Documentation

Contexte

L'ensemble du code est disponible dans le repository suivant : https://github.com/seballiot/legalstart_weather_data.

J'ai utilisé du code Python, d'abord exécuté en local dans un venv puis dans un conteneur basé sur une image Docker.
Pour la partie SQL, j'ai utilisé un environnement BigQuery pour l'exécution des requêtes.

Choix de l'API WeatherAPI
J'ai d'abord utilisé l'API OpenWeatherMap comme mentionné dans l'énoncé, mais je me suis heurté à un problème d'accès aux données historiques.
En effet il semblerait qu'avec un abonnement « free » (compte que j'ai utilisé) on a accès à la partie *Current Weather* mais pour l'accès à l'historique (endpoint https://history.openweathermap.org/data/2.5/history) il faudrait un abonnement différent.

J'ai pris la liberté de m'adapter et d'utiliser une autre API gratuite, Weatherapi, qui est similaire et propose l'accès aux mêmes types de données que l'API OpenWeatherMap.
Si jamais j'ai loupé quelque chose lors de l'utilisation d'OpenWeatherMap ou dans ma création du compte je suis très curieux de savoir le problème !

Partie 1 : extraction des données

Dossier *python/*

Logique générale

- Connexion à l'API météo en requérant les 7 derniers jours
- Parsing de la réponse pour constituer un premier dataframe avec les données
- Si un précédent fichier csv contenant d'anciennes données existe, on le charge puis on fusionne les deux dataframe en déduplicant les données (la donnée la plus fraîche est conservée en cas de doublon)
- Écriture du dataframe final en fichier csv

Gestion des appels API

J'ai utilisé la librairie *requests* pour les appels API en Python. L'API key (variable sensible) est stockée dans un fichier .env qui en théorie ne devrait pas être versionné.
J'ai implémenté une logique permettant de stopper l'exécution en cas de code retour (http) en erreur, tel que 401, 403, etc.
Pour les codes retour type 500, des erreurs côté serveur potentiellement temporaires, j'ai mis en place un retry avec limite pour retenter le call API. Un temps d'attente progressif est ajouté entre chaque retry.

Exécution du code

Docker permet la conteneurisation, c'est à dire que le code est exécuté dans un environnement séparé avec les bonnes versions des composants (Python etc). N'importe quel utilisateur qui a Docker installé sur son environnement peut exécuter le code, sans erreur de dépendances ou autre.

Pistes d'amélioration

- Refacto du code en POO : structure en classes et création d'objets
- Utiliser du logging plutôt que du print

Résultats

```
→  legalstart_weather_data git:(main) ✗ head output/historic_data.csv
city;date;avg_temp;min_temp;max_temp;humidity;description
Paris;2024-10-11;11.8;10.0;14.3;72;Patchy rain possible
Paris;2024-10-12;11.1;10.3;12.2;83;Light drizzle
Paris;2024-10-13;11.6;9.5;14.2;63;Overcast
Paris;2024-10-14;13.5;10.4;18.2;72;Cloudy
Paris;2024-10-15;16.3;14.1;18.7;84;Light drizzle
Paris;2024-10-16;19.6;16.2;24.7;74;Light rain shower
Paris;2024-10-17;16.8;15.7;18.5;91;Light rain shower
Paris;2024-10-18;16.3;14.5;18.8;90;Light rain
```

Partie 2 : chargement dans BigQuery

Fichier sql/*create_and_load.sql*

Logique générale

Je pars du principe que le fichier csv précédemment créé a été importé dans un bucket GCS nommé *historic_weather_bucket*.
Le script SQL peut directement s'exécuter dans BQ.

- Dans un premier temps je crée une table *cleaned_data* avec le schéma cible, qui contiendra les données dédupliquées.
- Ensuite je charge le csv depuis le bucket vers une table intermédiaire *raw_data*. J'utilise le mode OVERWRITE pour écraser le contenu de cette table intermédiaire.
- Enfin, avec un LEFT JOIN en filtrant sur les lignes à NULL dans la table cleaned j'insère les nouvelles données dans la table cleaned depuis la table raw.

Pistes d'amélioration

- Pour ne pas utiliser de bucket j'aurai pu importer le csv dans BigQuery à l'aide de l'outil de ligne de commande bq, en travaillant en local là où se trouve le fichier csv.
  - Par exemple: bq load —source_format=CSV mydataset.mytable /path/to/file.csv schema.json
- Pour gérer l'upsert et la déduplication, j'aurai pu utilisé l'instruction MERGE de BigQuery.

Résultats

cleaned_data

REQUÊTE | PARTAGER | COPIER | INSTANTANÉ | SUPPRIMER | EXPORTER

SCHÉMA | DÉTAILS | APERÇU | EXPLORATEUR DE TABLES BÊTA | INSIGHTS | TRAÇABILITÉ | PROFIL DE DONNÉES

Ligne	city	date	avg_temp	min_temp	max_temp	humidity	description
1	Paris	2024-10-13	11.6	9.5	14.2	63	Overcast
2	Paris	2024-10-14	13.5	10.4	18.2	72	Cloudy
3	Paris	2024-10-11	11.8	10.0	14.3	72	Patchy rain possible
4	Paris	2024-10-16	19.6	16.2	24.7	74	Light rain shower
5	Paris	2024-10-12	11.1	10.3	12.2	83	Light drizzle
6	Paris	2024-10-15	16.3	14.1	18.7	84	Light drizzle
7	Paris	2024-10-18	16.3	14.5	18.8	90	Light rain
8	Paris	2024-10-17	16.8	15.7	18.5	91	Light rain shower

Partie 3 : analyse des données

Fichier sql/*analyse.sql*

Logique générale

J'ai utilisé une CTE pour avoir la température minimale et la maximale des 7 derniers jours.
Ensuite, avec ces deux données, j'ai jointé avec la table pour obtenir la date et les infos complémentaires pour ces 2 jours.

Pistes d'amélioration

- Dans le cas où 2 jours auraient la même température minimale par exemple, la requête pourraient retourner plus que 2 lignes au total. On pourrait ajouté un LIMIT pour s'assurer du nombre de lignes retournées, avec un ORDER BY pour conserver le jour le plus récent par exemple.

Résultats