permet d’exécuter et de paralléliser les jobs sur d’énormes ensembles de données. Il offre :
Une mise en cache puissante
Une bonne constance du disque.
Un traitement rapide.
…</td>
</tr>
</table>

Partie 4 : Sql requests :

1. **1ere requête :** Calculer le montant total des ventes jour par jour :

sql/sales_total_amount_day_by_day

2. **2eme requête :** Calculer le montant des ventes des meubles et deco par client.

sql/meubles_deco_sales.sql