Nettoyage et préparation des données avec Pandas

Avant de pouvoir explorer les données contenues dans un jeu de données, il faut le nettoyer et le préparer pour le rendre exploitable à des fins d'analyse. Cela comprend les étapes suivantes : **La conversion des types de données, la détection et suppression des doublons, le traitement des valeurs manquantes, la correction des incohérences textuelles, la gestion des valeurs aberrantes, et la création de colonnes utiles pour l'analyse**. L'ordre dans lequel ces tâches de nettoyage des données sont effectuées, peut varier en fonction de l'ensemble de données.

Étape N°1 : Types de données

En utilisant Pandas pour lire des données, les colonnes se voient automatiquement attribuer un type de données.

Conseils :

→ Utiliser l'attribut **`.dtypes`** pour afficher le type de données de chaque colonne du DataFrame. Il faut noter que parfois, les colonnes numériques (**`int`**, **`float`**) et les colonnes de date et d'heure (**`datetime`**) ne sont pas reconnues correctement par Pandas et sont importées en tant que colonnes de texte (**`object`**).

→ Utiliser la méthode **`.info()`** en tant qu'alternative pour afficher des informations supplémentaires en plus des types de données.

→ Utiliser **`pd.to_datetime()`** pour convertir une colonne de type **`object`** en une colonne de type **`datetime`**.

→ Utiliser **`pd.to_numeric()`** pour convertir une colonne de type **`objet`** en colonne numérique. Pour supprimer les caractères non numériques ($, %, etc.), utiliser **`str.replace()`**.

Une alternative consiste à utiliser **`Series.astype()`** pour convertir en types de données plus spécifiques tels que **`int`**, **`float`**, **`object`** et **`bool`**, mais **`pd.to_numeric()`** peut gérer les valeurs manquantes (**`NaN`**), contrairement à **`Series.astype()`** qui lève une erreur dans ce cas. On peut utiliser l'argument **`errors="coerce"`** qui permet de convertir les valeurs valides tout en remplaçant celles qui ne le sont pas (ou non numériques) par **`NaN`**, sans générer d'erreur. À noter que **`astype(...)`** peut également être appliquée à un DataFrame pour convertir plusieurs colonnes en une seule instruction, par exemple : **`df.astype({"col1": int, "col2": float})`**

Étape N°2 : Gestion des données dupliquées

Lorsqu'on commence à travailler avec un tableau contenant beaucoup de données, il ne faut pas seulement prendre en compte la gestion des valeurs manquantes, mais aussi celle des informations dupliquées. En effet, cela peut fausser les conclusions que l'on fait sur un jeu de données si certaines informations sont dupliquées de nombreuses fois. Des informations dupliquées peuvent être des lignes entières dupliquées :

Étape	Méthode/Syntaxe	Explication
Détection de doublons :	`df.duplicated()`	Renvoie une série de booléens. **`True`** pour les lignes dupliquées (à partir de la 2e occurrence), **`False`** sinon.
	`df[df.duplicated()]`	Affiche uniquement les lignes dupliquées, sans la première occurrence.
	`df[df.duplicated(keep=False)]`	Affiche **toutes** les occurrences des lignes dupliquées (y compris la première).
	`df[df.duplicated(subset=['col 1', 'col2'])]`	Détecte les doublons en se basant seulement sur certaines colonnes.
	Paramètre **`keep`** :	
	`keep='first'` (par défaut)/`keep='last'`	Marque **toutes** les lignes dupliquées comme **`True`** sauf la **première /dernière**.
	`keep=False`	Marque toutes les occurrences comme **`True`**.
Nombre de doublons :	`df.duplicated().sum()`	Renvoie le **nombre de lignes dupliquées** (à partir de la 2e occurrence).
	`df[df.duplicated()].shape`	Renvoie **la dimension (lignes, colonnes)** du DataFrame contenant uniquement les lignes dupliquées.

Suppression des doublons :	`df.drop_duplicates(inplace=True)`	Supprime les doublons **en gardant la première occurrence**.
	`df.drop_duplicates(subset=['col1', 'col2'], keep='last', inplace=True)`	Supprime les doublons en se basant sur certaines colonnes, en gardant la dernière occurrence.

Étape N°3 : Gestion des données manquantes

Il existe différentes façons de représenter les données manquantes en Python : **`np.NaN`**, **`pd.NA`** et **`None`** (ne permet pas de calcul numérique). Il est important de prendre connaissance de la proportion des valeurs manquantes, voire de les traiter.

→ Identifier les valeurs manquantes en utilisant les méthodes suivantes :

- **`df.isna()`**: Renvoie un DataFrame de booléens (**`True`** si la valeur est **`NaN`**).
- **`df.isna().sum()`** : Affiche le nombre de **`NaN`** par colonne.
- **`df[df.isna().any(axis=1)]`** : Affiche les lignes contenant au moins un **`NaN`**.
- **`df.info()`** : Résume les colonnes et le nombre de valeurs non nulles.
- **`df['col'].value_counts(dropna=False)`** : Compte les valeurs, y compris les NaN.

→ Supprimer les valeurs manquantes en utilisant les méthodes suivantes :

- **`df.dropna()`**: Supprime les **lignes** contenant au moins un **`NaN`**.
- **`df.dropna(how='all')`**: Supprime les lignes où **toutes** les colonnes sont **`NaN`.**
- **`df.dropna(thresh=n)`** : Garde les lignes avec au moins n valeurs non-nulles.
- **`df.dropna(subset=['col1'])`** : Supprime les lignes où **`col1`** est **`NaN`**.
- **`df.dropna(axis='columns')`** : Supprime les colonnes contenant au moins un **`NaN`**.
- **`df[df['col'].notna()]`** : Garde uniquement les lignes où col n'est pas **`NaN`**.

Par défaut, ces opérations ne modifient pas le DataFrame original.

Ajouter **`inplace=True`** ou réassigner.

→ **Remplacer les valeurs manquantes en utilisant les méthodes suivantes** :

-**`df.fillna(0)`** : Remplace les **`NaN`** par 0

-**`df.fillna(df['col'].mean())`** : Remplace les **`NaN`** de **`col`** par sa **moyenne**.

-**`df['col'] = df['col'].fillna(df['col'].median())`** : Remplace les **`NaN`** de **`col`** par sa **médiane**.

-**`df.loc[i, 'col']= valeur`** : Met à jour une cellule en utilisant l'expertise métier.

Étape N°4 : Textes incohérents et Typos(fautes de frappe)

Les textes incohérents et les fautes de frappe (typos) dans un ensemble de données sont représentés par des valeurs qui sont soit incorrectes de quelques chiffres ou caractères, ou incompatibles avec le reste d'une colonne. Bien qu'il n'existe pas de méthode spécifique pour les identifier, on peut suivre les deux approches suivantes pour vérifier une colonne en fonction de son type de données :

-pour les données catégorielles : examiner les valeurs uniques dans la colonne (par exemple : RG, WI, Wimbledon, ici WI et Wimbledon désigne le même tournoi)

-pour les données numériques : consulter les statistiques descriptives (par exemple, l'intervalle [**`min(), max()`**].

On peut utiliser les méthodes suivantes pour résoudre ces problèmes :

→ **`.loc[]`** pour mettre à jour une valeur à un emplacement spécifique selon le domaine d'expertise.

→ **`np.where(condition, if_true, if_false)`** pour mettre à jour des valeurs dans une colonne en fonction d'une condition logique (par exemple changer les valeurs de Wimbledon par WI : **`df["col"] = np.where(df["col"] == "Wimbledon", "WI", df["col"]`**).

→ **`.map(dictionnaire)`** pour mapper un ensemble de valeurs vers un autre ensemble de valeurs (par exemple : **`df["col"].map({"Wimbledon" :"WI"}).`** Ici chaque valeur de Wimbledon sera remplacée par WI.

→ Méthodes de chaîne de caractères telles que **`str.lower()`**, **`str.strip()`** et **`str.replace()`** pour nettoyer les données textuelles.

Étape N°5 : Gestion des valeurs aberrantes

Un outlier est une valeur très différente des autres dans un jeu de données (beaucoup plus grande ou plus petite). Les outliers peuvent fausser les calculs statistiques (moyenne, variance…) et les modèles prédictifs s'ils ne sont pas détectés. Plusieurs approches permettent d'identifier les outliers :

→ **Histogramme** qui visualise la distribution d'une variable. Les valeurs éloignées de la majorité apparaissent comme des barres isolée. On peut utiliser **`seaborn`** pour afficher un histogramme :

```
import seaborn as sns

sns.histplot(df)
```

→ **Boxplot** qui trace les statistiques descriptives avec **`sns.boxplot(x=df.nom_col)`**. Les outliers sont les points **hors de l'intervalle** **`[Q1 - 1.5×IQR, Q3 + 1.5×IQR]`**, **`Q1`** (1er quartile où 25 % des valeurs sont inférieures ou égales à **`Q1`**), **`Q3`** (3e quartile où 75 % des valeurs sont inférieures ou égales à **`Q3`**), et **`IQR = Q3 – Q1`** (C'est la plage centrale contenant 50 % des valeurs).

→ **Écart-type σ**: Pour les données normalement distribuées, toute valeur à plus de 3σ de la moyenne est considérée comme un outlier (seuil ajustable à 2σ, 4σ…). Par exemple :

```
moy = np.mean(df.nom_col)

sd = np.sd(df.nom_col)

outliers = [elt for elt in df.nom_col if (elt < moy -3*sd) or (elt > moy + 3*sd)]
```

Étape N°6 : Création de nouvelles colonnes

Après avoir nettoyé les types de données et résolu les problèmes, on peut toujours ne pas disposer exactement des données dont on a besoin. Dans ce cas, il faut créer de nouvelles colonnes à partir de données existantes pour faciliter l'analyse. Cela peut être fait de plusieurs manières, en fonction du type de données qu'on a :

→ Pour les colonnes numériques, on peut calculer des pourcentages, appliquer des calculs conditionnels, etc.

→ Pour les colonnes de date et d'heure, on peut extraire des composants de date et d'heure, appliquer des calculs de date et d'heure, etc.

Par exemple, utiliser **df['col_l'].dt.composante**, telle que **composante** est égale à : **Date** pour la date sans heure, **year** pour l'année, **month** pour le mois (1-12), **day** pour le jour, **dayofweek** (0 pour lundi et 6 pour dimanche), **time** (heure sans date), **hour** pour l'heure (0-23) et **minute** et **second** pour minute et seconde (0-59).

Pour effectuer des calculs de date et d'heure entre colonnes, on utilise des opérations arithmétiques de base. On peut également **pd.to_timedelta(valeur, unit)** pour ajouter ou soustraire une période de temps spécifique. Par exemple : **valeur =3 et unit="W"** veut dire qu'on veut ajouter deux semaines, **unit** peut aussi prendre les valeurs suivantes : **D** pour jour, **H** pour heure, **T** pour minute et **S** pour seconde).

→ Pour les colonnes de texte, on peut extraire du texte, le diviser en plusieurs colonnes, rechercher des motifs, etc. Par exemple : **.str[deb : fin]** extrait un texte dont les positions de ses lettres vont de **deb** à **fin-1**, **.str.split(delimiteur)** pour séparer une colonne en plusieurs colonnes en utilisant un délimiteur, .**str.contains(pattern)** pour chercher un motif dans une chaine.