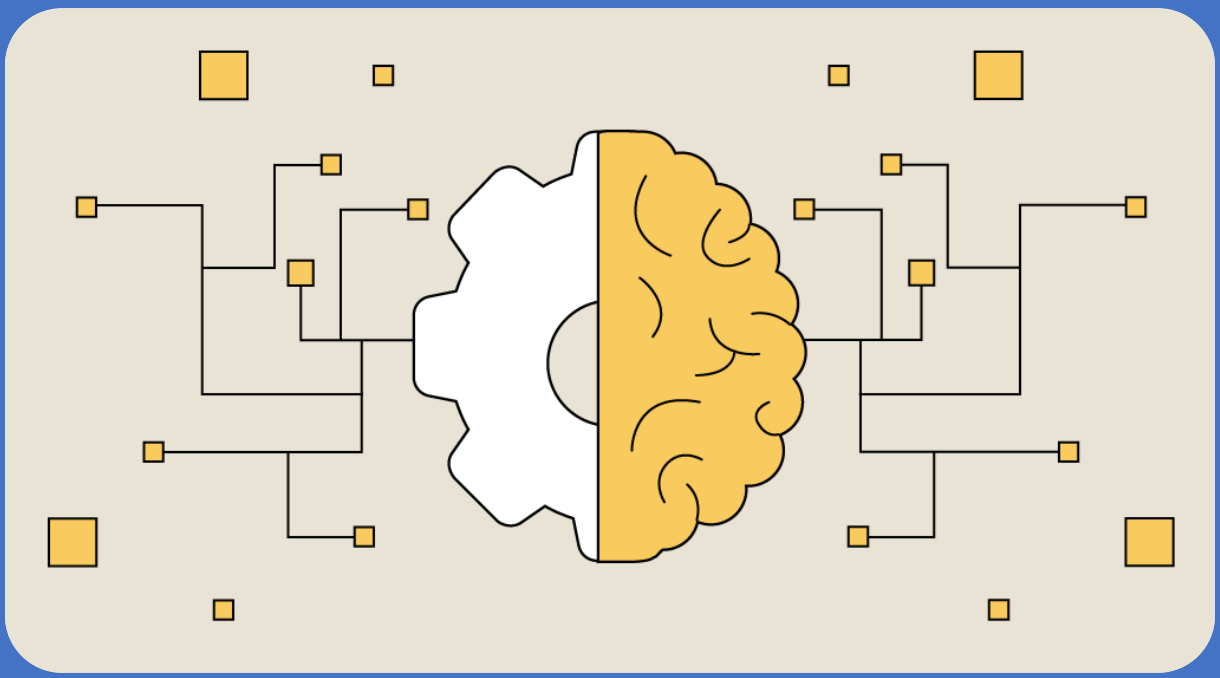
Rapport

Projet Machine Learning

Explication du prix de l'électricité



Equipe:4

SENECHAL-Morgan

SEBBANE-Ryan

SAVORY-Edwin

PIMENTA SILVA-Lionel

Professeur : CHAKCHOUK, Faten

Table des matières

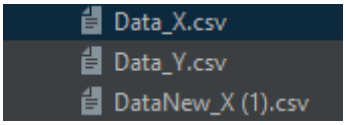
Introduction

L'alimentation électrique est un produit de tous les jours dont le montant varie en fonction de nombreux facteurs. Divers événements peuvent affecter simultanément la production et la demande d'électricité (rivalités géopolitiques, fluctuations des coûts des matières premières, trafic commercial entre les pays européens...). La modélisation du prix de l'alimentation électrique en fonction de ces facteurs peut donc être difficile. Dans ce contexte, notre projet cherche à modéliser le prix de l'alimentation électrique en France et en Allemagne en utilisant des données météorologiques, énergétiques et commerciales. Ce n'est pas seulement un problème de prédiction, c'est plutôt un problème d'explication de prix par d'autres variables simultanées. L'objectif est de créer un modèle qui évalue le changement quotidien du prix des contrats à terme (également appelés "futures") sur l'alimentation électrique en France ou en Allemagne, à partir des conditions actuelles du marché. Les futures sont des instruments financiers qui donnent une estimation de la valeur de l'alimentation électrique au moment de la maturité du contrat à partir des conditions actuelles du marché. Les variables explicatives comprennent des mesures quotidiennes de données météorologiques, de fabrication énergétique et d'utilisation de l'alimentation électrique. Le projet sera conduit en utilisant la méthodologie CRISP-DM (Cross-Industry Standard Process for Data Mining), qui est une méthode flexible et itérative en 6 étapes, allant de la compréhension du problème commercial à la mise en service. Pour mener à bien ce projet nous allons voir les étapes d'implémentation et description des données puis la préparation des données pour pouvoir par la suite mener une analyse exploratoire sur c'est donné et d'ainsi les modélisé en créant des modèles que nous évaluerons.

I. Description des données

1) Importation des données

Dans un premier temps nous avons importé les 3 fichiers.csv contenant les données du projet :



Data_X.csv : Les données d'entrée.

Data_Y.csv : Les données en sortie.

DataNew_X(1).csv : Nouvelles données en entrée non labélisées.

Les données en

L'affichage des données dans les fichiers se présente sous la forme ci-dessous:

Data_X.csv :

	ID	DAY_ID	COUNTRY	DE_CONSUMPTION	FR_CONSUMPTION	DE_FR_EXCHANGE	FR_DE_EXCHANGE	DE_NET_EXPORT	FR_NET_EXPORT
1	1054	206	FR	0.21009872848331146	-0.42745821316578636	-0.6065234951621371	0.6065234951621371	<null>	0.6928595710148415
2	2049	501	FR	-0.022399345587708043	-1.0034524652052401	-0.022863246760841804	0.022863246760841804	-0.5735195387252847	-1.1308376214004447
3	1924	687	FR	1.39503509281117 28	1.9786653993528185	1.0213052419055586	-1.0213052419055586	-0.622021238905383	-1.6825866471237902
4	297	720	DE	-0.9833238837730477	-0.8491980420890692	-0.8395859313179942	0.8395859313179942	-0.27086984128756536	0.5632299001791123
5	1101	818	FR	0.14380737864098167	-0.617037662202979	-0.9249899894840419	0.9249899894840419	<null>	0.9903236003835739
6	1520	467	FR	-0.2952958672247945	-0.765120231829888	-0.7174900813005115	0.7174900813005115	-1.1171388109374805	-0.20030541442285069
7	1546	144	FR	-0.23696489323430939	-0.6770853480417063	0.39958804637214672	-0.39958804637214672	0.1839977013977057	-0.9862353887047263
8	1069	1136	FR	0.3322228602385505	-0.5467972255793925	-0.5077730150785651	0.5077730150785651	<null>	0.5695847995077787
9	1323	83	FR	-0.023022673135169428	-0.6046935952244551	-1.1967874807816392	1.1967874807816392	-0.5193389458680845	0.37310888239179724
10	1618	307	FR	0.5646573611019932	0.2737763682136477	-0.4390628527311798	0.4390628527311798	-0.6366538007078375	0.342056985738985 15
11	1507	277	FR	-0.11337880789441634	-0.8919656812701807	-1.138858792817 14	1.138858792817 14	-1.4559140698345308	0.9643914639062722
12	819	116	DE	-0.05569154899950751	-0.8113372121877396	0.23710513845518927	-0.23710513845518927	-0.8510823699251139	-1.0911416792551087
13	918	406	DE	0.5321156316528088	-0.3311014722882093 5	0.3399421451193713	-0.3399421451193713	-0.1731225050749824	-1.31202947257174
14	283	1175	DE	-0.32828571025909403	-1.0622551112985645	-1.3804638084739533	1.3804638084739533	-1.046122110989568	1.0022432987844765
15	158	309	DE	1.0289867658474938	1.629315058059724	1.1296634068222267	-1.1296634068222267	-0.3912607250071 6147	-1.8231174180606853
16	1821	519	FR	0.009603258955591 73	-0.9141496794269045	-1.0996672639616083	1.0996672639616083	-0.7179005303121573	1.587396239000823 6
17	1263	673	FR	0.4530755278422514 7	-0.5784023737607062	-1.4001573886975316	1.4001573886975316	-1.1827884148523053	1.1828575019651306
18	240	626	DE	0.09642208050560877	-0.6607026444904905	-2.866913326642389	2.866913326642389	-1.2523719442618615	1.8794307012588838
19	1897	184	FR	1.877137085489515	1.6706747640015012	0.42634609880 8705765	-0.42634609880 8705765	0.2824880617765816	-1.8632559305215006
20	964	804	FR	1.5857485132777741	2.6321045528930447	<null>	<null>	<null>	<null>
21	519	1060	DE	0.23163298962546	-0.4923984193812 46	-1.0051772052067576	1.0051772052067576	-0.6898083095209305	1.1153512959412968
22	1199	657	FR	1.0497224854798348	0.7296704850079904	0.3213354070714872	-0.3213354070714872	1.5322902652388204	1.2522632040457966
23	1482	461	FR	-0.3229843249753953 6	-0.8957150894092788	-1.9112340533959569	1.9112340533959569	-1.649196080110503	0.7220109687498161
24	1104	555	FR	0.6304323724535154	-0.2544405552636929	0.32413945988477816	-0.32413945988477816	1.078580563892435	0.14420198033115422
25	971	845	FR	1.0153076722561896	1.3244946539805575	0.6498225862532923	-0.6498225862532923	<null>	<null>
26	583	23	DE	-0.07863837580269416	-0.6093302279412691	-0.6656694463634136	0.6656694463634136	-1.5226303464788802	1.0061996629468474
27	1715	27	FR	0.7104369283399956	0.3020150508553640 6	0.01028117638716606	-0.01028117638716606	-0.5742926485326668	0.19710352023049116
28	2004	204	FR	0.3452057496785858	-0.7167713471489976	0.20910888400930753	-0.20910888400930753	-0.5227577687191253	-1.0368983883540055
29	2129	1184	FR	0.5241534750453108	-0.6253585666988429	0.3001637214889648	-0.3001637214889648	-0.42119137946040636	-0.4263314051305955 3
30	837	479	DE	0.17237190620151308	-0.4660754023309044 5	0.6693857454157872	-0.6693857454157872	-0.1452002950859622 4	-2.335635366327745
31	1866	667	FR	1.0396438062298374	-8.569869195460 2e-05	1.1607253562036062	-1.1607253562036062	0.0832764451477640 4	-0.3540579831775868
32	902	659	DE	0.22200397452661622	-0.8099541025424614	-0.9720502556915993	0.9720502556915993	-1.4606183700775628	-0.7425043482261339
33	1059	249	FR	0.1243307186318711	-0.4751583816518 6464	-0.9417708326767598	0.9417708326767598	<null>	1.1432367744019094
34	1659	725	FR	0.6219259671665175	2.261158916245487	1.7430336570970 27	-1.7430336570970 27	0.2673622777974 8356	-1.6919924813325 15
35	1409	25	FR	1.2303587772624 44	1.1334539264663 58	0.9440307632137033	-0.9440307632137033	0.9881230696843646	0.7143130841288053

2) Identification des données

Les données d'entrée Data X et DataNew X correspondent aux mêmes variables explicatives, mais pour deux périodes distinctes.

Pour c'est deux fichiers, les données d'entrée possèdent 35 colonnes :

Identifiant :

-ID : identifiant associant jour (DAY_ID) et pays (COUNTRY)

-DAY_ID : Identifiant des jours

-COUNTRY : Identifiant des pays

Variation journalières du prix des matières premières :

-GAS RET : Gaz en Europe

-COAL RET : Charbon en Europe

-CARBON RET : Futures émissions de carbone

Mesure météorologique :

-X_TEMP : Température

-X_RAIN : Pluie

-X_WIND : Vent

Mesure de productions d'énergie journalière dans les pays :

X_GAS : Gaz naturel

X_COAL : Charbon

X_HYDRO : Hydraulique

X_NUCLEAR : Nucléaire

X_SOLAR: Photovoltaïque

X_WINDPOW: Eolienne

X_LIGNITE : Lignite

Mesure d'utilisation électrique journalières dans les pays :

-x_CONSUMPTON : Électricité totale consommée

-x_RESIDUAL LOAD : Électricité consommée après utilisation des d'énergies renouvelables

-x_NET_IMPORT : Électricité importée depuis l'Europe

-x_NET_EXPORT : Électricité exportée vers l'Europe

-DE_FR_EXCHANGE : Electricité ´échangée entre Allemagne et France

-FR_DE_EXCHANGE : Electricité échangée entre France et Allemagne.

Concernant les données en sortie Data_Y, les données se composent en deux colonnes :

Identifiant :

-ID : identifiant identique aux données d'entrée

Variation du prix de l'électricité :

-TARGET : Variation journalière du prix de l'électricité dans le futur (maturité 24h)

3) Importation des librairie python

Après avoir importé et identifié nos données, nous avons importé toutes les librairie python qui nous serons nécéssaire pour réalisé ce projet :

```
import pandas as pd
import numpy as np
import seaborn as sns
import sklearn.cluster as skc
import sklearn.preprocessing as skp
import csv
```

-pandas: Manipuler et analyser des données de manière efficace et intuitive.

-numpy: Outils pour travailler avec des tableaux multidimensionnels (arrays) et effectuer des opérations mathématiques sur ces derniers.

-seaborn: Créer facilement des visualisations statistiques attractives et informatives à partir de données structurées.

-sklearn.cluster: Fournit des algorithmes de clustering pour regrouper des données similaires en fonction de leurs caractéristiques communes. Utile pour la segmentation de données et la détection de motifs.

-sklearn.preprocessing: Outils pour prétraiter et transformer les données en vue de les utiliser dans des modèles d'apprentissage automatique, notamment pour la normalisation, la binarisation, l'encodage et l'imputation de données manquantes.

-csv: Fournit des fonctionnalités pour lire, écrire et manipuler des fichiers au format CSV

II. Préparation des données

1) Chargement des données

Pour commencer, nous avons créé une classe : DataLoad() nous permettant de charger les données à partir des fichiers csv :

```
class DataLoad():
    """Classe permettant de charger les données"""
    def __init__(self):
        self.dfX = pd.read_csv("Data_X.csv")
        self.dfY = pd.read_csv("Data_Y.csv")
        self.dfNew_X = pd.read_csv("DataNew_X (1).csv")
        self.FR_dfX = self.dfX[self.dfX['COUNTRY'] == 'FR']
        self.FR_dfY = self.dfY[self.dfX['COUNTRY'] == 'FR']
        self.DE_dfX = self.dfX[self.dfX['COUNTRY'] == 'DE']
        self.DE_dfY = self.dfY[self.dfX['COUNTRY'] == 'DE']
```

Lorsque la classe est instanciée, les fichiers "Data_X.csv", "Data_Y.csv" et "DataNew_X (1).csv" sont lus et les données sont stockées dans les attributs dfX, dfY et dfNew_X respectivement. Quatre nouveaux DataFrame sont créés pour les données correspondantes aux pays 'FR' et 'DE' dans les DataFrames dfX et dfY, et stockés dans les attributs FR_dfX, FR_dfY, DE_dfX et DE_dfY.

2) Valeurs manquantes

Une fois nos donnée afficher, nous avons remarqué qu'il y avait des données manquante.Nous avons donc crée une méthode missing_values() pour afficher le nombre de valeurs manquantes dans cacune des DataFrame :

```
def missing_values(self):
    print(self.dfX.isnull().sum())
    print(self.dfY.isnull().sum())
    print(self.dfNew_X.isnull().sum())
```

La méthode utilise la fonction isnull() de pandas pour déterminer les valeurs manquantes et la fonction sum() pour calculer la somme de ces valeurs manquantes. Les résultats sont affichés par un print() pour chaque DataFrame.

Output de la méthode :

Pour Data_X :

```
ID                   0
DAY_ID               0
COUNTRY              0
DE_CONSUMPTION       0
FR_CONSUMPTION       0
DE_FR_EXCHANGE      25
FR_DE_EXCHANGE      25
DE_NET_EXPORT      124
FR_NET_EXPORT       70
DE_NET_IMPORT      124
FR_NET_IMPORT       70
DE_GAS               0
FR_GAS               0
DE_COAL              0
FR_COAL              0
DE_HYDRO             0
FR_HYDRO             0
DE_NUCLEAR           0
FR_NUCLEAR           0
DE_SOLAR             0
FR_SOLAR             0
DE_WINDPOW           0
FR_WINDPOW           0
DE_LIGNITE           0
DE_RESIDUAL_LOAD     0
FR_RESIDUAL_LOAD     0
DE_RAIN             94
FR_RAIN             94
DE_WIND             94
FR_WIND             94
DE_TEMP             94
FR_TEMP             94
GAS_RET              0
COAL_RET             0
CARBON_RET           0
dtype: int64
```

Pour Data_Y :

```
ID        0
TARGET    0
dtype: int64
```

Pour DataNew_X :

```
ID                   0
DAY_ID               0
COUNTRY              0
DE_CONSUMPTION       0
FR_CONSUMPTION       0
DE_FR_EXCHANGE       9
FR_DE_EXCHANGE       9
DE_NET_EXPORT       47
FR_NET_EXPORT       24
DE_NET_IMPORT       47
FR_NET_IMPORT       24
DE_GAS               0
FR_GAS               0
DE_COAL              0
FR_COAL              0
DE_HYDRO             0
FR_HYDRO             0
DE_NUCLEAR           0
FR_NUCLEAR           0
DE_SOLAR             0
FR_SOLAR             0
DE_WINDPOW           0
FR_WINDPOW           0
DE_LIGNITE           0
DE_RESIDUAL_LOAD     0
FR_RESIDUAL_LOAD     0
DE_RAIN             40
FR_RAIN             40
DE_WIND             40
FR_WIND             40
DE_TEMP             40
FR_TEMP             40
GAS_RET              0
COAL_RET             0
CARBON_RET           0
dtype: int64
```

On peux voir que toutes les variable ou il y a des valeurs manquantes on une somme différentes de 0.

3) Normalisation des données

Après avoir trouver le nombre de valeursm manquantes dans nos DataFrame, nous avons normalisé nos DataFrame à l'aide de la méthode normalisaion() :

```python
def normalisation(self):
    #Normalisation des données
    pd.options.mode.chained_assignment = None
    self.FR_dfX.drop(['DE_FR_EXCHANGE','FR_NET_EXPORT','DE_NET_EXPORT'], axis=1, inplace=True)
    self.DE_dfX.drop(['DE_FR_EXCHANGE', 'FR_NET_EXPORT', 'DE_NET_EXPORT'], axis=1, inplace=True)

    self.FR_dfX.drop(['COUNTRY', 'DAY_ID'], axis=1, inplace=True)
    self.DE_dfX.drop(['COUNTRY', 'DAY_ID'], axis=1, inplace=True)

    self.FR_dfX['TARGET'] = self.dfY['TARGET']
    self.DE_dfX['TARGET'] = self.dfY['TARGET']
    self.FR_dfX.fillna(self.FR_dfX.mean(), inplace=True)
    self.DE_dfX.fillna(self.DE_dfX.mean(), inplace=True)
```

Tout d'abord, les colonnes 'DE_FR_EXCHANGE', 'FR_NET_EXPORT' et 'DE_NET_EXPORT' sont supprimées des deux DataFrames. Ensuite, les colonnes 'COUNTRY' et 'DAY_ID' sont également supprimées. Les colonnes 'TARGET' sont ensuite ajoutées aux deux DataFrames à partir du DataFrame dfY. Enfin, les valeurs manquantes de chaque DataFrame sont remplacées par la moyenne de la colonne correspondante. Le résultat final est deux DataFrames normalisés qui peuvent être utilisés pour entamer l'analyse de données.

De plus, nous utilisons la méthode verifNA pour être sur qu'il y a aucune valeur manquante dans nos DataFrame :

```
def verifNA(self):
    print(self.FR_dfX.isnull().sum())
    print(self.DE_dfX.isnull().sum())
    print(self.FR_dfY.isnull().sum())
    print(self.DE_dfY.isnull().sum())
```

isnull() est utilisé pour détecter les valeurs manquantes et la fonction sum() pour calculer la somme de ces valeurs manquantes. Nous avons choisie d'utiliser cette méthode pour vérifier rapidement s'il y a des valeurs manquantes dans les données avant de poursuivre l'analyse des données.

Output de la méthode :

FR_dfX :

```
ID                  0
DE_CONSUMPTION      0
FR_CONSUMPTION      0
FR_DE_EXCHANGE      0
DE_NET_IMPORT       0
FR_NET_IMPORT       0
DE_GAS              0
FR_GAS              0
DE_COAL             0
FR_COAL             0
DE_HYDRO            0
FR_HYDRO            0
DE_NUCLEAR          0
FR_NUCLEAR          0
DE_SOLAR            0
FR_SOLAR            0
DE_WINDPOW          0
FR_WINDPOW          0
DE_LIGNITE          0
DE_RESIDUAL_LOAD    0
FR_RESIDUAL_LOAD    0
DE_RAIN             0
FR_RAIN             0
DE_WIND             0
FR_WIND             0
DE_TEMP             0
FR_TEMP             0
GAS_RET             0
COAL_RET            0
CARBON_RET          0
TARGET              0
dtype: int64
```

DE_dfX :

```
ID                  0
DE_CONSUMPTION      0
FR_CONSUMPTION      0
FR_DE_EXCHANGE      0
DE_NET_IMPORT       0
FR_NET_IMPORT       0
DE_GAS              0
FR_GAS              0
DE_COAL             0
FR_COAL             0
DE_HYDRO            0
FR_HYDRO            0
DE_NUCLEAR          0
FR_NUCLEAR          0
DE_SOLAR            0
FR_SOLAR            0
DE_WINDPOW          0
FR_WINDPOW          0
DE_LIGNITE          0
DE_RESIDUAL_LOAD    0
FR_RESIDUAL_LOAD    0
DE_RAIN             0
FR_RAIN             0
DE_WIND             0
FR_WIND             0
DE_TEMP             0
FR_TEMP             0
GAS_RET             0
COAL_RET            0
CARBON_RET          0
TARGET              0
dtype: int64
```

FR_dfY :

```
ID        0
TARGET    0
dtype: int64
```

DE_dfY :

```
ID        0
TARGET    0
dtype: int64
```

On voit bien que toute nos variable on une somme de 0. Se qui se traduit pas l'absence de valeur null dans nos DataFrame.

4) Comparaison des attributs

Enfin, pour finir la préparation des données, nous allons vérifier si les valeurs des colonnes des données de nos DataFrame sont compparables avec la variable cible à étudié dans ce projet (TARGET) :

```
def comparable_values(self):
    for i in self.FR_dfX.columns:
        if(self.FR_dfX[i].dtypes==self.FR_dfX['TARGET'].dtypes):
            print("Les valeurs sont comparables")
        else:
            print("Les valeurs ne sont pas comparables")
```

La méthode parcourt toutes les colonnes du DataFrame FR_dfX en utilisant une boucle for et compare le type de données de chaque colonne avec celui de la colonne TARGET en utilisant la méthode dtypes() de pandas. Si les types de données sont les mêmes, la méthode affiche "Les valeurs sont comparables", sinon elle affiche "Les valeurs ne sont pas comparables". Cette méthode est utile pour vérifier si toutes les colonnes du DataFrame FR_dfX ont des types de données appropriés pour l'analyse ou la modélisation de données.

Output de la méthode :

```
Les valeurs ne sont pas comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
Les valeurs sont comparables
```

Nous pouvons voir que toutes les variables sont comparables avec TARGET sauf ID étant donnée que ID est de type INT et non de type FLOAT.

III. Analyse exploratoire des données

1) Identification de la variable cible

Comme dit précédemment, TARGET est la variable cible de ce projet. Etant donc la variable à étudier, nous avons donc décidé de l'implémenter dans les DataFrame X d'entrée pour FR et DE. Cela nous permet de faciliter l'études de Target avec les différentes DataFrame de FR et DE.

```
def normalisation(self):
    #Normalisation des données
    pd.options.mode.chained_assignment = None
    self.FR_dfX.drop(['DE_FR_EXCHANGE','FR_NET_EXPORT','DE_NET_EXPORT'], axis=1, inplace=True)
    self.DE_dfX.drop(['DE_FR_EXCHANGE', 'FR_NET_EXPORT', 'DE_NET_EXPORT'], axis=1, inplace=True)

    self.FR_dfX.drop(['COUNTRY', 'DAY_ID'], axis=1, inplace=True)
    self.DE_dfX.drop(['COUNTRY', 'DAY_ID'], axis=1, inplace=True)

    self.FR_dfX['TARGET'] = self.dfY['TARGET']
    self.DE_dfX['TARGET'] = self.dfY['TARGET']
    self.FR_dfX.fillna(self.FR_dfX.mean(), inplace=True)
    self.DE_dfX.fillna(self.DE_dfX.mean(), inplace=True)
```

2) Aperçus et éxamination des variables

Pour commencer, nous avons utilisé la méthode info nous permettant d'obtenir les types des colonnes de nos DataFrame :

```
#Les types des colonnes
    5 usages (4 dynamic)
    def info(self):
        print(self.FR_dfX.info())
        print(self.DE_dfX.info())
        print(self.FR_dfY.info())
        print(self.DE_dfY.info())
```

La méthode info() affiche des informations sur les colonnes de chaque DataFrame, telles que le nom de la colonne, le nombre de valeurs non nulles, le type de données et la mémoire utilisée. Cela nous permet de comprendre les données et à diagnostiquer les problèmes potentiels tels que les valeurs manquantes ou les types de données incorrects.

Output de la méthode :

FR_dfX :

```
Index: 851 entries, 0 to 1492
Data columns (total 34 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   ID                851 non-null    int64
 1   DE_CONSUMPTION    851 non-null    float64
 2   FR_CONSUMPTION    851 non-null    float64
 3   DE_FR_EXCHANGE    851 non-null    float64
 4   FR_DE_EXCHANGE    851 non-null    float64
 5   DE_NET_EXPORT     851 non-null    float64
 6   FR_NET_EXPORT     851 non-null    float64
 7   DE_NET_IMPORT     851 non-null    float64
 8   FR_NET_IMPORT     851 non-null    float64
 9   DE_GAS            851 non-null    float64
 10  FR_GAS            851 non-null    float64
 11  DE_COAL           851 non-null    float64
 12  FR_COAL           851 non-null    float64
 13  DE_HYDRO          851 non-null    float64
 14  FR_HYDRO          851 non-null    float64
 15  DE_NUCLEAR        851 non-null    float64
 16  FR_NUCLEAR        851 non-null    float64
 17  DE_SOLAR          851 non-null    float64
 18  FR_SOLAR          851 non-null    float64
 19  DE_WINDPOW        851 non-null    float64
 20  FR_WINDPOW        851 non-null    float64
 21  DE_LIGNITE        851 non-null    float64
 22  DE_RESIDUAL_LOAD  851 non-null    float64
 23  FR_RESIDUAL_LOAD  851 non-null    float64
 24  DE_RAIN           851 non-null    float64
 25  FR_RAIN           851 non-null    float64
 26  DE_WIND           851 non-null    float64
 27  FR_WIND           851 non-null    float64
 28  DE_TEMP           851 non-null    float64
 29  FR_TEMP           851 non-null    float64
 30  GAS_RET           851 non-null    float64
 31  COAL_RET          851 non-null    float64
 32  CARBON_RET        851 non-null    float64
 33  TARGET            851 non-null    float64
dtypes: float64(33), int64(1)
memory usage: 232.7 KB
```

DE_dfX :

```
Index: 643 entries, 3 to 1493
Data columns (total 34 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   ID               643 non-null    int64
 1   DE_CONSUMPTION   643 non-null    float64
 2   FR_CONSUMPTION   643 non-null    float64
 3   DE_FR_EXCHANGE   643 non-null    float64
 4   FR_DE_EXCHANGE   643 non-null    float64
 5   DE_NET_EXPORT    643 non-null    float64
 6   FR_NET_EXPORT    643 non-null    float64
 7   DE_NET_IMPORT    643 non-null    float64
 8   FR_NET_IMPORT    643 non-null    float64
 9   DE_GAS           643 non-null    float64
 10  FR_GAS           643 non-null    float64
 11  DE_COAL          643 non-null    float64
 12  FR_COAL          643 non-null    float64
 13  DE_HYDRO         643 non-null    float64
 14  FR_HYDRO         643 non-null    float64
 15  DE_NUCLEAR       643 non-null    float64
 16  FR_NUCLEAR       643 non-null    float64
 17  DE_SOLAR         643 non-null    float64
 18  FR_SOLAR         643 non-null    float64
 19  DE_WINDPOW       643 non-null    float64
 20  FR_WINDPOW       643 non-null    float64
 21  DE_LIGNITE       643 non-null    float64
 22  DE_RESIDUAL_LOAD 643 non-null    float64
 23  FR_RESIDUAL_LOAD 643 non-null    float64
 24  DE_RAIN          643 non-null    float64
 25  FR_RAIN          643 non-null    float64
 26  DE_WIND          643 non-null    float64
 27  FR_WIND          643 non-null    float64
 28  DE_TEMP          643 non-null    float64
 29  FR_TEMP          643 non-null    float64
 30  GAS_RET          643 non-null    float64
 31  COAL_RET         643 non-null    float64
 32  CARBON_RET       643 non-null    float64
 33  TARGET           643 non-null    float64
dtypes: float64(33), int64(1)
memory usage: 175.8 KB
```

FR_dfY :

```
Index: 851 entries, 0 to 1492
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   ID      851 non-null    int64
 1   TARGET  851 non-null    float64
dtypes: float64(1), int64(1)
memory usage: 19.9 KB
```

DE_dfY :

```
Index: 643 entries, 3 to 1493
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   ID      643 non-null    int64
 1   TARGET  643 non-null    float64
dtypes: float64(1), int64(1)
memory usage: 15.1 KB
```

Nous pouvons bien voir le type chaque variable dans les Data Frame étant tous des FLOAT mis à part ID qui est un INT.

Ensuite nous affichons les distributions et les plages de valeurs des variables de nos DataFrame grâce à la méthode describe :

```
#Les distributions des colonnes, les moyennes, les écarts-types, les min/max et les quartiles
    8 usages (7 dynamic)
    def describe(self):
        print(self.FR_dfX.describe())
        print(self.DE_dfX.describe())
        print(self.FR_dfY.describe())
        print(self.DE_dfY.describe())
```

La méthode describe() calcule des statistiques de base pour chaque colonne de chaque DataFrame, telles que le nombre d'observations, la moyenne, l'écart type, le minimum et le maximum. Cela nous aide à comprendre la distribution des données et à identifier les valeurs aberrantes ou les erreurs potentielles.

Output de la méthode :

FR_dfX :

	ID	DE_CONSUMPTION	...	CARBON_RET	TARGET
count	851.000000	851.000000	...	851.000000	851.000000
mean	1532.453584	0.463432	...	0.074805	0.046026
std	352.601046	0.664918	...	1.096573	1.023512
min	933.000000	-2.265563	...	-4.281790	-6.519268
25%	1222.500000	0.027082	...	-0.530523	-0.178023
50%	1525.000000	0.410629	...	0.054056	-0.003619
75%	1839.500000	0.967994	...	0.633048	0.174344
max	2146.000000	2.033851	...	5.471818	7.786578

[8 rows x 31 columns]

DE_dfX :

	ID	DE_CONSUMPTION	...	CARBON_RET	TARGET
count	643.000000	643.000000	...	643.000000	643.000000
mean	464.360809	0.379809	...	0.088062	0.148044
std	269.765446	0.682092	...	1.102142	1.047022
min	0.000000	-2.265563	...	-4.281790	-3.075929
25%	234.000000	-0.069473	...	-0.506085	-0.324693
50%	468.000000	0.288112	...	0.054056	0.005057
75%	690.500000	0.895751	...	0.580153	0.386596
max	930.000000	2.033851	...	5.471818	7.138604

[8 rows x 31 columns]

FR_dfY :

	ID	TARGET
count	851.000000	851.000000
mean	1532.453584	0.046026
std	352.601046	1.023512
min	933.000000	-6.519268
25%	1222.500000	-0.178023
50%	1525.000000	-0.003619
75%	1839.500000	0.174344
max	2146.000000	7.786578

DE_dfY :

	ID	TARGET
count	643.000000	643.000000
mean	464.360809	0.148044
std	269.765446	1.047022
min	0.000000	-3.075929
25%	234.000000	-0.324693
50%	468.000000	0.005057
75%	690.500000	0.386596
max	930.000000	7.138604

Nous pouvons voir ici les différentes informations statistique fournit :count(Nombres de valeurs), mean(Moyenne), std(écart type), min(minimum), 25%(1er quartile), 50%(2ème quartile ou médiane), 75%(3ème quartile), max(maximum).

On remarque que pour **FR_dfX** et **DE_dfX,** la taille des tableaux nous est donnée avec (…) étant donnée le terminal ne peux pas tout nous afficher en 1 seule fois.

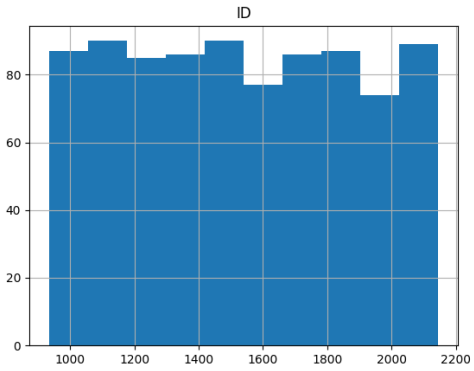
3) Les Histogramme

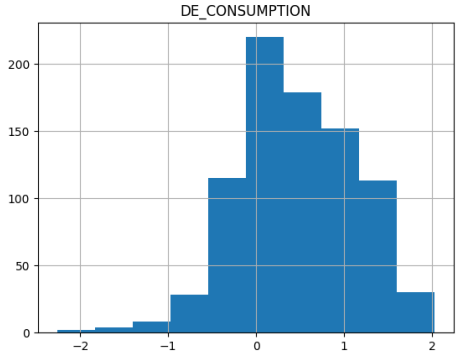
Maintenant, nous allons afficher les histogrammes de chaque variable deb nos DataFrame :

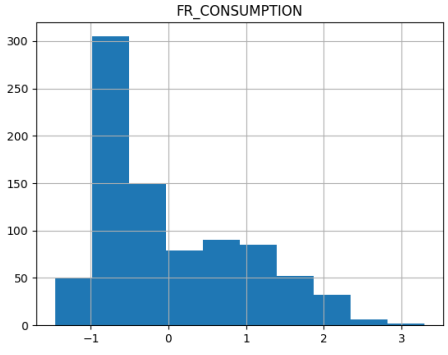
```
def showHisto(self):
    """
    Affiche les histogrammes pour chaqune des colones
    """
    for i in self.FR_dfX.columns:
        self.FR_dfX[i].hist()
        plt.title(i)
        plt.show()
    for i in self.DE_dfX.columns:
        self.DE_dfX[i].hist()
        plt.title(i)
        plt.show()
    for i in self.FR_dfY.columns:
        self.FR_dfY[i].hist()
        plt.title(i)
        plt.show()
    for i in self.DE_dfY.columns:
        self.DE_dfY[i].hist()
        plt.title(i)
        plt.show()
```

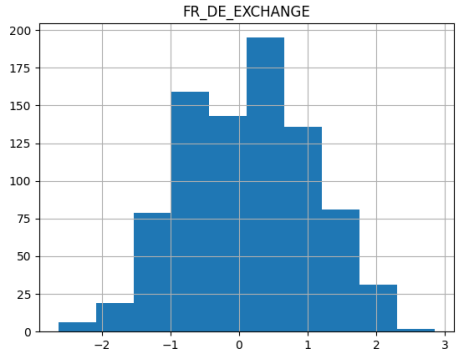
A l'aide de la fonction hist() de la bibliothèque matplotlib. La méthode itère à travers chaque colonne des DataFrames et crée un histogramme pour chaque colonne, puis affiche le titre de la colonne. show() est appelée à la fin de chaque boucle pour afficher chaque histogramme des DataFrame.

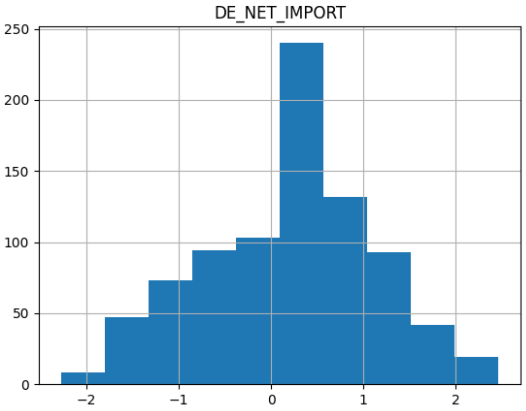
Output de la méthode :

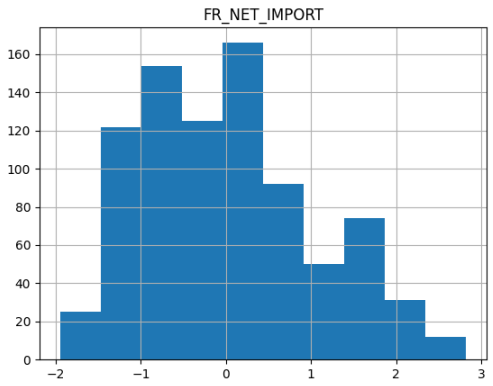


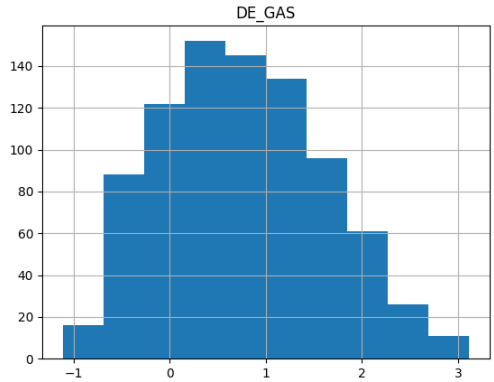


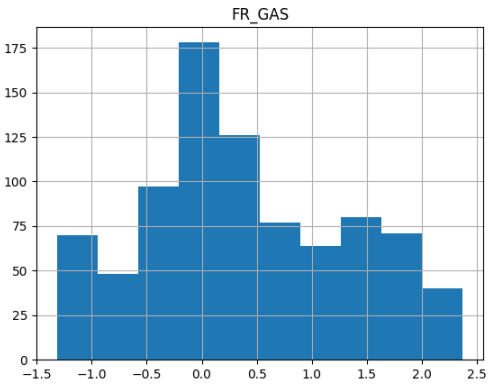


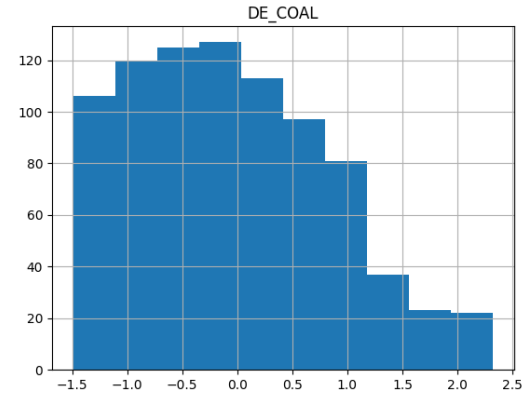


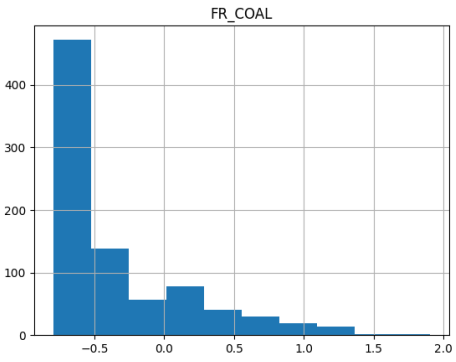

DE_NET_IMPORT

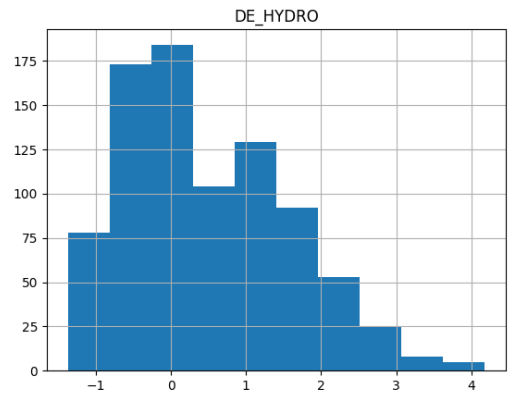

FR_NET_IMPORT

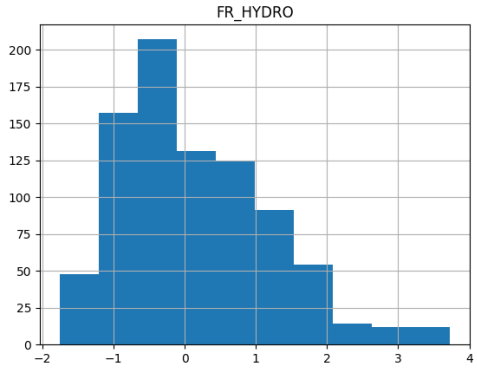

DE_GAS

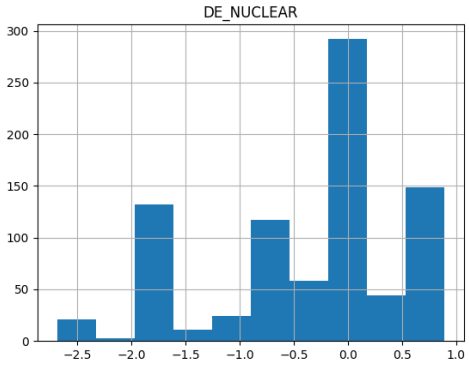

FR_GAS

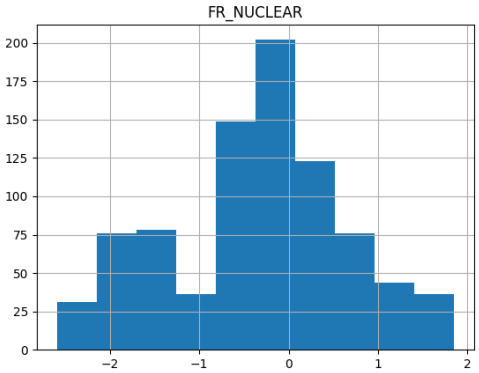

DE_COAL

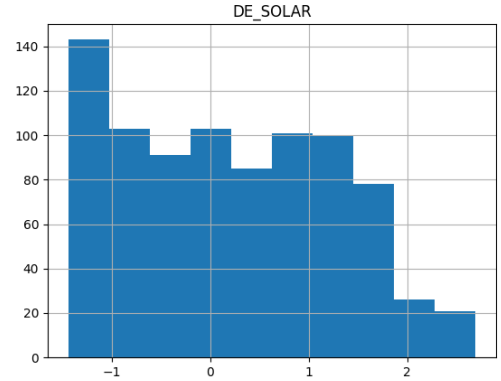

FR_COAL

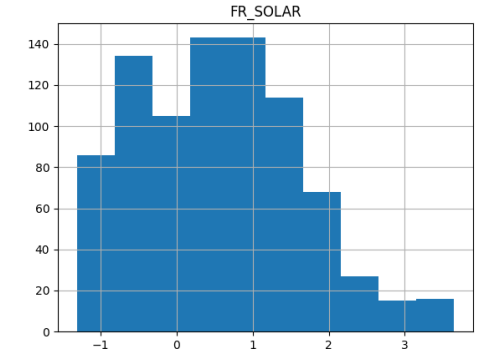

DE_HYDRO

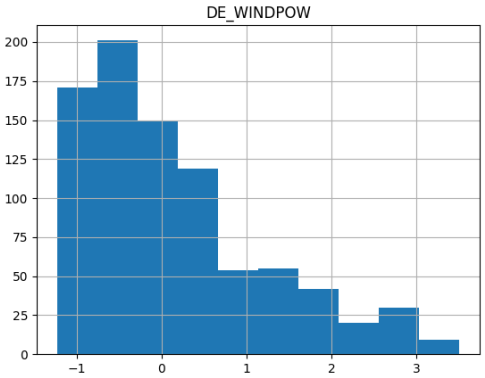

FR_HYDRO

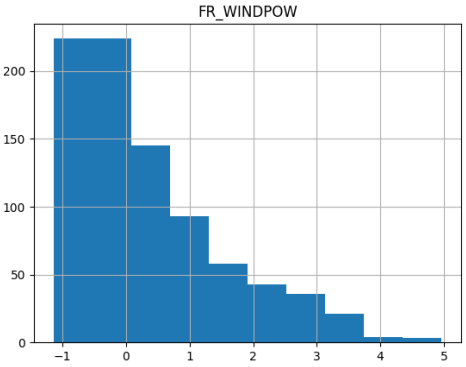

DE_NUCLEAR

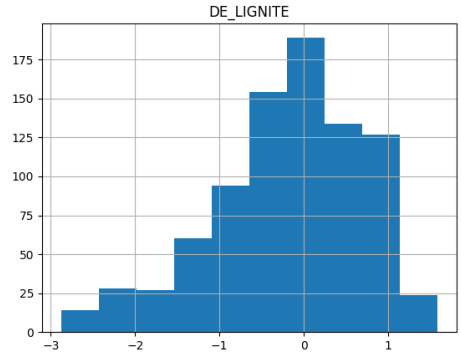

FR_NUCLEAR

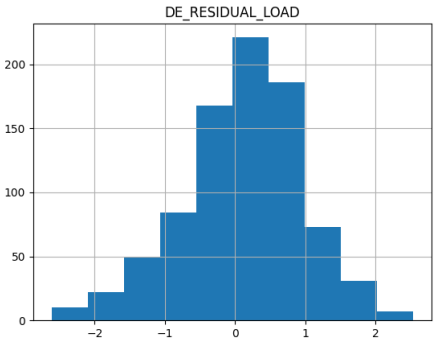

DE_SOLAR

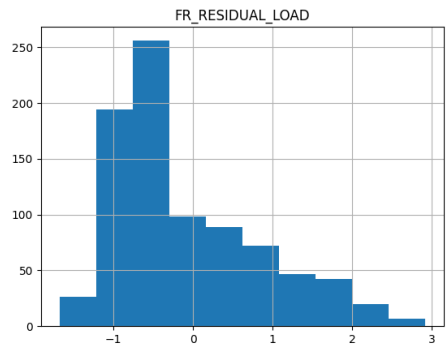

FR_SOLAR

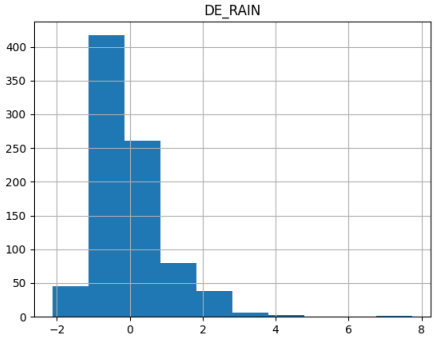

DE_WINDPOW

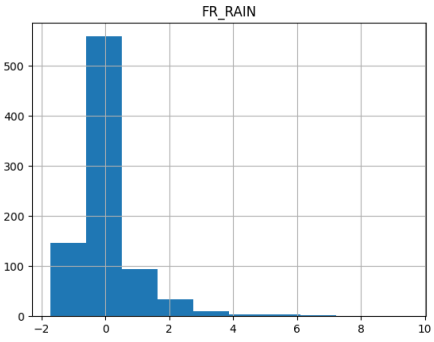

FR_WINDPOW

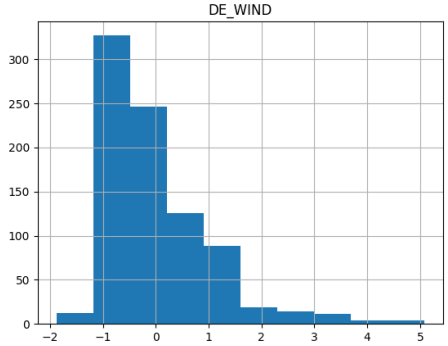

DE_LIGNITE

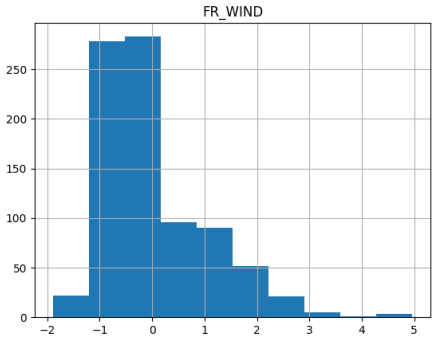

DE_RESIDUAL_LOAD

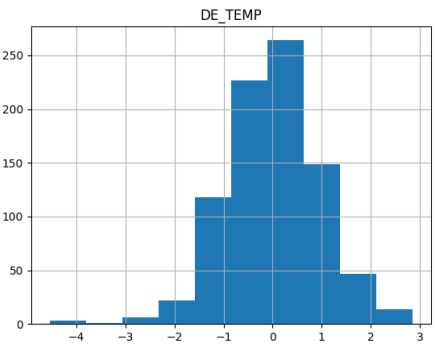

FR_RESIDUAL_LOAD

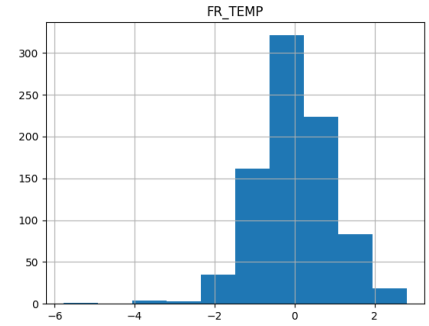

DE_RAIN

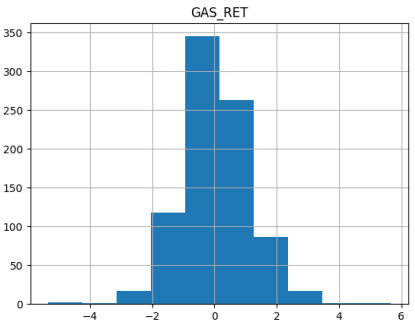

FR_RAIN

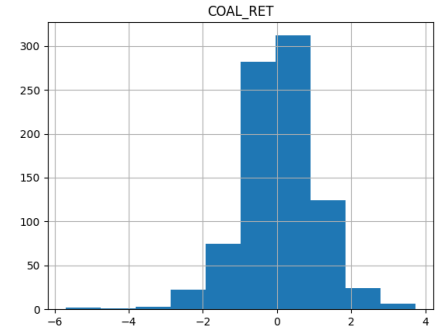

DE_WIND

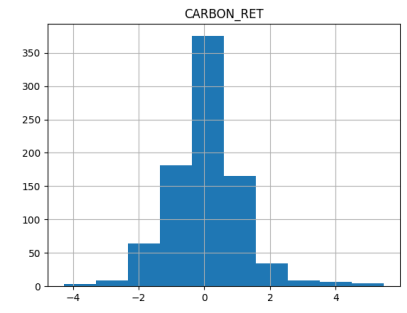

FR_WIND


DE_TEMP


FR_TEMP


GAS_RET


COAL_RET


CARBON_RET

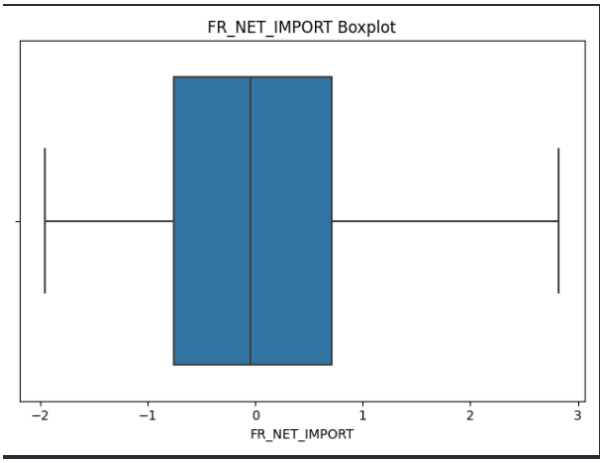
Une fois les histogramme afficher, on peut s'intéréssé a une autre manière de représenté les variables de nos DataFrame qui sont les Boxplot. Nous avons choisit de coder 2 méthode pour les Boxplot : Une pour afficher les DataFrame de FR et l'autre pour les DataFrame de DE.

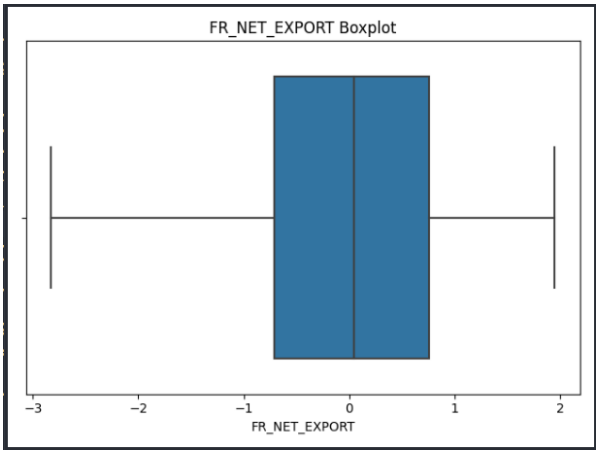
Pour FR :

```
def boxplot(self):
    for i,j in enumerate(self.FR_dfX.describe().columns):
        #plt.subplot(3, 3, i+1)
        sns.boxplot(x=self.FR_dfX[j])
        plt.title('{} Boxplot'.format(j))
        plt.tight_layout()
        plt.show()
```

Cette méthode crée des boxplots pour chacune des colonnes du DataFrame **self.FR_dfX** en utilisant la bibliothèque Seaborn. Les boxplots permettent de visualiser la distribution des données en affichant des quartiles, des valeurs aberrantes et la médiane.

Output de la méthode :



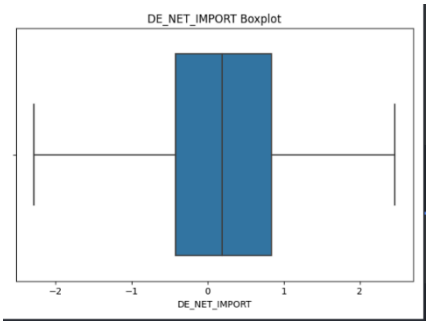


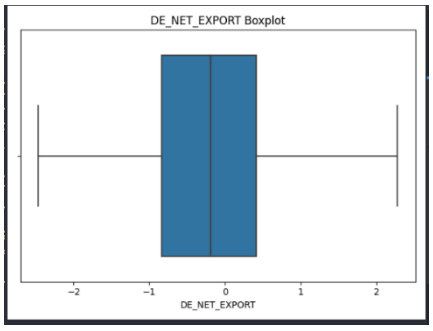
Pour DE :

```
def DEBoxplot(self):
    for i, j in enumerate(self.DE_dfX.describe().columns):
        # plt.subplot(3, 3, i+1)
        sns.boxplot(x=self.DE_dfX[j])
        plt.title('{} Boxplot'.format(j))
        plt.tight_layout()
        plt.show()
        print("success")
```

Cette méthode et la même que boxplot, nous avons juste changer modifié le DataFrame pour l'appliqué cette fois-ci à DE.

Output de la méthode :





Nous avons décidé de ne pas afficher tout les Boxplot pour FR_dfX et DE_dfX car il fournissent moins d'information nécéssaire que les histogrammes.

4) Matrice des corrélations

Ici, nous avons crée une matrice des corrélations afin de connêtre les corrélations entre TARGET et les autres variables des DataFrame FR_dfX et DE_dfX. Pour cela, nous utilisons la méthode showCorrelation() :

```
def showCorrelation(self):
    dfco= self.FR_dfX.select_dtypes(exclude=['object'])
    dfco2 = self.DE_dfX.select_dtypes(exclude=['object'])
    print(dfco.corr())
    print(dfco2.corr())
```

La méthode calcule la matrice de corrélation entre les variables numériques des DataFrames FR_dfX et DE_dfX. Il exclut les variables de type "object" (variables catégorielles) de ces deux DataFrames avant de calculer la matrice de corrélation pour éviter les erreurs de calcul. Les matrice sont affiché à l'aide du print().

Output de la méthode :

Matrice pour FR :

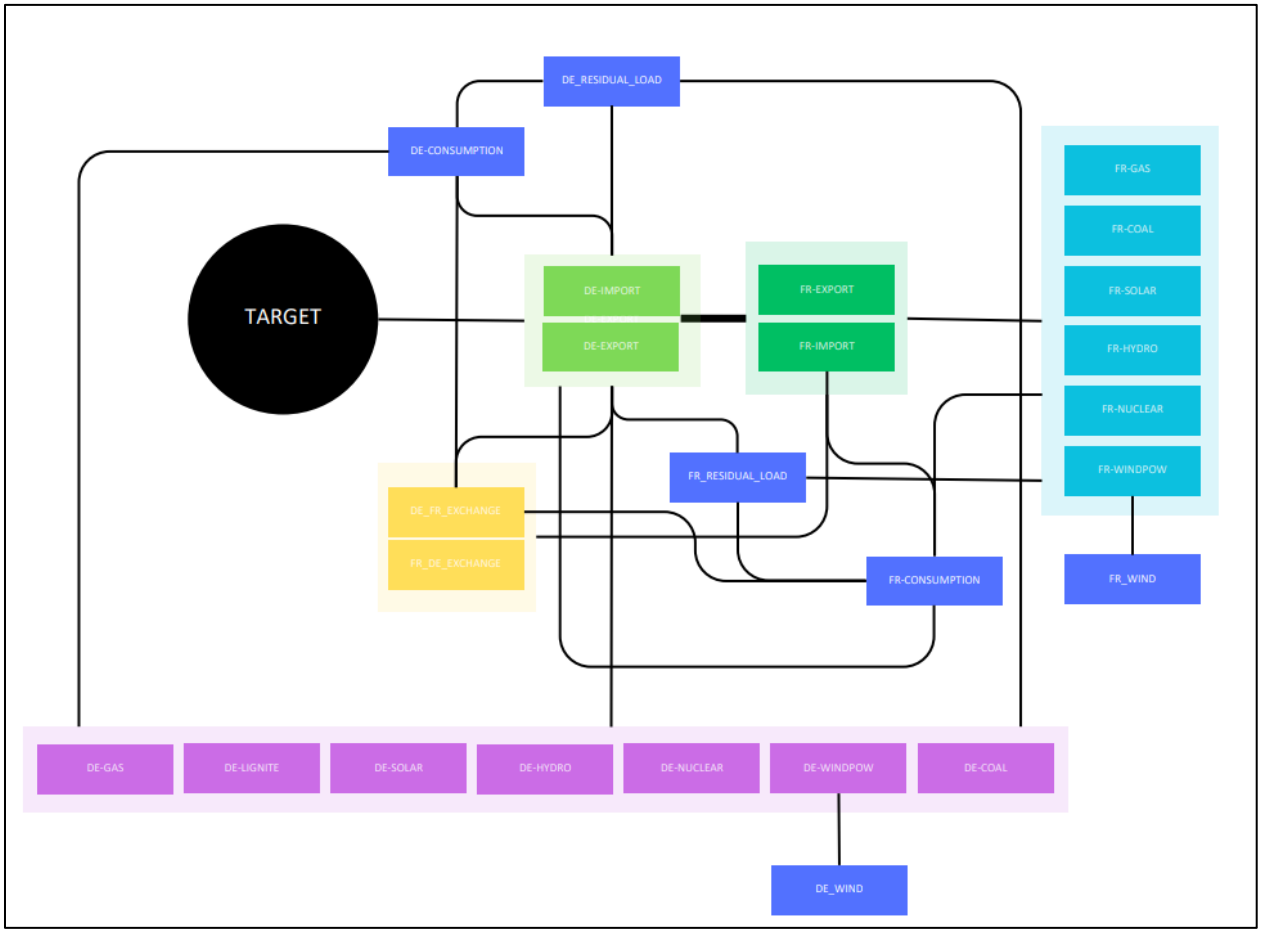
	ID	DE_CONSUMPTION	...	CARBON_RET	TARGET
ID	1.000000	-0.122745	...	0.030683	-0.010347
DE_CONSUMPTION	-0.122745	1.000000	...	-0.009315	-0.048551
FR_CONSUMPTION	-0.190134	0.817513	...	-0.055423	0.009998
DE_FR_EXCHANGE	0.223102	0.423781	...	-0.017042	0.034180
FR_DE_EXCHANGE	-0.223102	-0.423781	...	0.017042	-0.034180
DE_NET_EXPORT	-0.119942	0.527090	...	-0.087281	-0.067811
FR_NET_EXPORT	-0.465643	-0.122164	...	-0.026648	-0.042689
DE_NET_IMPORT	0.119942	-0.527090	...	0.087281	0.067811
FR_NET_IMPORT	0.465643	0.122164	...	0.026648	0.042689
DE_GAS	0.002465	0.249686	...	0.076407	0.031626
FR_GAS	0.071332	0.631322	...	-0.024409	0.007433
DE_COAL	-0.149092	0.454099	...	0.041833	0.021014
FR_COAL	-0.191322	0.511505	...	0.004457	0.031529
DE_HYDRO	0.009178	-0.363750	...	0.070531	0.052827
FR_HYDRO	-0.369836	0.414153	...	-0.050332	0.070002
DE_NUCLEAR	-0.700959	0.283447	...	0.017768	0.037859
FR_NUCLEAR	-0.592205	0.627447	...	-0.040680	0.007304
DE_SOLAR	0.164157	-0.648407	...	0.012136	0.029904
FR_SOLAR	0.359833	-0.589741	...	0.012084	0.013828
DE_WINDPOW	0.056017	0.460286	...	-0.105270	-0.073254
FR_WINDPOW	0.129825	0.371494	...	-0.062313	-0.107403
DE_LIGNITE	-0.309844	0.357043	...	0.051627	0.005673
DE_RESIDUAL_LOAD	-0.211151	0.235448	...	0.104482	0.048185
FR_RESIDUAL_LOAD	-0.263166	0.764470	...	-0.040585	0.038176
DE_RAIN	-0.023630	0.124751	...	0.063732	-0.066065
FR_RAIN	-0.019908	0.060211	...	0.055235	-0.039813
DE_WIND	0.654784	0.150052	...	0.003758	-0.045335
FR_WIND	0.678781	0.065927	...	0.031636	-0.032737
DE_TEMP	0.046404	0.020897	...	0.035049	-0.042812
FR_TEMP	0.094109	-0.140279	...	-0.017343	-0.032827
GAS_RET	0.066085	-0.053785	...	0.435978	0.057536
COAL_RET	-0.025066	0.046824	...	0.322401	-0.002330
CARBON_RET	0.030683	-0.009315	...	1.000000	0.064776
TARGET	-0.010347	-0.048551	...	0.064776	1.000000

Matrice pour DE :

	ID	DE_CONSUMPTION	...	CARBON_RET	TARGET
ID	1.000000	0.065917	...	0.032089	-0.015254
DE_CONSUMPTION	0.065917	1.000000	...	0.030435	-0.051182
FR_CONSUMPTION	-0.032720	0.807458	...	-0.021087	-0.019582
DE_FR_EXCHANGE	0.454821	0.408951	...	-0.005872	-0.084194
FR_DE_EXCHANGE	-0.454821	-0.408951	...	0.005872	0.084194
DE_NET_EXPORT	0.079323	0.521918	...	-0.057404	-0.250616
FR_NET_EXPORT	-0.564410	-0.138751	...	-0.020565	0.001544
DE_NET_IMPORT	-0.079323	-0.521918	...	0.057404	0.250616
FR_NET_IMPORT	0.564410	0.138751	...	0.020565	-0.001544
DE_GAS	-0.171043	0.218463	...	0.097121	0.177561
FR_GAS	0.113821	0.636512	...	-0.006712	0.048483
DE_COAL	0.341462	0.453990	...	0.067699	0.097803
FR_COAL	0.165405	0.537287	...	0.040573	0.004934
DE_HYDRO	-0.268177	-0.363369	...	0.069217	0.123676
FR_HYDRO	-0.307214	0.363737	...	-0.035065	0.038172
DE_NUCLEAR	-0.666844	0.171554	...	0.045496	-0.006729
FR_NUCLEAR	-0.458701	0.577100	...	-0.006897	0.009855
DE_SOLAR	0.092331	-0.629401	...	-0.013889	0.007162
FR_SOLAR	0.291754	-0.552846	...	0.002730	0.022717
DE_WINDPOW	0.049938	0.499109	...	-0.081380	-0.250204
FR_WINDPOW	0.052644	0.409527	...	-0.051674	-0.168554
DE_LIGNITE	0.169911	0.318878	...	0.073441	0.095619
DE_RESIDUAL_LOAD	-0.078528	0.189010	...	0.112265	0.250898
FR_RESIDUAL_LOAD	-0.081118	0.739510	...	-0.007027	0.029686
DE_RAIN	-0.037453	0.131831	...	0.083207	-0.005246
FR_RAIN	0.007414	0.054356	...	0.041185	-0.039429
DE_WIND	0.638564	0.285461	...	0.013900	-0.139548
FR_WIND	0.673225	0.186874	...	0.036499	-0.081716
DE_TEMP	0.077865	0.134466	...	0.006144	-0.043109
FR_TEMP	0.127300	-0.093784	...	-0.047991	-0.062655
GAS_RET	0.031087	-0.004098	...	0.425886	0.020087
COAL_RET	0.003830	0.057806	...	0.300706	-0.019866
CARBON_RET	0.032089	0.030435	...	1.000000	0.003538
TARGET	-0.015254	-0.051182	...	0.003538	1.000000

Ci-dessous les corrélation des variables avec la variable cible TARGET pour les DataFrame FR_dfX et DE_dfX.

5) Interprétation des résultats



Pour interpréter les résultats, nous avons choisie de représenté cela sous la forme d'un organigramme. Dans cette organigramme, **TARGET** et nôtre variable cible. En regardant, nos différentes corrélations avec **TARGET**, nous avons sélectionner les corrélations les plus importantes que nous représentons ci-dessous.

Comme nous pouvons le voir, **TARGET** est le plus corrélé avec **DE-IMPORT** et **DE-EXPORT**. De plus, nous pouvons remarquer d'autre corrélation secondaire qui reste corrélé avec TARGET comme FR-EXPORT, FR-IMPORT, DE-RESIDUAL_LOAD, DE_CONSUMPTION, FR-RESIDUAL_LOAD, FR_CONSUMPTION, DE_FR_EXCHANGE, FR_DE_EXCHANGE…

Les liaisons faites avec entre chaque variable représentent le niveau de corrélation entre elle.

Par exemple si l'on prend **TARGET** et **DE_IMPORT**, on peut voir que la liaison entre ces deux variables est directe ce qui montre une très forte corrélation. Tandis que si l'on prend **FR_IMPORT**, avec TARGET, on peut voir que celui-ci n'est pas en lien directement avec notre variable cible. Ce qui nous montre que la corrélation entre c'est deux variable et plus faible que la précédente. Ce qui est logique car en effet, la France est l'un des pays les plus avancés en terme de nucléaire et donc n'a pas forcément de grand besoin d'importé de l'électricité contrairement à l'Allemagne qui utilise beaucoup moins le nucléaire et qui a donc besoin d'importer plus d'électricité.

IV. Modélisation des données

1) Régression linéaire simple

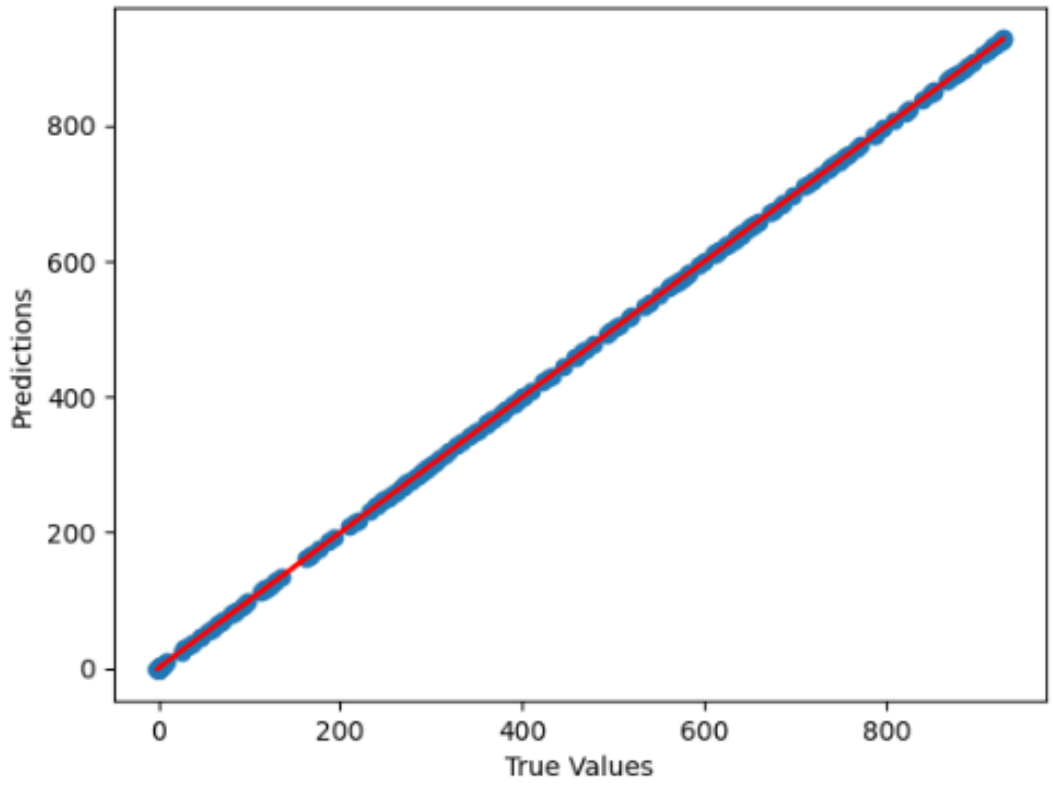
Nous allons effectué une régression linéaire sur les données de DE et FR. Pour cela, nous avons choisit de crée une fonction qui fait une régrétion linaire sur FR et une autre sur DE :

Pour DE :

```
def regressionlinearDE(self):
    DEX_train, DEX_test, DEy_train, DEy_test = train_test_split(self.DE_dfX, self.DE_dfY, test_size=0.3,random_state=42)
    model2 = LinearRegression()
    model2.fit(DEX_train, DEy_train)
    DEy_pred = model2.predict(DEX_test)
    plt.scatter(DEy_test, DEy_pred)
    plt.plot(DEy_test, DEy_test, color='red')
    plt.xlabel("True Values")
    plt.ylabel("Predictions")
    plt.show()
```

Régression linéaire sur les données DE_dfX et DE_dfY. La fonction train_test_split de Scikit-learn pour diviser les données en ensemble d'entraînement et ensemble de test avec un ratio de 0,3. Ensuite, il crée une instance de l'algorithme de régression linéaire de Scikit-learn et l'entraîne sur l'ensemble d'entraînement. Il prédit ensuite les valeurs de la variable cible sur l'ensemble de test en utilisant la méthode predict de l'instance du modèle de régression linéaire. Enfin, il affiche un nuage de points pour comparer les valeurs prédites aux valeurs réelles, ainsi qu'une ligne rouge qui représente la relation linéaire idéale entre les deux.

Output de la méthode :

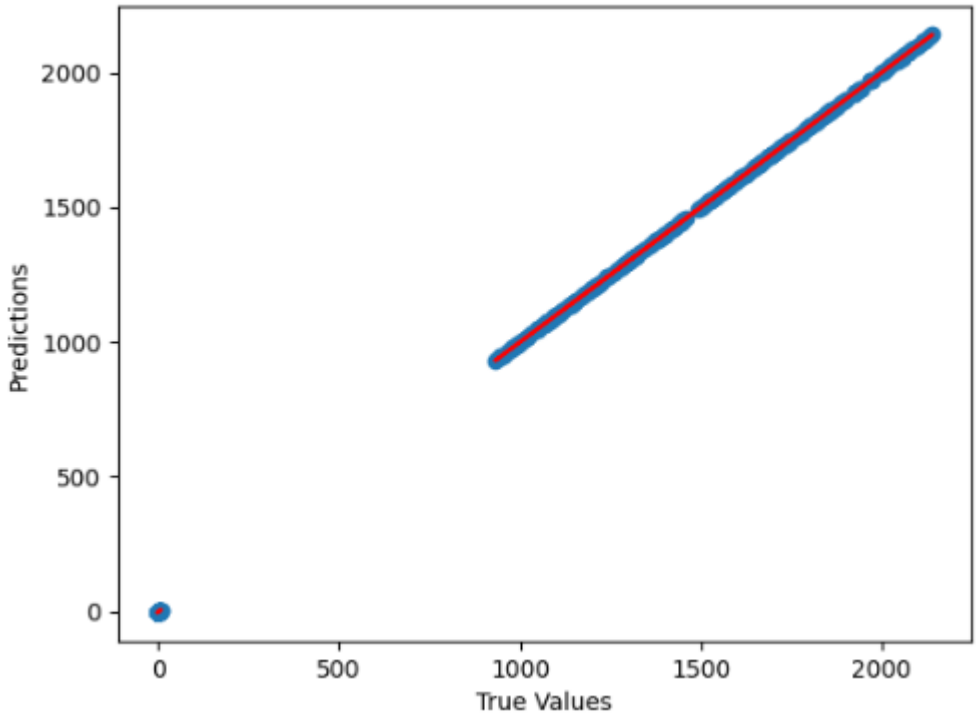


Pour FR :

```
def regressionlinearFR(self):
    FRX_train, FRX_test, FRy_train, FRy_test = train_test_split(self.FR_dfX, self.FR_dfY, test_size=0.3,
                                                                random_state=42)
    model = LinearRegression()
    model.fit(FRX_train, FRy_train)
    FRy_pred = model.predict(FRX_test)
    plt.scatter(FRy_test, FRy_pred)
    plt.plot(FRy_test, FRy_test, color='red')
    plt.xlabel("True Values")
    plt.ylabel("Predictions")
    plt.show()
```

Même fonctionnement que la méthode regressionlinearDE.

Output de la méthode :



Puis on a réalisé les autres modèles de modélisation à part le KNN afin de pouvoir modéliser nos données.

```
1 usage
def RIDGEregression(self):
    ridge = Ridge()
    parameters = {'alpha': [1e-15, 1e-10, 1e-8, 1e-3, 1e-2, 1, 5, 10, 20, 30, 35, 40, 45, 50, 55, 100]}
    ridge_regressor = GridSearchCV(ridge, parameters, scoring='neg_mean_squared_error', cv=5)
    ridge_regressor.fit(self.DE_dfX, self.DE_dfY)
    DEX_train, DEX_test, DEy_train, DEy_test = train_test_split(self.DE_dfX, self.DE_dfY, test_size=0.3,random_state=42)
    predictions = ridge_regressor.predict(DEX_test)
    print(predictions)
    sns.displot(DEy_test - predictions)
    plt.show()
    print(ridge_regressor.best_params_)
    print(ridge_regressor.best_score_)
    rmse = np.sqrt(mean_squared_error(DEy_test, predictions))
    print("rmse",rmse)
    print("R2 score : %.2f" % r2_score(DEy_test, predictions))
    res= stats.spearmanr(DEy_test, predictions)
    print(res)
```

```
def LASSOregression(self):
    lasso = Lasso()
    parameters = {'alpha': [1e-15, 1e-10, 1e-8, 1e-3, 1e-2, 1, 5, 10, 20, 30, 35, 40, 45, 50, 55, 100]}
    lasso_regressor = GridSearchCV(lasso, parameters, scoring='neg_mean_squared_error', cv=5)
    lasso_regressor.fit(self.DE_dfX, self.DE_dfY)
    DEX_train, DEX_test, DEy_train, DEy_test = train_test_split(self.DE_dfX, self.DE_dfY, test_size=0.3,random_state=42)
    predictions = lasso_regressor.predict(DEX_test)
    print(predictions)
    sns.displot(DEy_test - predictions)
    plt.show()
    print(lasso_regressor.best_params_)
    print(lasso_regressor.best_score_)
    rmse = np.sqrt(mean_squared_error(DEy_test, predictions))
    print("rmse",rmse)
    print("R2 score : %.2f" % r2_score(DEy_test, predictions))
    res= stats.spearmanr(DEy_test, predictions)
    print(res)
```

```
def DecisionTreeregression(self):
    regressor = DecisionTreeRegressor()
    regressor.fit(self.DE_dfX, self.DE_dfY)
    DEX_train, DEX_test, DEy_train, DEy_test = train_test_split(self.DE_dfX, self.DE_dfY, test_size=0.3,random_state=42)
    predictions = regressor.predict(DEX_test)
    print(predictions)
    sns.displot(DEy_test - predictions)
    plt.show()
    print(regressor.best_params_)
    print(regressor.best_score_)
    rmse = np.sqrt(mean_squared_error(DEy_test, predictions))
    print("rmse",rmse)
    print("R2 score : %.2f" % r2_score(DEy_test, predictions))
    res= stats.spearmanr(DEy_test, predictions)
    print(res)

    plt.show()
```

```
def RandomForestregression(self):
    regressor = RandomForestRegressor()
    regressor.fit(self.DE_dfX, self.DE_dfY)
    DEX_train, DEX_test, DEy_train, DEy_test = train_test_split(self.DE_dfX, self.DE_dfY, test_size=0.3,random_state=42)
    predictions = regressor.predict(DEX_test)
    print(predictions)
    sns.displot(DEy_test - predictions)
    plt.show()
    print(regressor.best_params_)
    print(regressor.best_score_)
    rmse = np.sqrt(mean_squared_error(DEy_test, predictions))
    print("rmse",rmse)
    print("R2 score : %.2f" % r2_score(DEy_test, predictions))
    res= stats.spearmanr(DEy_test, predictions)
    print(res)
```

La régression ridge ajoute une pénalité L2 à la fonction de coût de la régression linéaire. Cette pénalité est égale à la somme des carrés des coefficients de la régression. Cela signifie que les coefficients sont régularisés en étant contraints à de plus petites valeurs, ce qui peut réduire le surajustement (overfitting) du modèle. La force de la régularisation est contrôlée par un hyperparamètre λ, qui doit être choisi à l'aide d'une validation croisée.

La régression Lasso ajoute une pénalité L1 à la fonction de coût de la régression linéaire. Cette pénalité est égale à la somme des valeurs absolues des coefficients de la régression. Cela signifie que certains coefficients peuvent être mis à zéro, ce qui peut réduire la complexité du modèle et

améliorer sa généralisation. La force de la régularisation est contrôlée par l'hyperparamètre λ, qui doit être choisi à l'aide d'une validation croisée.

La méthode des forêts aléatoires (ou Random Forest en anglais) est un algorithme d'apprentissage automatique supervisé qui combine plusieurs arbres de décision pour améliorer la précision et la stabilité des prédictions. Elle est utilisée pour des tâches de classification ou de régression.

Le fonctionnement des forêts aléatoires consiste à créer un grand nombre d'arbres de décision indépendants, chacun étant entraîné sur un sous-ensemble aléatoire des données d'entraînement et des variables d'entrée. Cela permet de réduire les effets du surapprentissage (overfitting) et d'améliorer la capacité de généralisation du modèle.

Lors de la prédiction, chaque arbre de décision donne une prédiction pour la variable cible, et la prédiction finale est obtenue en moyennant (pour une tâche de régression) ou en effectuant un vote majoritaire (pour une tâche de classification) sur les prédictions de tous les arbres.

La méthode des forêts aléatoires présente plusieurs avantages : elle est robuste aux valeurs aberrantes, peut gérer de grandes quantités de données et de variables d'entrée, et fournit des informations sur l'importance des variables pour la prédiction.

Cependant, elle peut être moins interprétable que d'autres modèles d'apprentissage automatique, et peut nécessiter une optimisation des hyperparamètres pour obtenir les meilleures performances.

V. Evaluation des modèles

1) Corrélation de Spearman, R2 et RMSE

Nous avons introduit la corrélation de Spearman (une mesure non paramétrique) pour savoir si on avait une monotonie entre nôtre Data set de teste et notre prédiction.

Ensuite nous avons calculer le R2_score qui nous permet de savoir si la droite de régression colle à l'ensemble des points du data frame.

Enfin nous avons calculé le RMSE qui est l'erreur quadratique moyenne qui permet de fournir une indication sur la dispersion ou la variabilité de la qualité de la prédiction. Il est relié à la variance du modèle.

```
def RIDGEregressionFR(self):
    ridge = Ridge()
    parameters = {'alpha': [1e-15, 1e-10, 1e-8, 1e-3, 1e-2, 1, 5, 10, 20, 30, 35, 40, 45, 50, 55, 100]}
    ridge_regressor = GridSearchCV(ridge, parameters, scoring='neg_mean_squared_error', cv=5)
    ridge_regressor.fit(self.FR_dfX, self.FR_dfY)
    FRX_train, FRX_test, FRy_train, FRy_test = train_test_split(self.FR_dfX, self.FR_dfY, test_size=0.3,random_state=42)
    predictions = ridge_regressor.predict(FRX_test)
    print(predictions)
    sns.displot(FRy_test - predictions)
    plt.show()
    print(ridge_regressor.best_params_)
    print(ridge_regressor.best_score_)
    rmse = np.sqrt(mean_squared_error(FRy_test, predictions))
    print(rmse)
    print("R2 score : %.2f" % r2_score(FRy_test, predictions))
    res= stats.spearmanr(FRy_test, predictions)
    print(res)
```

Output de la méthode des méthodes avec Spearman :

```
RMSE
1.3623453024795026e-13
R2
1.0
SignificanceResult(statistic=array([[1.        , 0.04447045, 1.        , 0.04447045],
       [0.04447045, 1.        , 0.04447045, 1.        ],
       [1.        , 0.04447045, 1.        , 0.04447045],
       [0.04447045, 1.        , 0.04447045, 1.        ]]), pvalue=array([[0.        , 0.53915333, 0.        , 0.53915333],
       [0.53915333, 0.        , 0.53915333, 0.        ],
       [0.        , 0.53915333, 0.        , 0.53915333],
       [0.53915333, 0.        , 0.53915333, 0.        ]]))
RMSE
1.1413377797527236e-13
R2
1.0
SignificanceResult(statistic=array([[1.        , 0.00290649, 1.        , 0.00290112],
       [0.00290649, 1.        , 0.00290649, 0.99999982],
       [1.        , 0.00290649, 1.        , 0.00290112],
       [0.00290112, 0.99999982, 0.00290112, 1.        ]]), pvalue=array([[0.        , 0.96309003, 0.        , 0.96315811],
       [0.96309003, 0.        , 0.96309003, 0.        ],
       [0.        , 0.96309003, 0.        , 0.96315811],
       [0.96315811, 0.        , 0.96315811, 0.        ]]))
```

On a ensuite calculé les meilleurs paramètres pour chaque modèle

```
1 usage
def RIDGEregression(self):
    ridge = Ridge()
    parameters = {'alpha': [1e-15, 1e-10, 1e-8, 1e-3, 1e-2, 1, 5, 10, 20, 30, 35, 40, 45, 50, 55, 100]}
    ridge_regressor = GridSearchCV(ridge, parameters, scoring='neg_mean_squared_error', cv=5)
    ridge_regressor.fit(self.DE_dfX, self.DE_dfY)
    DEX_train, DEX_test, DEy_train, DEy_test = train_test_split(self.DE_dfX, self.DE_dfY, test_size=0.3,random_state=42)
    predictions = ridge_regressor.predict(DEX_test)
    print(predictions)
    sns.displot(DEy_test - predictions)
    plt.show()
    print(ridge_regressor.best_params_)
    print(ridge_regressor.best_score_)
    rmse = np.sqrt(mean_squared_error(DEy_test, predictions))
    print("rmse",rmse)
    print("R2 score : %.2f" % r2_score(DEy_test, predictions))
    res= stats.spearmanr(DEy_test, predictions)
    print(res)
1 usage
```

```
print(ridge_regressor.best_params_)
print(ridge_regressor.best_score_)
```

On a ainsi pu obtenir les meilleurs paramètres pour notre modèle en ayant fait varier le alpha. GridSearchCV nous a permis de chercher automatiquement les meilleurs paramètres optimums pour notre modèle de modélisation des données afin d'obtenir le meilleur résultat possible

```
{'alpha': 1e-15}
-1.6224519428503906e-25
rmse 3.963785438377839e-13
R2 score : 1.00
SignificanceResult(statistic=array([[1.        , 0.00290649, 1.        , 0.00290112],
       [0.00290649, 1.        , 0.00290649, 0.99999982],
       [1.        , 0.00290649, 1.        , 0.00290112],
       [0.00290112, 0.99999982, 0.00290112, 1.        ]]), pvalue=array([[0.        , 0.96309003, 0.        , 0.96315811],
       [0.96309003, 0.        , 0.96309003, 0.        ],
       [0.        , 0.96309003, 0.        , 0.96315811],
       [0.96315811, 0.        , 0.96315811, 0.        ]]))
```

Ainsi grâce au GridSearchCV on a trouvé le alpha optimal ainsi que le MSE optimal. Alors on a pu aussi déterminer une liste des modèles en termes de performance :

1-Forêts Aléatoires

2-Régression RIDGE

3-LASSO

4-Arbres de décision

5-Régression linéaire simple

6-K-NN

On a pu obtenir une très bonne performance en augmentant le n_estimators (améliore la précision)

on a augmenté Maxdepth pour avoir une bonne performance (améliorer la précision) et on a augmenté la profondeur sans overfitter la modélisation.

Pour éviter l'overfitting on a augmenté le min_samples_split et le min_samples_leaf pour éviter l'overfitting. On a réduit max_features pour réduire le sure apprentissage.

Conclusion

Notre projet de modélisation du prix de l'alimentation électrique en France et en Allemagne en utilisant des données météorologiques, énergétiques et commerciales a été mené à bien en respectant les différentes étapes de la méthodologie CRISP-DM. Nous avons réussi à créer un modèle qui évalue le changement quotidien du prix des contrats à terme sur l'alimentation électrique en se basant sur les conditions actuelles du marché et des variables explicatives telles que les mesures quotidiennes de données météorologiques, de fabrication énergétique et d'utilisation de l'alimentation électrique.

Malgré les difficultés rencontrées, ce projet nous a apporté une grande expérience en matière de traitement et de modélisation des données, ainsi que de gestion de projet. Nous avons également appris à travailler en équipe et à collaborer avec des experts dans différents domaines, ce qui a renforcé notre capacité à résoudre des problèmes complexes. En outre, nous avons pu acquérir une compréhension plus approfondie du marché de l'énergie et de ses facteurs d'influence, ainsi que de l'importance de l'analyse prédictive pour prendre des décisions éclairées.

Finalement, ce projet nous a permis de mettre en pratique nos connaissances en analyse de données et de développer des compétences utiles pour notre future carrière professionnelle. Notre travail nous rend ainsi plus confiant et prêts quant à nos futurs examens.