Jour 6 : Introduction aux fichiers *parquet* et manipulation / nettoyage de données avancé

Sommaire

Introduction

Aujourd'hui, vous allez étudier une variété de techniques de manipulation et de nettoyage de données semi-avancées. À mesure que vous avancerez dans votre exploration de la Big Data, maîtriser ces méthodes deviendra essentiel afin d'appréhender efficacement divers ensembles de données. Bien que ces techniques soient plus complexes que celles que nous avons abordées lors du jour 3, elles ne sont en aucun cas les plus avancées disponibles — il y a toujours plus à apprendre !

Cette journée se concentre sur l'exploration et la mise en œuvre de différentes méthodes de manipulation et nettoyage de données. Cependant, elle n'ira pas profondément dans les spécificités de pourquoi et / ou quand chaque méthode devrait être utilisée. C'est intentionnel, nous voulons que vous vous soyez proactifs et expérimentiez avec les techniques, et que vous recherchiez leurs applications. En explorant ces méthodes de manière indépendante, vous aurez une meilleure une compréhension de leurs applications pratiques et de quand elles pourraient être les plus efficaces.

Considérez cette journée comme une grande expérience. Ne vous contentez pas de réaliser chaque exercice, renseignez-vous sur chaque technique demandée, testez-les avec les datasets fournis avec le sujet ou d'autres trouvés sur Kaggle et ailleurs.

Partie 0 : Prérequis

Pour cette journée, vous aurez besoin des librairies Python suivantes :

- pandas, que vous avez déjà utilisé lors du jour 2,
- PyArrow, une librairie Big Data développée par Apache,
- fastparquet, une dépendance de pandas permettant de gérer les fichiers Parquet.

Ajoutez ces librairies au fichier *requirements.txt* comme vu lors des jours précédents et installez les prérequis dans un nouvel environnement virtuel pour la journée.

Partie 1 : Introduction aux fichiers *Parquet*

« Apache Parquet est un format de fichier de données orienté colonne et open source, conçu pour un stockage et une récupération efficaces des données. Il offre des schémas de compression et d'encodage haute performance pour gérer des données complexes en masse et est pris en charge dans de nombreux langages de programmation et outils d'analyse. » *Apache Parquet*

De façon générale, ce format permet non seulement d'économiser de l'espace de stockage mais accélère également considérablement les opérations de lecture et d'écriture, ce qui en fait un choix idéal pour analyser des grosses quantités de données.

Un fichier « flights.parquet » vous est fourni avec le sujet pour les exercices suivants.

Exercice 1 : Lire un fichier Parquet avec Pandas

Prototype : *read_parquet*(filename: str) -> pd.DataFrame
Rendu : partie1.py

Écrire une fonction *read_parquet* qui charge le fichier Parquet pris en paramètre dans un DataFrame Pandas et retourne les 10 premières lignes du DataFrame.

Exercice 2 : Extraire des colonnes spécifiques

Prototype : *read_parquet_columns*(filename: str, columns: list) -> pd.DataFrame
Rendu : partie1.py

Écrire une fonction *read_parquet_columns* qui charge uniquement les colonnes « columns » du fichier Parquet pris en paramètre dans un DataFrame et retourne le DataFrame résultant.

Exercice 3 : Lire un fichier Parquet par blocs

Prototype : *read_parquet_batch*(filename: str, batch_size: list) -> pd.DataFrame
Rendu : partie1.py

Écrire une fonction *read_parquet_batch* qui utilise la fonction iter_batches de la libarie Python PyArrow afin de lire le fichier Parquet pris en paramètre par blocs de données. Utiliser le paramètre de la fonction pour définir la taille de chaque bloc.
La fonction doit retourner dans un DataFrame les deux premières lignes de chaque bloc.
Attention à bien réinitialiser les index avant de retourner le DataFrame final.

Indice : Il est possible de convertir un batch PyArrow en DataFrame avec la fonction to_pandas.

Expérience : CSV vs Parquet

Les fichiers Parquet, conçus pour traiter d'immenses volumes de données, peuvent aisément contenir des dizaines de millions de « lignes ». Charger de tels fichiers dans leur totalité peut alors demander une quantité de mémoire vive très/trop importante en fonction de votre machine. Cependant, le format de données orienté colonne permet la lecture sélective, où seules les colonnes nécessaires sont extraites, réduisant ainsi considérablement le temps d'exécution et la quantité de mémoire requise. Il est également possible de lire les données par blocs, permettant de traiter les données par petits morceaux et de minimiser la charge en mémoire, particulièrement utile dans les environnements à ressources limitées.

En comparaison, le traitement de fichiers CSV, étant orientés ligne, impose la lecture complète du fichier afin de pouvoir filtrer les colonnes que l'on souhaite analyser. Ce processus peut entraîner une augmentation considérable du temps d'exécution, surtout avec de grands ensembles de données.

Utiliser la librairie Python time pour écrire un script qui calcule le temps de chargement dans un DataFrame :

- D'un fichier Parquet **VS** d'un fichier CSV. Les deux fichiers contenant exactement les mêmes données,
- D'un fichier Parquet **VS** une ou deux colonnes du même fichier Parquet,
- D'une ou deux colonnes d'un fichier Parquet **VS** les mêmes colonnes d'un fichier CSV.

Observez les différences de temps de chargement et tirez-en une conclusion !

Un fichier « flights.csv » contenant les mêmes données en format CSV que le fichier « flights.parquet », vous est fourni avec ce sujet. Vous pouvez également trouver d'autres Datasets sur Kaggle, comme celui-ci, contenant tous les articles Wikipedia en date du 1er juillet 2023, répartis en plusieurs fichiers Parquet.

Indice : Avec Pandas, vous pouvez convertir un fichier Parquet en CSV. Cela peut être utile si vous trouvez un Dataset pertinent mais qui n'est pas disponible dans les deux formats.

Partie 2 : Manipulation de données avancée

1. Indexation multiple

L'indexation multiple, ou indexation hiérarchique, est une fonctionnalité puissante de pandas qui vous permet de stocker et de manipuler des données avec plusieurs niveaux d'indices sur un seul axe. Cela permet une organisation et une récupération des données plus granulaires, facilitant ainsi les analyses et manipulations sur des ensembles de données complexes. En structurant les données sur plusieurs dimensions, l'indexation multiple améliore à la fois la flexibilité et l'efficacité des opérations sur les données, particulièrement dans des scénarios impliquant de grands ensembles de données multidimensionnels.

Pour les exercices suivants, un fichier CSV nommé « sales.csv » vous est fourni. Les DataFrames qui vous seront passés seront tous, sauf indication contraire, chargés depuis ce fichier. À vous d'analyser celui-ci et de comprendre ce qui vous est demandé.

Exercice 1 : Création d'un DataFrame Multi-Indexé

Prototype : *create_multi_index_df*(df: pd.DataFrame) -> pd.DataFrame
Rendu : partie2.py

Écrire une fonction *create_multi_index_df* qui prend un DataFrame et l'utilise pour créer un nouveau DataFrame multi-indexé par « year » puis par « region ».

Afin que le DataFrame soit indexé et découpé efficacement, il doit être trié par ses index. Deux méthodes sont possibles, trier le DataFrame avant de créer les nouveaux index, ou trier directement les index.

Indice : set_index peut accepter plusieurs colonnes à utiliser comme index.

Exercice 2 : Récupération de données Multi-Indexées

Prototype : *retrieve_multi_index_data*(df: pd.DataFrame, year: int, region: str) -> pd.DataFrame
Rendu : partie2.py

Écrire une fonction *retrieve_multi_index_data* qui prend un DataFrame issu de l'exercice précédent. Retourner un DataFrame contenant uniquement les données pour l'année et la région prisent en paramètre.

Exercice 3 : Agrégation multi-indexé

Prototype : *multi_index_aggregate*(df: pd.DataFrame) -> pd.DataFrame
Rendu : partie2.py

Écrire une fonction *multi_index_aggregate* qui prend un DataFrame et calcule le nombre total de produit vendu et le total des ventes pour chaque année et chaque région. Arrondir les résultats à la deuxième décimale.

Exercice 4 : Création de colonnes Multi-Indexé

Prototype : *columns_multi_index*(df: pd.DataFrame) -> pd.DataFrame
Rendu : partie2.py

Écrire une fonction *columns_multi_index* qui prend un DataFrame et reproduit les calculs d'agrégation de l'exercice précédent, en ajoutant la catégorie en troisième attribut d'agrégation. Vous devez ensuite pivoter l'index catégorie pour créer des colonnes Multi-Indexé. Arrondir les résultats à la deuxième décimale.

Indice : unstack

Exemple :

```
df = pd.read_csv('resources/sales.csv',
keep_default_na=False, na_values='')
micdf = columns_multi_index(df)
print(micdf)
```

```
(.venv) → Jour6 git:(main) × python3 partie2.py | cat -e
                quantity                ...      total_price             $
category     Classic Cars Motorcycles  ... Trucks and Buses Vintage Cars$
year region                            ...                              $
2003 APAC           472.0       194.0  ...         11297.54     77262.50$
     EMEA          3828.0       672.0  ...        129254.54    176809.84$
     Japan          701.0        43.0  ...         30221.96     15663.72$
     NA            2989.0       948.0  ...         86471.63    222817.88$
2004 APAC           643.0       233.0  ...         54752.44     45269.56$
     EMEA          5718.0       835.0  ...         83267.91    362479.36$
     Japan          262.0       235.0  ...         12121.24      1257.27$
     NA            3149.0      1337.0  ...        149610.68    218296.45$
2005 APAC           123.0        83.0  ...         26756.62     40403.61$
     EMEA          2285.0       685.0  ...         47088.82     47298.97$
     Japan          122.0         NaN  ...              NaN      7425.37$
     NA            1124.0       260.0  ...         32809.37    136283.74$
$
[12 rows x 14 columns]$
```

Exercice 5 : Inverser des colonnes Multi-Indexé

Prototype : swap_*columns_multi_index*(df: pd.DataFrame) -> pd.DataFrame
Rendu : partie2.py

Si vous observez bien le DataFrame résultant de l'exercice précédent, vous remarquerez que les colonnes principales sont « quantity » et « total _price » et chacune contient les valeurs pour chaque catégorie. Cette organisation ne semble pas très logique, alors essayons d'inverser les colonnes pour que les catégories soient les colonnes principale et contenant chacune leurs valeurs de « quantity » et « total_price ».

Écrire une fonction swap_*columns_multi_index* qui prends un DataFrame issu de l'exercice précédent et inverse les colonnes de catégories avec les colonnes « quantity » et « total_price » de telle sorte que chaque « category » ait en sous-colonne « quantity » et « total_price ». Retourner le DataFrame résultant.

Indice : swaplevel. N'oubliez pas de trier les colonnes afin que les informations soient regroupées correctement.

Exemple :

```
df = pd.read_csv('resources/sales.csv',
keep_default_na=False, na_values='')
micdf = columns_multi_index(df)
swapdf = swap_columns_multi_index(micdf)
print(swapdf)
```

```
(.venv) →  Jour6 git:(main) ✗ python3 partie2.py | cat -e
category    Classic Cars                ... Vintage Cars             $
               quantity total_price     ...     quantity total_price$
year region                             ...                          $
2003 APAC          472.0    61108.85    ...        766.0    77262.50$
     EMEA         3828.0   527131.56    ...       1773.0   176809.84$
     Japan         701.0   104143.21    ...        182.0    15663.72$
     NA           2989.0   427333.34    ...       2238.0   222817.88$
2004 APAC          643.0    87687.32    ...        434.0    45269.56$
     EMEA         5718.0   737970.11    ...       3415.0   362479.36$
     Japan         262.0    38429.17    ...         21.0     1257.27$
     NA           3149.0   428790.01    ...       2204.0   218296.45$
2005 APAC          123.0     7983.86    ...        370.0    40403.61$
     EMEA         2285.0   278242.53    ...        566.0    47298.97$
     Japan         122.0    18835.02    ...         69.0     7425.37$
     NA           1124.0   149842.23    ...       1146.0   136283.74$
$
[12 rows x 14 columns]$
(.venv) →  Jour6 git:(main) ✗
```

Exercice 6 : Récupération de colonnes et lignes Multi-Indexé

Prototypes :

- *retrieve_multi_index_column*(df: pd.DataFrame, category: str) -> pd.DataFrame
- *retrieve_multi_index_basic(*df: pd.DataFrame, category: str, year: int) -> pd.DataFrame
- *retrieve_multi_index_advanced(*df: pd.DataFrame, region: str, sub_column: str) -> pd.DataFrame

Rendu : partie2.py

Écrire une fonction *retrieve_multi_index_columns* qui prends un DataFrame issu de l'exercice précédent. Retourner un DataFrame contenant uniquement les données pour la catégorie donnée.

Écrire une fonction *retrieve_multi_index_basic* qui prends un DataFrame issu de l'exercice précédent. Retourner un DataFrame contenant uniquement les données pour la catégorie et l'année donnée.

Écrire une fonction *retrieve_multi_index_advanced* qui prends un DataFrame issu de l'exercice précédent. Retourner un DataFrame contenant uniquement les données pour la région (toute années confondues) et la sous colonne donnée (« quantity » ou « total_price »).

Indice : Pour récupérer les bonnes lignes, vous aurez besoin d'effectuer du slicing comme décrit dans la documentation de pandas. Pour les colonnes, lisez la documentation de la fonction xs.

2. Tables de pivot

Ce que vous avez faire dans les exercices précédents, c'est la création (et la manipulation) de table de pivots de façon manuelle. Cependant, il existe une fonction pivot_table qui permet de créer les tables de pivots bien plus simplement. Celle-ci simplifie le processus de création car elle gère automatiquement le regroupement (groupby) et l'agrégation (aggregate) des données.

Exercice 7 : Créer une table de pivot

Prototype :

- create_pivot_table_basic(df: pd.DataFrame) -> pd.DataFrame
- create_pivot_table_advanced(df: pd.DataFrame) -> pd.DataFrame

Rendu : partie2.py

Écrire une fonction *create_pivot_table_basic* prenant en paramètre un DataFrame. En utilisant la fonction pivot_table, reproduire le DataFrame de l'exercice 4, partie 2. Retourner le DataFrame résultant. Arrondir les résultats à la deuxième décimale.

Écrire une fonction *create_pivot_table_advanced* prenant en paramètre un DataFrame. En utilisant la fonction pivot_table et l'inversion de colonnes, reproduire le DataFrame de l'exercice 5, partie 2 et y ajouter le nombre de vente, le prix et la quantité moyenne pour chaque catégorie. Retourner le DataFrame résultant. Arrondir les résultats à la deuxième décimale.

3. Fonctions de fenêtrage et fenêtres glissantes

Les fonctions de fenêtrage et les fenêtres glissantes sont des outils essentiels dans l'analyse de données, permettant des calculs sur des sous-ensembles de données au sein d'un DataFrame. Les fonctions de fenêtrage vous permettent d'effectuer des opérations qui prennent en compte une plage de lignes liées à la ligne actuelle, comme des calculs de moyennes mobiles (SMA, EMA, …), et de tout autre types de calculs cumulatifs.

Les fenêtres glissantes, un type de fonction de fenêtrage, se concentrent spécifiquement sur des intervalles glissants de données, utiles pour lisser ou moyenner des données aberrantes, identifier des tendances et d'autres analyses statistiques sur une période spécifiée.

Ces techniques sont particulièrement utilisées dans l'analyse de séries temporelles (cours de bourse / crypto, températures, …), où il est crucial de comprendre les changements et les patterns dans le temps.

Exercice 8 : Utiliser une fenêtre glissante

Prototype : avg*_price_rolling_window*(df: pd.DataFrame) -> pd.DataFrame
Rendu : partie2.py

Écrire une fonction *avg_price_rolling_window* prenant en paramètre un DataFrame. Créer une fenêtre glissante qui calcule le prix total moyen sur une période de 7 jours (moyenne mobile simple). Il doit y avoir à minima 3 valeurs dans la fenêtre pour effectuer la moyenne. Stocker le résultat dans une nouvelle colonne « rolling_avg » et retourner le DataFrame. Arrondir les résultats à la deuxième décimale.

Indice : Il est possible d'avoir un index au format datetime ou équivalent et de définir une période temporelle pour la fenêtre glissante avec la fonction rolling.

Information : Afin de ne pas avoir à faire de transformations supplémentaires, toutes les dates dans le Dataset de test sont uniques.

Exercice 9 : Mettre en avant les valeurs aberrantes

Prototype : highlight_outliers(df: pd.DataFrame) -> pd.DataFrame
Rendu : partie2.py

Écrire une fonction *highlight_outliers* qui prends un DataFrame issue de l'exercice précédent. Créer une nouvelle colonne « outliers » contenant un booléen indiquant si le prix total de vente est supérieur à 2.5 fois la moyenne mobile associée.

Indice : Il est possible de créer une nouvelle colonne basée sur les valeurs des autres colonnes grâce à apply et une lambda.

Partie 3 : Nettoyage de données avancé

Pour les exercices 1 et 2, un fichier CSV nommé « sales_unclean.csv » vous est fourni. Pour les exercices 3 et 4, un fichier CSV nommé « sales_outliers.csv » vous est fourni. Les DataFrames qui vous seront passés seront tous, sauf indication contraire, chargés depuis ces fichiers pour les exercices associés. À vous d'analyser ceux-ci et de comprendre ce qui vous est demandé.

Exercice 1 : Gérer les données manquantes

Lors du jour 2, vous avez commencé à explorer les techniques de nettoyage de données de base, y compris la suppression simple des lignes contenant des données manquantes. Cependant, dans des scénarios de données avancés, il est souvent plus avantageux d'adapter notre approche en fonction du type et de l'importance des données manquantes. Plutôt que de les supprimer purement et simplement, les valeurs manquantes peuvent souvent être imputées en utilisant des informations provenant d'autres données du même Dataset. Il peut simplement s'agir d'une déduction logique (par exemple, si le pays est « France », on peut déduire que la région est « EMEA ») ou de calculs statistiques comme des moyennes /médianes. Il est également possible d'utiliser des techniques plus avancées comme une régression linéaire / stochastique ou un algorithme de clustering comme le k-means.

Chaque méthode comporte des avantages et des inconvénients à considérer ; par exemple, l'utilisation de moyennes est rapide mais peut être trompeuse si les données ne sont pas uniformément distribuées et peut poser des problèmes en fonction du type d'analyse, tandis que des techniques plus avancées comme le k-means offrent une précision accrue au prix d'une plus grande demande de puissance de calcul.

Prototypes :

- *impute_region*(df: pd.DataFrame) -> pd.DataFrame
- *impute_quantity*(df: pd.DataFrame) -> pd.DataFrame
- *impute_category(*df: pd.DataFrame) -> pd.DataFrame

Rendu : partie3.py

Écrire une fonction *impute_region* prenant un DataFrame. Compléter les données manquantes de région grâce aux informations de pays. Si le pays est vide, supprimer la ligne, sinon imputer la région. Réinitialiser les index puis retourner le DataFrame résultant.

Indice : Analysez le dataset pour savoir quelle région correspond à quel ensemble de pays. Par exemple, la région « Japan » n'est pas uniquement associée au pays « Japan ».

Écrire une fonction *impute_quantity* prenant un DataFrame. Compléter les données manquantes de quantité par la moyenne de la colonne. Retourner le DataFrame résultant.

Écrire une fonction *impute_category* prenant un DataFrame. Compléter les données manquantes de catégorie par la catégorie la plus fréquente. Retourner le DataFrame résultant.

Exercice 2 : Gérer les données inconsistantes

Les données inconsistantes sont tout aussi courantes que les données manquantes dans des ensembles de données et peuvent survenir sous diverses formes et sont souvent dues à des saisies utilisateur. Les exemples incluent des adresses (« USA » vs. « United States » vs. « United States of America »), des formats de date variés (« 24/12/2024 » vs. « 12/24/2024 » vs. « 2024/12/24 »), ou des désignations de genre incohérentes (« F » vs. « Femme » vs. « f »). Il est crucial de traiter ces incohérences pour uniformiser le format de stockage des données, facilitant ainsi les analyses futures et améliorant la propreté de votre stockage de données. Utiliser des règles de validation ou des procédures de standardisation peut aider à résoudre ces problèmes et assurer l'intégrité et l'utilité des données collectées.

Prototypes :

- *handle_inconsistent_dealsize*(df: pd.DataFrame) -> pd.DataFrame
- *handle_inconsistent_dates*(df: pd.DataFrame) -> pd.DataFrame

Rendu : partie3.py

Écrire une fonction *handle_inconsistent_dealsize* qui prend en paramètre un DataFrame. Convertir toutes les tailles dans le format « S / M / L ». Les formats valides sont les suivants (case-insensitive) :

- S -> « s », « Small », « 1 »
- M -> « m », « Medium », « 2 »
- L -> « l », « Large », « 3 »

Si le format n'est pas reconnu, assigner M par défaut. Retourner le DataFrame résultant.

Écrire une fonction *handle_inconsistent_dates* qui prend en paramètre un DataFrame. Convertir toutes les dates dans le format « 2024-12-24 23:42:00 ». Il peut y avoir plusieurs types de formats différents dans le DataFrame, analysez celui-ci pour trouver tous les formats existants afin de les convertir proprement. En cas de doute sur le format entre dd/mm/yyyy et mm/dd/yyyy, privilégier le format avec le jour en premier. Convertir toutes les valeurs de la colonne « date » en type « datetime » puis retourner le DataFrame résultant.

Information : En condition réelle, il faudrait s'assurer du format (le jour ou mois en premier, par exemple avec la date 08/10/2024, le mois est août ou octobre ?). Dans notre dataset, cela est possible car nous avons une colonne avec le mois de la transaction, cependant cela compliquerait l'implémentation et ce n'est pas le but ici. Vous pouvez cependant vous amuser à le faire à la fin de la journée.

Exercice 3 : Identifier les valeurs aberrantes

La notion de valeurs aberrantes a déjà été légèrement introduite lors de la partie précédente, mais celles-ci étaient aberrantes par rapport à un sous-ensemble de données. Ici, l'objectif est d'identifier les valeurs aberrantes par rapport à la totalité de l'ensemble. En fonction des analyses que l'on souhaite faire, les valeurs aberrantes peuvent avoir un impact plus ou moins important sur le résultat, par exemple, en faussant la moyenne ou la médiane. Il est donc essentiel d'être capable de les identifier afin de prendre des mesures appropriées, telles que la suppression, l'ajustement des valeurs, ou la création de limites maximales et minimales, selon le besoin spécifique de l'analyse.

Prototypes :

- *retrieve_quantity_outliers*(df: pd.DataFrame) -> pd.DataFrame
- *handle_unit_price_outliers*(df: pd.DataFrame) -> pd.DataFrame

Rendu : partie3.py

Écrire une fonction *retrieve_quantity_outliers* qui prend un DataFrame en paramètre. Identifier les quantités aberrantes avec la méthode de l'écart interquartile. On considère une valeur aberrante comme une valeur étant supérieur à 1,5 fois l'IQR au-dessus du troisième quartile ou en-dessous du premier quartile. Retourner un DataFrame contenant uniquement les valeurs aberrantes identifiées trié par quantité croissante.

Écrire une fonction *handle_unit_price_outlier* qui prends un DataFrame en paramètre. Identifier les prix unitaires aberrants avec la méthode de l'écart interquartile. Supprimer les valeurs aberrantes sous le premier quartile et modifier les valeurs aberrantes au-dessus du troisième quartile par la valeur équivalente à Q3 + 1.5 * IQR. Retourner le DataFrame résultant trié par prix unitaire croissant.

Exercice 4 : Normaliser les données

La normalisation des données est une étape importante en Machine Learning (ML) et en analyse de données, car elle permet de mettre toutes les valeurs sur une échelle commune (par exemple, entre 0 et 1). Ce processus aide les algorithmes à analyser et à comparer les données de manière uniforme, permettant de converger plus rapidement et évitant que les caractéristiques à grande échelle dominent le processus d'apprentissage. Ce processus est à faire après le nettoyage des données manquantes et inconsistantes pour éviter les biais et distorsions. Plusieurs techniques de normalisation peuvent être utilisées, telles que la normalisation Min-Max, Z-Score, la normalisation logarithmique, … chacune adaptée à différents types de données et objectifs d'analyse.

Prototypes :

- *normalize_total_price(*df: pd.DataFrame) -> pd.DataFrame
- *normalize_quantity*(df: pd.DataFrame) -> pd.DataFrame
- *normalize_unit_price*(df: pd.DataFrame) -> pd.DataFrame

Rendu : partie3.py

Écrire une fonction *normalize_total_price* qui prend en paramètre un DataFrame. Appliquer une normalisation logarithmique sur la colonne « total_price ». Retourner le DataFrame résultant. Arrondir les résultats à la dixième décimale.

Information : On applique une normalisation logarithmique sur la colonne « total_price » car elle a une variance extrêmement élevée (les données sont très dispersées par rapport à la moyenne). Cela permet de changer la distribution des données.
Vous pouvez voir la variance des colonnes d'un DataFrame avec df.var (à condition que le DataFrame ne contiennent que des colonnes représentant des nombres).

Écrire une fonction *normalize_quantity* qui prend en paramètre un DataFrame. Appliquer une normalisation Z-Score sur la colonne « quantity ». Retourner le DataFrame résultant. Arrondir les résultats à la dixième décimale.

Indice : Observez la moyenne de la colonne « quantity » après avoir appliqué la normalisation. Celle-ci devrait tendre vers 0.

Écrire une fonction *normalize_unit_price* qui prend en paramètre un DataFrame. Appliquer une normalisation Min-Max sur la colonne « unit_price ». Retourner le DataFrame résultant. Arrondir les résultats à la dixième décimale.

Expérience : Observez la variance des colonnes avant et après la normalisation.