Ce document permet de compléter le code en termes de choix d'implémentation et de répondre aux questions qui ne nécessitent pas d'implémentation

Choix du framework

Le projet étant libre, les différents options envisagées ont été :

- pure python : peu de dépendance, possibilité de customiser le code en profondeur mais ne tire pas partie des librairies matures sur l'analyse et le traitement de la données

- python + librairies type pandas: permet de profiter d'un arsenal de fonctionnalités adaptés au traitement de la donnée

- pyspark : permet d'accéder à un grand nombre de fonctionnalités qu'on retrouve sur des librairies standards tel que pandas avec l'avantage d'une scalabilité accrue. Dans l'optique de gérer une volumétrie plus importante sans avoir à réadapter le code de manière importante, le choix s'est porté sur pyspark.

Dans un contexte projet établie, le choix d'un framework dépendra d'autres facteurs, les besoins et contraintes à long terme du projet, l'expertise de l'équipe et/ou le temps disponible pour monter en compétence si nécessaire etc.

Au sein de pyspark, plusieurs choix possible pour l'implémentation. Par rapport au problème exposé, les plans d'optimisation via dataframe semblent adaptés. Dans le cas de industrialisation d'un code de data scientist faisant un usage intensif de Pandas comme base d'outil analytic, il aurait peut être été intéressant d'utiliser Pandas avec Arrow.

Considérations

Modularité

Diviser les jobs pour permettre la réutilisation dans d'autres pipeline

Code et documentation

Idempotence des fonctions
Limité la duplication de code
Documentation et coding style qui suivent les conventions standards sauf consensus au sein de l'équipe.
Documentation concise et claire, l'objectif permettre à un développeur de comprendre le code de manière autonome, en limitant au mieux les sources d'ambiguïté et d'incohérence dans la nomenclature des variables par exemple.

Choix d'implémentation

Json invalide: Le fichier json fourni en input possède un « trailling coma » qui ne respecte pas le standard json.
Quelques possibilités pour gérer ce point:

- Retourner un log d'erreur indiquant que le format json n'est pas respecté, on considère ici que le problème doit être résolue lors de l'extraction des données (idéal mais pas toujours possible).

- Utiliser un « workaround », lecture comme un « yaml » ou « jsoncomment » par exemple, le nettoyer pour respecter le format json

- Utiliser une regex pour nettoyer et standardiser le fichier au format json

Pour la résolution « ad-hoc » du problème, le choix s'est porté sur la lecture du json au format yaml. Cependant, avec plus de temps et pour gérer un plus grand nombre d'erreurs possibles dans un contexte ou la qualité de la donnée est très variable, une regex adaptée peut être plus intéressante.

Choix du format du graphe final

Le fichier json final est composé des structures suivantes:

Noeud drug


```
{
	"drug": "TETRACYCLINE",
	"atccode": "S03AA"
}
```


Relation drug -> id (pubmed, trial ou journal)

Noeud pubmed


```
{
	"id": "4",
	"source_type": "pubmed",
	"title": "Tetracycline Resistance Patterns of Lactobacillus buchneri Group Strains.",
	"type": "node"
}
```


Noeud clinical_trial


```
{
	"source_type": "clinical trial",
	"title": "Glucagon Infusion in T1D Patients With Recurrent Severe Hypoglycemia: Effects on Counter-Regulatory Responses",
	"type": "node"
}
```


Noeud journal


```
{
	"journal": "American journal of veterinary research",
	"source_type": "journal",
	"title": "American journal of veterinary research",
	"type": "node"
}
```


Relation drug -> id (pubmed, trial ou journal)


```
{
	"drug": "TETRACYCLINE",
	"id": "5",
	"date": "02/01/2020",
	"type": "relation"
}
```

Ce format a été choisit pour limiter la duplication de données et permettre une indépendance des noeuds et relation dans le json final. Cela permet de parcourir le graphe selon les nécessités des requêtes sans être contraint par des champs imbriqués par exemple.

Améliorations

Par manque de temps, un certain nombre de points n'ont pas été implémentés mais me paraissent, pour la plupart, indispensable.

Config file pour les path

Les chemin de fichiers sont actuellement codés en dur dans le job run. La première chose à faire est d'externaliser dans un fichier de conf ces chemins et les passer en paramètre au job.

Modulariser les jobs

Actuellement le pipeline consiste en un job « run ». Les fonctions utilisées sont dans des modules différents « extract », « transform », « load » mais pour utiliser ces différentes parties dans un DAG, il est nécessaire de créer un job pour l'extract un job pour transform et un pour le load. On peut sauvegarder les résultats intermédiaires pour compartimenter les étapes.

Algorithmes

Indexer les journaux et les drugs

L'id utilisée pour les journaux est actuellement le nom du journal. Une version plus optimisée serait d'indexer les journaux pour prendre moins de place en mémoire. On peut aussi créer un index pour les « drugs » afin de réduire le poids du json.

Cleaning des données

Les contenus nécessite d'être nettoyés et stockés sous des formats normalisés.

En particulier les dates sont dans des formats hétérogènes qu'il est nécessaire de normaliser pour pour effectuer des recherches de manière efficace. Certains sample de données ne permettent pas de déterminer de manières certaines le format des dates (fichier pubmed.csv) avec un format qui peut être dd/mm/yyyy ou mm/dd/yyyy.
Au vue des données fournies, on peut estimer que les dates sont dans un format européen mais ce serait à valider avec les experts métiers ou un sample plus large.

Certains lignes n'ont pas d'id ce qui peut poser problème par exemple, selon le type de base utilisées pour le stockage. Il pourrait être judicieux de compléter les id (avec un uuid par exemple) ou de créer un master id interne.

Gestion d'un nombre de fichiers sources indéterminés

Actuellement, le code ne gère qu'un nombre de fichier limité. On imagine que les répertoires contenants les fichiers sources peuvent contenir de nombreux fichiers csv et json. Il faudrait donc gérer la lecture du répertoire et charger tous les fichiers au sein d'un des Dataframe pubmed, trial, drugs. On peut imaginer une fonction particulière pour faire ca et modifier le code pour séparer le merge des données par catégorie (les csv et json pubmed ensemble) en amont de la phase transform actuelle.

Améliorer la documentation

La documentation peut être améliorée.

Développement continu, automation et monitoring

Tests unitaires

Implémenter des tests pour s'assurer de la non regression de code lors des futures évolutions.

Exemple
Regex = extrait multiple drugs in title
Regex = extract drug with different symbol around
Contrôle des résultats lors des différentes transformations des données
Contrôle des résultats lors du chargement des données via les différents formats
Contrôle du nombre du fichier exporté lors de la génération du graph en json

Logging anomalies

Une chaine de traitement de données doit permettre d'identifier et de traiter les anomalies qui se sont produites lors de son execution. Outre les logs de Spark, il peut être nécessaire de produire des logs propres à l'application développée pour permettre de tracer les erreurs mais aussi les résultats à la fin de chaque étape de la chaine de traitement.

Pour une application Spark, on peut utiliser le module Log4j pour enrichir les logs de l'application et gérer la sauvegarde de ces derniers (path, niveau de priorité, message à générer etc).

Containerisation

Déployer le pipeline au sein d'un Docker permettrait de faciliter le déploiement de manière agnostique en terme d'environnement. Par exemple, un code développé en local par différents développeurs et exécuté dans un cluster on premise ou dans le cloud sur AWS, GCP ou Azure ne subirait pas les problématiques liés à des environnements différents.

Optimisations

Cache intermediate files

On peut sauvegarder les Dataframes intermédiaires pour rendre le pipeline plus modulable. Il peut être intéressant de cacher en mémoire (persist) les Dataframe intermédiaire qui réutilisent des Dataframe calculés précédemment.

Streaming

En l'état le projet est orienté « batch » et n'est pas optimisé pour le streaming.
Développer en pyspark permet d'avoir accès aux api comme Spark streaming permettant le micro batch qui peut, si nécessaire gérer l'intégration de nouvelles données au fils de l'eau.

Pour aller plus loin

Quels sont les éléments à considérer pour faire évoluer votre code afin qu'il puisse gérer de grosses volumétries de donn.es (fichiers de plusieurs To ou millions de fichiers par exemple) ?
Pourriez-vous décrire les modifications qu'il faudrait apporter, s'il y en a, pour prendre en considération de telles volumétries ?

Le code actuel écrit en pyspark permet en théorie de monter en volumétrie. Le pipeline n'ayant été testé qu'en local mode, il est probable que des ajustements soient à faire.

Cependant l'implémentation actuelle convertie la liste des « drugs » en liste python donc en un objet non parallélisé qui peut potentiellement poser problème si la liste de « drug » est importante (on pourrait estimer un ordre de grandeur:
longueur moyenne d'un string * nombre d'entrées) et comparer à la mémoire du driver Spark pour avoir un ordre de grandeur.

Pour gérer une volumétrie plus importante, on pourrait indexer le vocabulaire (mots composant les « titles » et « scientific titles » via des integers, faire de même pour les médicaments et faire une recherche soit via une liste d'integers plus légère en mémoire que les tokens précédemment utilisés ou bien exploser la table et faire la jointure via ces id integer.

Une autre solution pourrait être, par exemple de nettoyer les titres des mots communs avec une liste de stop words voir, des méthodes plus avancées (identification d'entité nommée), puis exploser la liste de mots restant comme dans la solution actuelle, puis faire la jointure avec le dataframe « drugs » pour rester dans un contexte parallélisé. A noter que la reconnaissance d'entité nommée est dépendante de l'entrainement d'un modèle et ne serait pas fiable à 100%.

De plus, certaines parties du code ne sont pas écrite en pyspark, la recherche du journal qui cite le plus de « drugs » par exemple. Etant donné que la taille du json peut devenir extrêmement large , il pourrait être intéressant de le charger de manière partielle pour effectuer des recherches sur la partie qui nous intéresse. On pourrait aussi stocker le graphe dans un autre format ou dans une base de donnée graph pour optimiser les requêtes. On pourrait aussi rester dans un contexte Spark avec Graphx qui permettrait de paralléliser les données.

Partie II

2. Première partie du test

```
SELECT date, SUM(prod_price*prod_qty) as ventes
FROM TRANSACTION
GROUP BY date
ORDER BY date ASC
```

3. Seconde partie du test

```
SELECT client_id,
            SUM(IF(product_type="MEUBLE", vente, 0)) as vente_meuble ,
            SUM(IF(product_type="DECO", vente, 0)) as vente_deco
FROM
   (SELECT client_id, vente, product_type
    FROM
       (SELECT client_id, prod_id, vente
        FROM
          (SELECT client_id, date, prod_id, SUM(prod_price*prod_qty) as vente
           FROM TRANSACTION
           WHERE TRANSACTION.date >= '01/01/2020' and TRANSACTION.date <'01/01/2021'
           GROUP BY client_id, date, prod_id)) AS VENTE_PRODUCT

           INNER JOIN

           (SELECT product_id, product_type
            FROM PRODUCT_NOMENCLATURE) AS PRODUCT

           ON VENTE_PRODUCT.prod_id=PRODUCT.product_id)
GROUP BY client_id
```