Extraction des données de consommation éCO²Mix dans les métropoles

# Objectif

L’objectif est de récupérer la donnée de consommation électrique pour les villes de Toulon et Nice en temps réel. En pratique, temps réel sera adaptable et à repréciser par la suite ; pour le moment et ici, l’objectif est de récupérer les données au pas de temps journalier, et si possible horaire.

# Informations

Lien de l’API :

<https://odre.opendatasoft.com/api/explore/v2.1/catalog/datasets/eco2mix-metropoles-tr/records>

Contraintes :

-Les données sont mises à jour 4 fois par jour, à minuit, 6h, 12h et 18h

-Une seule requête ne peut pas récupérer + de 100 lignes. Il faudra alors faire plusieurs requêtes si on veut par exemple récupérer chaque quart d’heure sur plusieurs jours

-L’API possède un paramètre “offset”, qui permet de sauter les x premières lignes d’une requête. Par exemple, si on trie par date décroissante, lorsque offset = 0, une requête récupèrera les 100 lignes les plus récentes. En fixant offset à 1000, on saute les 1000 lignes les plus récentes pour remonter plus loin dans l’historique. La contrainte est la suivante : Le offset (ajouté à la limite qui vaut 100 maximum) ne peut dépasser 10 000.

On le verra par la suite, mais cela implique qu’en extraction ¼ horaire, il est impossible de remonter plus loin qu’à il y a 51 jours (9900 lignes / 192 lignes par jour = 51.6 jours)

# Extraction ¼ horaire

Une image contenant texte, capture d’écran, menu, noir et blanc

Description générée automatiquement

On extraie les données telles qu’elles sont organisées initialement dans l’API. En filtrant bien sûr sur les villes, et en triant par date décroissante.

L’appel à la fonction se fait ainsi :

quart\_df = extract\_conso\_quarth(n\_days = 8, days\_delay = 0)

Ici on a choisi de récolter les données sur 8 jours, et sur les 8 derniers puisque l’on met 0 comme délai.

Avantage : Pas temporel très précis

Inconvénients : Lourdeur des requêtes, difficulté d’application du ¼ horaire qui risque de ne pas être retenu en pratique pour être croisé avec d’autres données, l’horaire devant être mieux adapté.

# Extraction horaire

Une image contenant texte, capture d’écran, menu, Police

Description générée automatiquement

Cette extraction nécessite l’extraction ¼ horaire, puisqu’on réalise un groupby sur le DataFrame ¼ horaire. La colonne « missing\_percentage » indique la part de données omises pour faire le calcul au sein d’une heure. Par exemple, à 7h à Toulon, on a 75% de données manquantes, c’est-à-dire qu’un seul ¼ d’heure a été renseigné au lieu de 4 pour les heures précédentes. Cette colonne fait office d’avertissement.

Ces données sont ensuite envoyées en base dans le serveur PostgreSQL, sous 2 formes :

* Dans conso\_elec\_heure\_test\_v0, une table « snapshot », qui se vide et se reremplit, avec le nombre de jours spécifiés, à chaque exécution
* Dans conso\_elec\_heure\_test\_hist, une table « historique », jamais supprimée, seulement mise à jour. Les nouvelles lignes sont insérées et les lignes avec de nouvelles valeurs de consommation sont mises à jour.

Voici ce que cela donne un lundi matin après le week-end :

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

La 1re table a été rechargée entièrement depuis l’API avec les 2367 derniers couples (heure, ville) correspondant à 50 jours.

La seconde a été remplie de 132 lignes supplémentaires, issues du week-end, et a mis à jour la donnée incomplète du vendredi.

Résultat dans DBeaver :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Avantage : Pas temporel précis, Bonne possibilité de croisement avec d’autres données temps réel

Inconvénients : Lourdeur des requêtes (la même que pour le ¼ horaire)

# Extraction journalière

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

Ici on ne passe plus par l’extraction ¼ horaire, on requête différemment l’API (group by sur les dates).

Gain : Lors de la requête à l’API, 1 ligne représente 1 jour, on a donc pour une ville 1 jour = 1 ligne au lieu de 1 jour = 96 lignes précédemment. Cela efface les contraintes du offset et permet de remonter sans problème à + de 1000 jours, tout en étant plus rapide que l’extraction ¼ horaire, car moins de requêtes nécessaires.

On peut donc remonter à l’historique sur plusieurs années, et mettre en perspective les consommations journalières avec le nombre de données « manquantes » par jour. Ici, on considèrera arbitrairement qu’un quart d’heure « manquant » est un quart d’heure avec une consommation < 50 MWh, voici ce que cela donne pour Nice :

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

Et pour Toulon :

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

Conclusion, les données ne semblent fiables que depuis l’été 2023. Il y a un gros trou à Toulon de fin 2022 à mi-2023.

Ces données sont ensuite envoyées en base dans le serveur PostgreSQL, sous 2 formes :

* Dans conso\_elec\_jour\_test\_v0, une table « snapshot », qui se vide et se reremplit, avec le nombre de jours spécifiés, à chaque exécution
* Dans conso\_elec\_jour\_test\_hist, une table « historique », jamais supprimée, seulement mise à jour. Les nouvelles lignes sont insérées et les lignes avec de nouvelles valeurs de consommation sont mises à jour.

Voici ce que cela donne un lundi matin après le week-end :

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Les 10 jours ont été rechargés entièrement dans la 1re table (20 lignes car 2 villes).

Dans la 2e table, les 6 lignes insérées correspondent à samedi, dimanche et lundi pour les 2 villes, tandis que les 2 lignes mises à jour sont les 2 lignes du vendredi, qui étaient incomplètes en quittant le bureau vendredi soir.

Voici le résultat dans DBeaver (bas de la table) :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Avantage : Facilité de manipulation (historique lointain, rapidité des requêtes)

Inconvénient : Pas temporel + large