methodology-final

September 30, 2024

Chargement des bibliothèques et du dataset

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
sns.set(style="whitegrid")
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.metrics import mean_absolute_percentage_error, r2_score,␣
 ↪accuracy_score
from sklearn.cluster import KMeans
import warnings
warnings.filterwarnings("ignore")

data_path  = "data/"
data = pd.read_csv(data_path+"raw_data.csv", index_col="Date",
                   na_values=['#REF!'], decimal=",")
# data = pd.read_csv(data_path+"structured_data.csv",
#                    index_col="Date", parse_dates=True)
# data
```

1 Tables des matières

1.1 I - Contexte

L'objectif de ce projet est de prévoir la consommation horaire d'énergie de la population béninoise afin d'anticiper la demande future. Les données couvrent une période allant du **1 Jan 2017 au 31**

Dec 2023, avec un total de **2556 jours**. Chaque jour est représenté par 24 colonnes correspondant aux heures de la journée, de **00:00 à 23:00**, indiquant la consommation d'énergie en Mégawatts.

Ce travail s'inscrit dans une problématique classique de série temporelle.

[2]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2556 entries, 01-01-17 to 31-12-23
Data columns (total 24 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   00:00   2556 non-null   float64
 1   01:00   2556 non-null   float64
 2   02:00   2556 non-null   float64
 3   03:00   2556 non-null   float64
 4   04:00   2556 non-null   float64
 5   05:00   2556 non-null   float64
 6   06:00   2556 non-null   float64
 7   07:00   2556 non-null   float64
 8   08:00   2556 non-null   float64
 9   09:00   2556 non-null   float64
 10  10:00   2556 non-null   float64
 11  11:00   2556 non-null   float64
 12  12:00   2556 non-null   float64
 13  13:00   2555 non-null   float64
 14  14:00   2556 non-null   float64
 15  15:00   2556 non-null   float64
 16  16:00   2556 non-null   float64
 17  17:00   2555 non-null   float64
 18  18:00   2556 non-null   float64
 19  19:00   2556 non-null   float64
 20  20:00   2556 non-null   float64
 21  21:00   2556 non-null   float64
 22  22:00   2556 non-null   float64
 23  23:00   2556 non-null   float64
dtypes: float64(24)
memory usage: 499.2+ KB
```

1.2 II - Preprocessing

1.2.1 Structuration des données

Pour une meilleure pertinence, le DataFrame a été restructuré afin d'obtenir une forme plus adaptée et contextualisée pour les prochaines étapes. Disposer de 24 colonnes représentant les heures et les valeurs de consommation d'énergie n'était pas optimal. Il a donc été nécessaire de réorganiser les données pour obtenir une structure plus cohérente, avec chaque heure associée à son jour respectif.

[2]:
```
data_reset = data.reset_index()
data_reset.rename(columns={'index': 'Date'}, inplace=True)

data_melted = pd.melt(data_reset, id_vars=['Date'],
                      var_name='Hour', value_name='Power')

data_melted['Date'] = pd.to_datetime(data_melted['Date'] +
                                     ' ' + data_melted['Hour'],
                                     format='%d-%m-%y %H:%M')

data_structured = data_melted.set_index('Date').drop(['Hour'], axis=1)
data_structured = data_structured.sort_index()
data_structured
```

[2]:

Date	Power
2017-01-01 00:00:00	134.47
2017-01-01 01:00:00	165.92
2017-01-01 02:00:00	142.98
2017-01-01 03:00:00	146.74
2017-01-01 04:00:00	148.82
…	…
2023-12-31 19:00:00	233.90
2023-12-31 20:00:00	260.09
2023-12-31 21:00:00	252.54
2023-12-31 22:00:00	252.74
2023-12-31 23:00:00	237.69

[61344 rows x 1 columns]

Après restructuration, la fréquence des données a été définie de manière à créer des lignes pour chaque heure de la journée, conformément au contexte de prévision horaire. Cette étape a permis de mettre en évidence les valeurs manquantes sous-jacentes dans notre DataFrame.

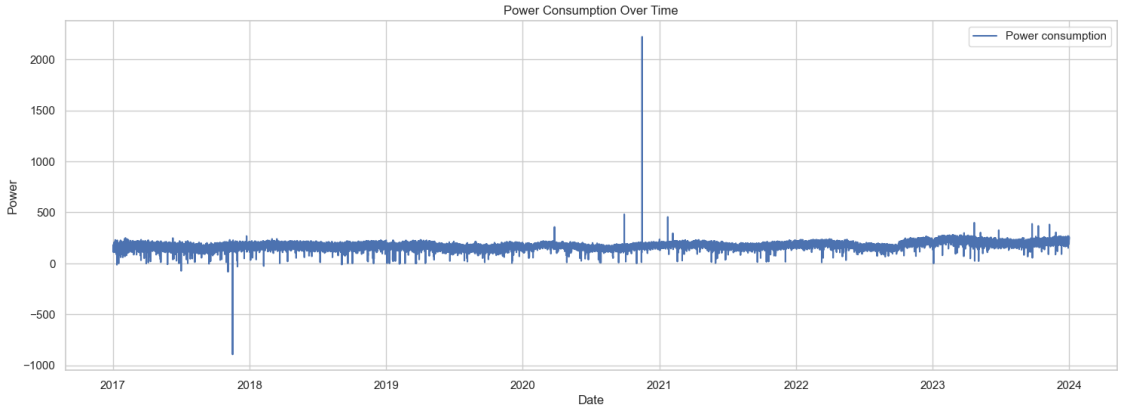
[3]:
```
hourly_data = data_structured.asfreq('H')
hourly_data.info()
print("\n", "Total de valeurs manquantes dans le dataset : ",
      hourly_data.isna().sum().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 61344 entries, 2017-01-01 00:00:00 to 2023-12-31 23:00:00
Freq: H
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Power   61342 non-null  float64
dtypes: float64(1)
memory usage: 958.5 KB
```

```
Total de valeurs manquantes dans le dataset :  2
```

Jetons à présent un aperçu de la consommation d'énergie au fil du temps. Le graphique ci-dessous illustre l'évolution de la consommation horaire sur l'ensemble de la période étudiée

[5]:
```
plt.figure(figsize=(18, 6))
plt.plot(hourly_data.index, hourly_data['Power'],
         label='Power consumption')
plt.title('Power Consumption Over Time')
plt.xlabel('Date')
plt.ylabel('Power')
plt.legend()
plt.show()
```



1.2.2 Valeurs abérantes

Le jeu de données présente des valeurs aberrantes, comme l'indiquent les premières analyses visuelles. On observe en effet des valeurs égales à 0, ainsi que des valeurs négatives, ce qui est problématique. Étant donné qu'il s'agit de la consommation d'énergie électrique par la population, les valeurs de la variable **Power** ne peuvent descendre en dessous de 0. Les valeurs égales à 0 quant à elles, peuvent indiquer des périodes de coupure d'électricité.

Dans l'un ou l'autre des cas, ces valeurs aberrantes sont susceptibles de fausser les analyses ultérieures et doivent être corrigées. Nous commencerons donc par les supprimer de notre dataset.

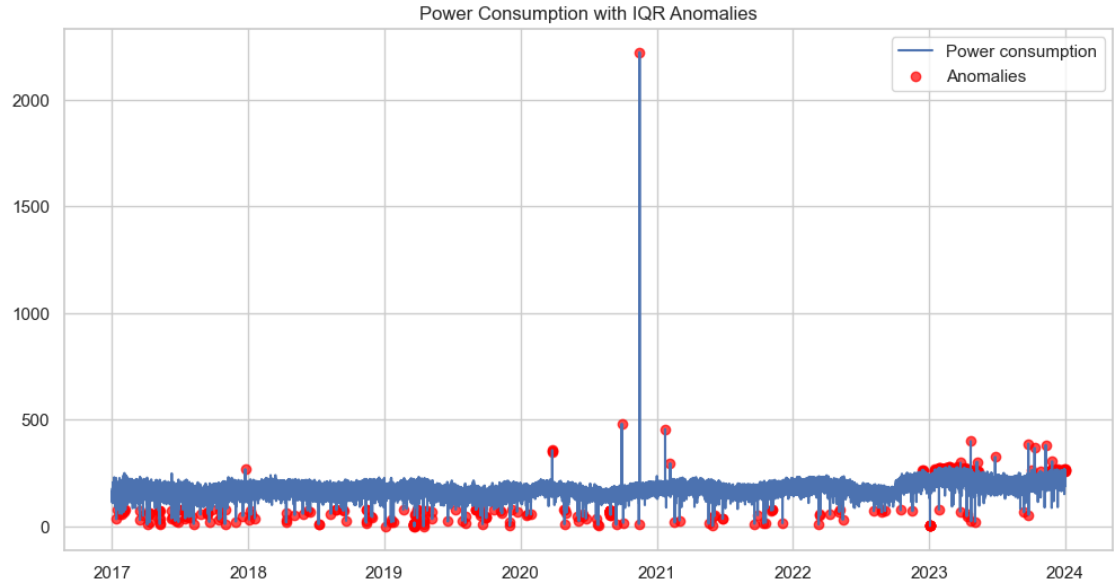
[4]:
```
hourly_data_no_negative = hourly_data[hourly_data['Power'] > 0].asfreq('H')
print("Total de valeurs manquantes dans le dataset : ",
      hourly_data_no_negative.isna().sum().sum())
```

```
Total de valeurs manquantes dans le dataset :  77
```

On remarque que la suppression a entrainée un total de **2** (valeurs manquantes initiales) + **75** (valeurs <= 0) valeurs

A présent les valeurs inférieures ou égales à 0 supprimées, utilisons la technique de l'IQR pour identifier d'autres valeurs abérantes.

[5]:
```
Q1 = hourly_data_no_negative['Power'].quantile(0.25)
Q3 = hourly_data_no_negative['Power'].quantile(0.75)
IQR = Q3 - Q1
anomalies = hourly_data_no_negative[(hourly_data_no_negative['Power'] < (Q1 - 1.
 ↪5 * IQR))
                                    | (hourly_data_no_negative['Power'] > (Q3 +␣
 ↪1.5 * IQR))]

# Visualisation
plt.figure(figsize=(12, 6))
plt.plot(hourly_data_no_negative.index, hourly_data_no_negative['Power'],
         label='Power consumption')
plt.scatter(anomalies.index, anomalies['Power'], color='red',
            label='Anomalies', alpha=0.7)
plt.title('Power Consumption with IQR Anomalies')
plt.legend()
plt.show()
print(anomalies.shape)
```




```
(507, 1)
```

[6]:
```
cleaned_data = hourly_data_no_negative[(hourly_data_no_negative['Power'] >= (Q1␣
 ↪- 1.5 * IQR)) &
                                       (hourly_data_no_negative['Power'] <= (Q3␣
 ↪+ 1.5 * IQR))]
```

```
print("Total de valeurs manquantes dans le dataset : ",
      cleaned_data.asfreq("H").isna().sum().sum())
```

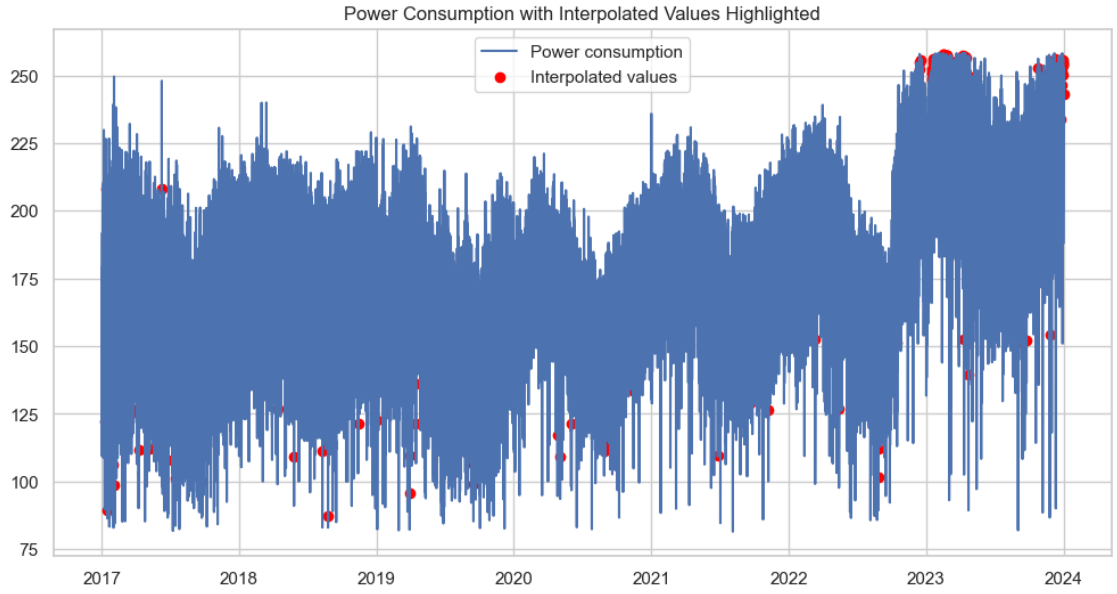
```
Total de valeurs manquantes dans le dataset :  584
```

1.2.3 Valeurs manquantes

La suppression des valeurs aberrantes dans le dataset a entraîné la création de valeurs manquantes qu'il convient maintenant d'imputer. Étant donné qu'il s'agit d'un problème de séries temporelles et compte tenu des caractéristiques de notre jeu de données, l'interpolation linéaire simple apparaît comme la méthode la plus appropriée

[7]:
```
missing_values = cleaned_data["Power"].asfreq("H").isna()
hourly_cleaned_data = cleaned_data.asfreq("H")
hourly_cleaned_data["Power"] = hourly_cleaned_data["Power"].
 ↪interpolate("linear")
interpolated_values = hourly_cleaned_data[missing_values]
print("Visualisation du dataset épuré")
plt.figure(figsize=(12, 6))
plt.plot(hourly_cleaned_data.index, hourly_cleaned_data["Power"],
         label='Power consumption')
plt.scatter(interpolated_values.index, interpolated_values["Power"],
            color='red', label='Interpolated values', alpha=1)
plt.title('Power Consumption with Interpolated Values Highlighted')
plt.legend()
plt.show()
```

```
Visualisation du dataset épuré
```



[9]:
```
print(hourly_cleaned_data.isna().sum())
```

```
Power    0
dtype: int64
```

Le dataset est désormais épuré, sans valeurs aberrantes ni manquantes.

1.2.4 Stationarité

Vérifions si notre série est stationnaire avec le test de Dickey-Fuller

[11]:
```
from statsmodels.tsa.stattools import adfuller

result = adfuller(hourly_cleaned_data['Power'])
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')

if result[1] < 0.05:
    print("La série est stationnaire")
else:
    print("La série n'est pas stationnaire")
```

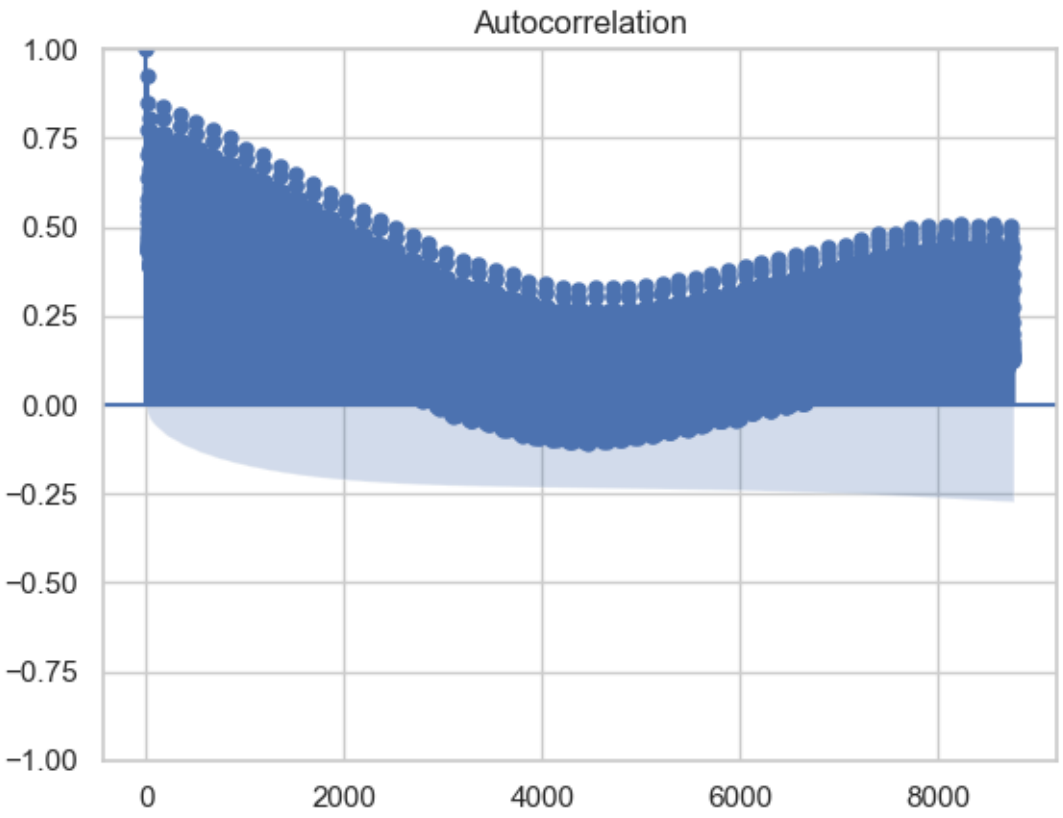
```
ADF Statistic: -10.260962557944296
p-value: 4.237943205100269e-18
La série est stationnaire
```

1.3 III- Analyse Exploratoire des données

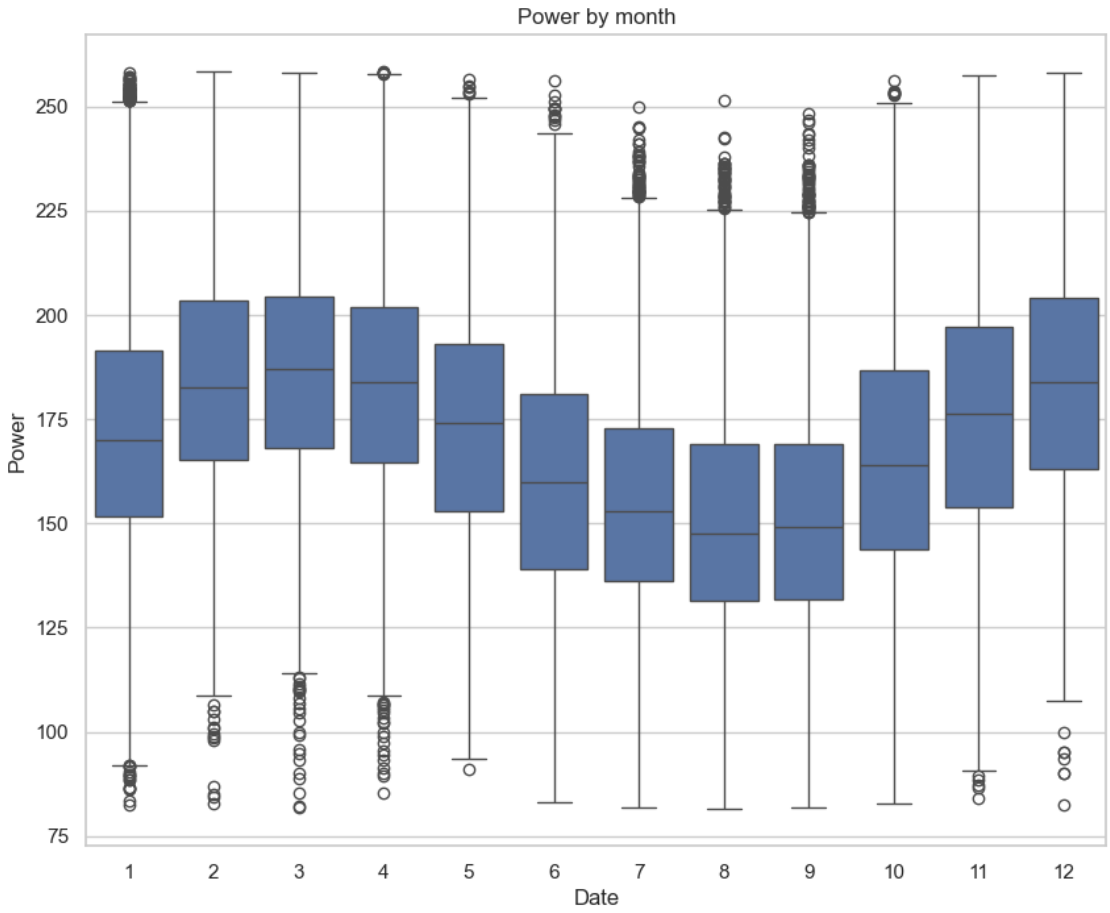
1.3.1 ACF

L'analyse de la fonction d'autocorrélation (ACF) sur une période de **365 jours** a révélé visuellement une tendance annuelle parfaitement distincte. Cette observation indique une saisonnalité claire au cours de l'année, avec un motif récurrent qui se manifeste une fois par an, formant un cycle sinusoïdal.

[12]:
```
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(hourly_cleaned_data["Power"], lags=365*24)
plt.show()
```



[25]:
```
fig, ax = plt.subplots(figsize = (10, 8))
sns.boxplot(hourly_cleaned_data, x = hourly_cleaned_data.index.month, y =␣
↪'Power')
ax.set_title('Power by month')
plt.show()
```



On remarque ainsi une saisonalité claire dans les données au cours d'une année

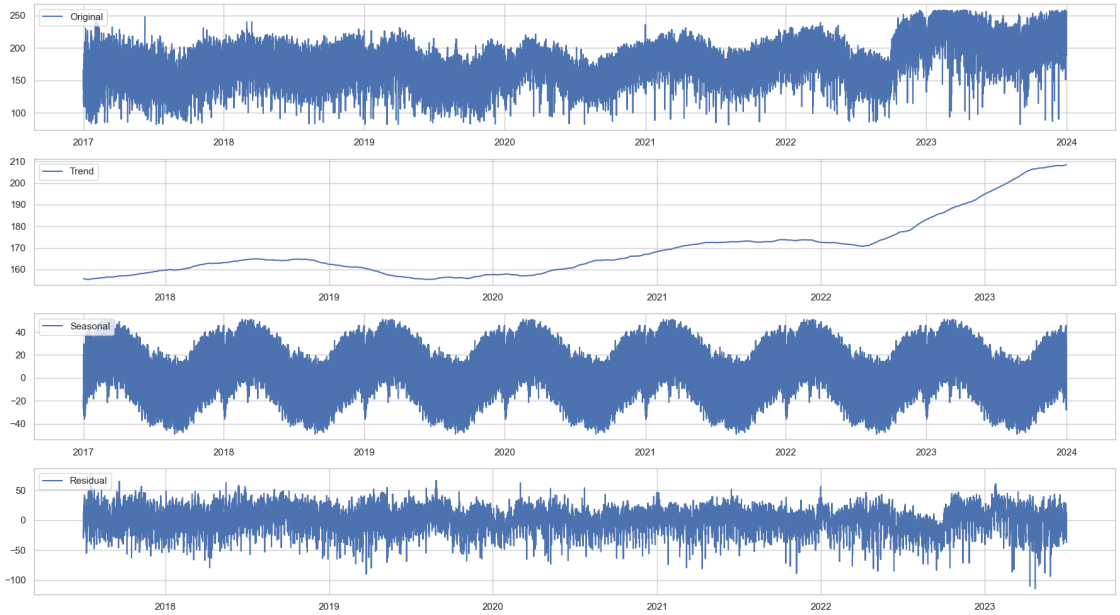
1.3.2 Seasonal Decomposition - Step 1

Essayons de mieux comprendre cette saisonnalité en effectuant une décomposition saisonnière pour la rendre plus perceptible.

- Commençons par visualiser les données sur l'ensemble de la période de 7 ans, en mettant particulièrement l'accent sur la recherche d'une saisonnalité annuelle.

[13]:
```
from statsmodels.tsa.seasonal import seasonal_decompose

start_date = '2017-01-01'
end_date = '2023-12-31'
data_three_years = hourly_cleaned_data[start_date:end_date]

# Décomposition
decomposition = seasonal_decompose(data_three_years['Power'],
                                   model='additive', period=24*365)
```

```python
# Visualisation
plt.figure(figsize=(18, 10))
plt.subplot(4, 1, 1)
plt.plot(data_three_years['Power'], label='Original')
plt.legend(loc='upper left')
plt.subplot(4, 1, 2)
plt.plot(decomposition.trend, label='Trend')
plt.legend(loc='upper left')
plt.subplot(4, 1, 3)
plt.plot(decomposition.seasonal, label='Seasonal')
plt.legend(loc='upper left')
plt.subplot(4, 1, 4)
plt.plot(decomposition.resid, label='Residual')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



Interprétation

De cette analyse, il ressort clairement une saisonnalité annuelle parfaitement identifiable **(Graphique 3 - Seasonal)**, ainsi qu'une tendance haussière au fil des années **(Graphique 2 - Trend)**.

La population béninoise tend à consommer l'électricité de manière similaire à des moments précis de l'année, de façon répétitive, avec des besoins croissants au fil des années, probablement en raison de la croissance démographique ou de l'utilisation accrue d'appareils électroniques.

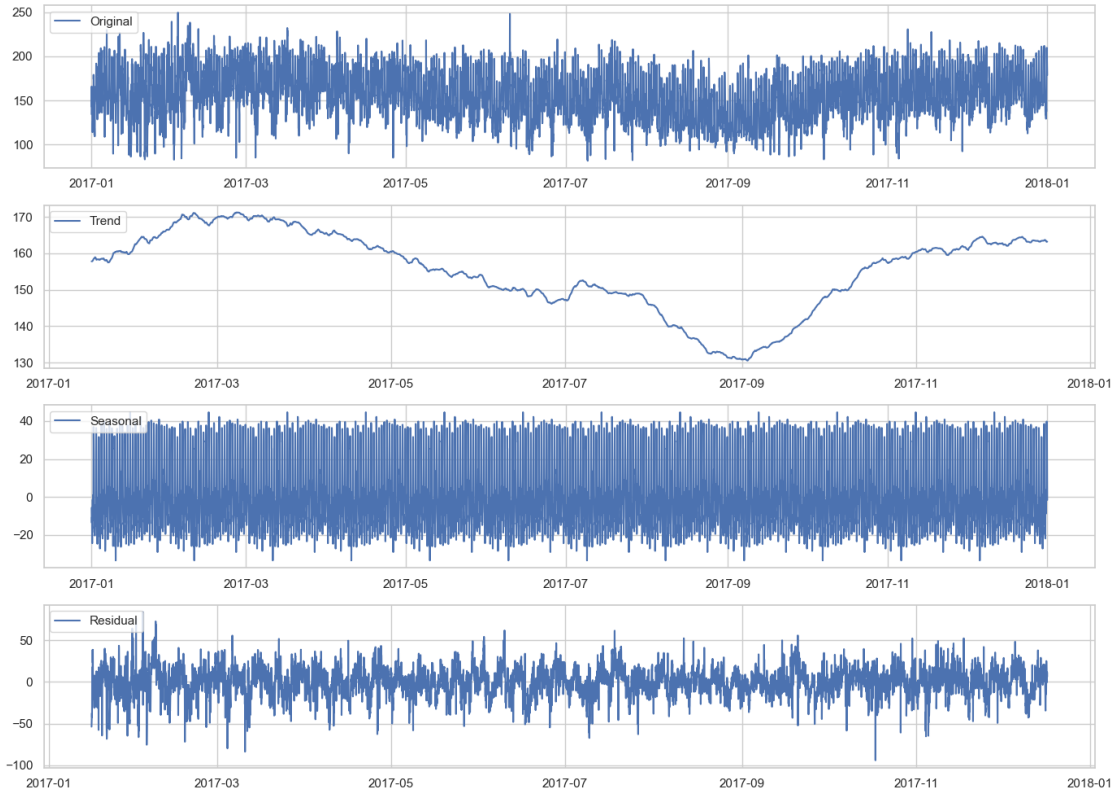
Cela suggère qu'une première piste de modélisation pourrait être l'utilisation de modèles de séries temporelles plus complexes, excluant ainsi les modèles linéaires tels que l'**ARIMA** au profil de

modèles plus complexe tels que SARIMA ou les Réseaux de neurones.

1.3.3 Seasonal Decomposition - Step 2

- Examinons plus en détail les variations saisonnières à une échelle plus petite **(Une année)** afin d'obtenir une meilleure compréhension des motifs récurrents au sein de la saisonalité annuelle identifié précédemment.

[14]:
```
# Données de l'année 2017
selected_year = 2017
data_year = hourly_cleaned_data[hourly_cleaned_data.index.year == selected_year]
decomposition = seasonal_decompose(data_year['Power'], model='additive',
                                   period=24*30)

# Visualisation
plt.figure(figsize=(14, 10))
plt.subplot(4, 1, 1)
plt.plot(data_year['Power'], label='Original')
plt.legend(loc='upper left')
plt.subplot(4, 1, 2)
plt.plot(decomposition.trend, label='Trend')
plt.legend(loc='upper left')
plt.subplot(4, 1, 3)
plt.plot(decomposition.seasonal, label='Seasonal')
plt.legend(loc='upper left')
plt.subplot(4, 1, 4)
plt.plot(decomposition.resid, label='Residual')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

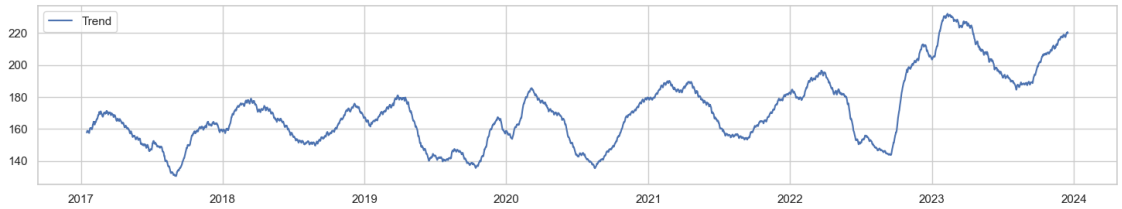


On remarque en effet la présence d'un schéma saisonnier au sein de l'année, avec une première phase de **Janvier à fin Août**, suivie d'une seconde phase de **Septembre jusqu'à la fin de l'année**.

1.3.4 Seasonal Decomposition - Step 3

- Entreprenons pour finir une analyse supplémentaire pour confirmer la saisonalité menseulle (Jan - Aout, Sept - Dec) et vérifier si elle se répète de manière cohérente chaque année.

[15]:
```
start_date = '2017-01-01'
end_date = '2023-12-31'
data_three_years = hourly_cleaned_data[start_date:end_date]
decomposition = seasonal_decompose(data_three_years['Power'],
                                   model='additive', period=24*30)

plt.figure(figsize=(18, 3))
plt.plot(decomposition.trend, label='Trend')
plt.legend(loc='upper left')
plt.show()
```



1.4 IV- Feature Engineering

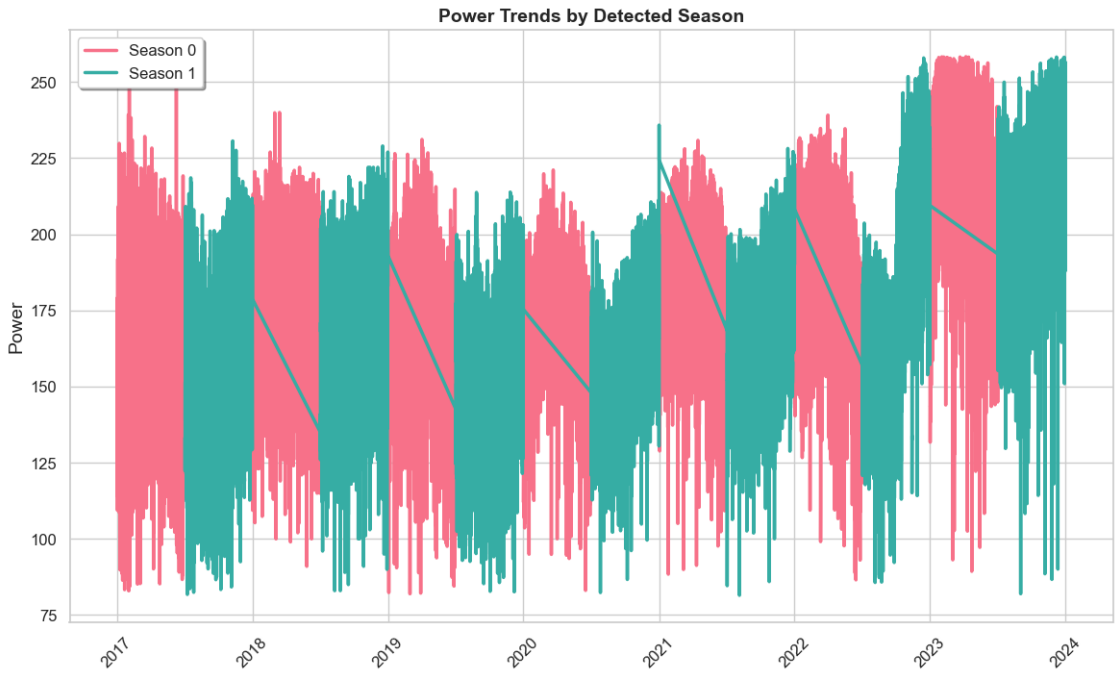
Au regard des différentes analyses menées, de nouvelles variables ont été créées pour améliorer la capacité du modèle à capturer les particularités temporelles de la série et ainsi fournir des prévisions de meilleure qualité. Les initiatives à entreprendre se résument en quatre parties :

- **Part 1** : Création de la variable **season_cluster** qui décrit la période de l'année (saisonnalité mensuelle) à laquelle appartient chaque observation, en fonction des résultats de la décomposition saisonnière.
- **Part 2** : Création de variables telles que **day_of_week**, **is_week_end** et **is_holiday**, susceptibles d'influencer la consommation d'électricité.
- **Part 3** : Création de la variable **power_cluster** qui indique dans quelle gamme de valeurs se situe la consommation en fonction des variables temporelles. Un sous-modèle de classification pourrait être utilisé pour aider le modèle principal à mieux faire ses prévisions.
- **Part 4** : Sélection des variables pertinentes parmi toutes celles créées.

1.4.1 Part - 1

[11]:
```
# Entrainement du modèle de clusteurisation sur une année
selected_year = 2017
data_year = hourly_cleaned_data.loc[hourly_cleaned_data.index.year ==␣
 ↪selected_year].copy()
data_year['day_of_year'] = data_year.index.dayofyear
data_year['hour_of_day'] = data_year.index.hour
features = data_year[['day_of_year', 'hour_of_day']]
kmeans = KMeans(n_clusters=2, n_init=10, random_state=0)
data_year['season_cluster'] = kmeans.fit_predict(features)
hourly_cleaned_data.loc[hourly_cleaned_data.index.year == selected_year,
                        'season_cluster'] = data_year['season_cluster']

# Classification pour l'ensemble du DataFrame
hourly_cleaned_data['day_of_year'] = hourly_cleaned_data.index.dayofyear
hourly_cleaned_data['hour_of_day'] = hourly_cleaned_data.index.hour
features_all = hourly_cleaned_data[['day_of_year', 'hour_of_day']]
hourly_cleaned_data['season_cluster'] = kmeans.predict(features_all)

#Visulaisation
```

```
hourly_cleaned_data_sorted = hourly_cleaned_data.sort_index()
palette = sns.color_palette("husl", len(np.
 ↪unique(hourly_cleaned_data_sorted['season_cluster'])))
plt.figure(figsize=(14, 8))
for i, cluster in enumerate(np.
 ↪unique(hourly_cleaned_data_sorted['season_cluster'])):
    cluster_data =␣
 ↪hourly_cleaned_data_sorted[hourly_cleaned_data_sorted['season_cluster'] ==␣
 ↪cluster]
    plt.plot(cluster_data.index, cluster_data['Power'], label=f'Season␣
 ↪{cluster}',
             color=palette[i], linewidth=2.5)
plt.title('Power Trends by Detected Season', fontsize=14, fontweight='bold')
plt.ylabel('Power', fontsize=14)
plt.legend(loc='upper left', fontsize=12, frameon=True, shadow=True,␣
 ↪fancybox=True)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



1.4.2 Part - 2

Ajout d'autres variables temporelles notamment :

- Le jour de la semaine (Lun - Dim)

- Le week-end de l'année (1-54)
- Le mois de l'année (1-12)
- Une booléenne pour les jours week-end (0,1)
- Une booléenne pour les jours fériés (0,1)

[12]:
```
# Mois de l'année
hourly_cleaned_data['month_of_year'] = hourly_cleaned_data.index.month
# Semaine de l'année
hourly_cleaned_data['week_of_year'] = hourly_cleaned_data.index.to_series().
 ↪apply(
                                                        lambda x: x.
 ↪isocalendar()[1])
# Jour de la semaine
hourly_cleaned_data['day_of_week'] = hourly_cleaned_data.index.day_name()
# Encodage des jours de la semaine
day_of_week_mapping = {day: i for i, day in enumerate(['Monday', 'Tuesday',
                                                       'Wednesday', 'Thursday',
                                                       'Friday', 'Saturday',
                                                       'Sunday'])}
hourly_cleaned_data['day_of_week'] = hourly_cleaned_data['day_of_week'].
 ↪map(day_of_week_mapping)
# Jour de Week-end
hourly_cleaned_data['is_weekend'] = hourly_cleaned_data.index.to_series().dt.
 ↪dayofweek.isin([5, 6])
hourly_cleaned_data['is_weekend'] = hourly_cleaned_data['is_weekend'].
 ↪astype(int)
```

Ajout des jours fériés

[13]:
```
special_dates = ['01-01', # Jour de l'an
                 '01-10', # Fête du Vodoun
                 '04-01', # Lundi de Pâques
                 '04-10', # Aïd el-Fitr
                 '05-01', # Fête du Travail
                 '05-09', # Ascension
                 '05-20', # Lundi de Pentecôte
                 '06-16', # Tabaski
                 '08-01', # Indépendance
                 '08-15', # Assomption,
                 '09-16', # Maouloud
                 '11-01', # Toussaint
                 '12-25'] # Noël
def mark_special_dates(date):
    return 1 if date.strftime('%m-%d') in special_dates else 0
hourly_cleaned_data['is_holiday'] = hourly_cleaned_data.index.to_series().apply(
                                                                    ␣
 ↪mark_special_dates)
```

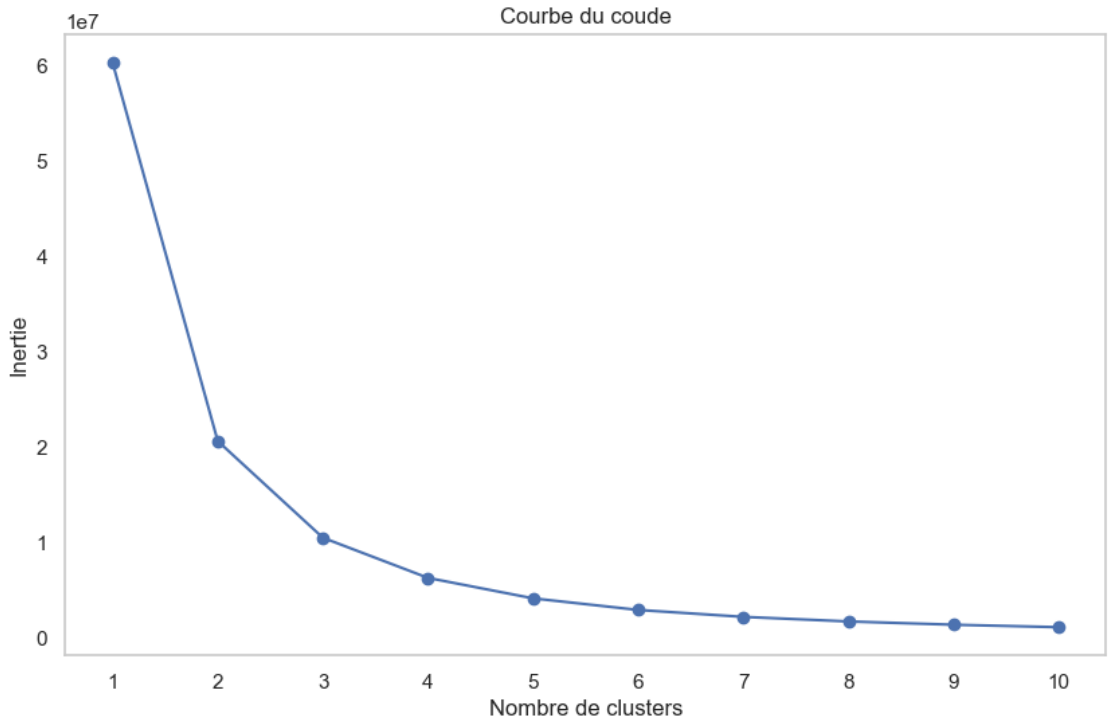
1.4.3 Part - 3

Les valeurs de Power seront regroupées en 5 gammes de valeurs, correspondant à **5 clusters**. Bien que l'augmentation du nombre de clusters puisse améliorer la performance du modèle final, cela augmente également le risque de biais introduit par le modèle intermédiaire de classification. Il a fallu donc, déterminer un nombre de classes qui équilibre les performances du modèle final et la précision du modèle de classification.

Ainsi, nous allons :

- Déterminer les gammes de valeurs pour constituer les classes à l'aide de l'algorithme **K-means**,
- Etiqueter les classes,
- Entrainer un modèle de classification (**DecisionTreeClassifier**) sur une partie des données d'entrainement (**80%**)
- Evaluer le modèle sur la partie restante (**20%**) afin d'effectuer des réajustements au besoin
- L'utiliser pour classifier toutes les valeurs que prendra la variable **Power** en fonction du temps

[15]:
```
X = hourly_cleaned_data[['Power']].values
inertias = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)
    inertias.append(kmeans.inertia_)
plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), inertias, marker='o')
plt.title('Courbe du coude')
plt.xlabel('Nombre de clusters')
plt.ylabel('Inertie')
plt.xticks(range(1, 11))
plt.grid()
plt.show()
```



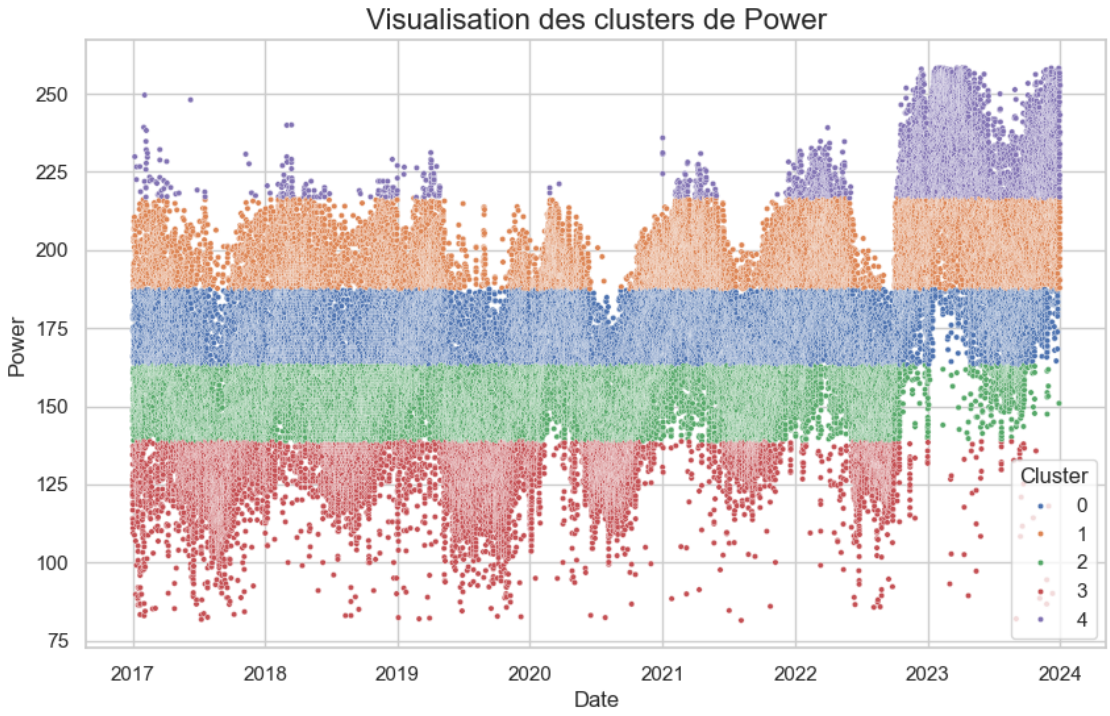
[16]:
```
from sklearn.tree import DecisionTreeClassifier
power_data = hourly_cleaned_data[['Power']]
X_train, X_test = train_test_split(power_data, test_size=0.2, random_state=0)
# Entraînement de l'algorithme KMeans pour 5 clusters sur les données␣
 ↪d'entraînement
kmeans = KMeans(n_clusters=5, n_init=10, random_state=0)
X_train['Power_cluster'] = kmeans.fit_predict(X_train)
# Attribution des étiquettes de cluster aux données de test
X_test['Power_cluster'] = kmeans.predict(X_test)
# Entraînement du modèle Decision Tree sur les données d'entraînement
clf = DecisionTreeClassifier(random_state=0)
clf.fit(X_train[['Power']], X_train['Power_cluster'])
y_pred = clf.predict(X_test[['Power']])
print(f"Accuracy Score: {accuracy_score(X_test['Power_cluster'], y_pred)}")
```

```
Accuracy Score: 1.0
```

[17]:
```
hourly_cleaned_data['Power_cluster'] = clf.
 ↪predict(hourly_cleaned_data[['Power']])
```

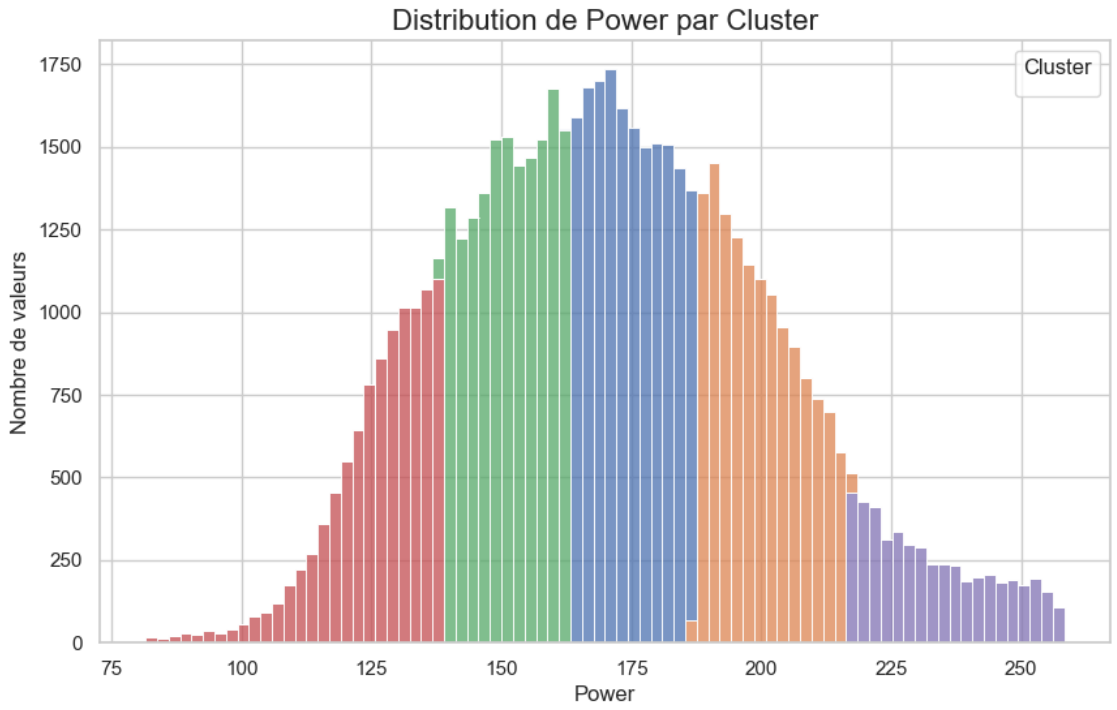
[18]:
```
plt.figure(figsize=(10, 6))
sns.scatterplot(x=hourly_cleaned_data.index, y=hourly_cleaned_data['Power'],
                hue=hourly_cleaned_data['Power_cluster'], palette='deep', s=10)
plt.title('Visualisation des clusters de Power', fontsize=16)
```

```
plt.ylabel('Power', fontsize=12)
plt.legend(title='Cluster')
plt.show()
```



[19]:
```
plt.figure(figsize=(10, 6))
sns.histplot(data=hourly_cleaned_data, x='Power',
             hue='Power_cluster',
             multiple='stack', palette='deep')
plt.title('Distribution de Power par Cluster', fontsize=16)
plt.xlabel('Power', fontsize=12)
plt.ylabel('Nombre de valeurs', fontsize=12)
plt.legend(title='Cluster')
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
```



[20]:
```
hourly_cleaned_data.sample(5)
```

[20]:

Date	Power	season_cluster	day_of_year	hour_of_day	month_of_year	week_of_year	day_of_week	is_weekend	is_holiday	Power_cluster
2023-07-12 06:00:00	149.63100	1	193	6	7	28	2	0	0	2
2023-03-25 03:00:00	212.32500	0	84	3	3	12	5	1	0	1
2019-02-08 02:00:00	160.95144	0	39	2	2	6	4	0	0	2
2018-03-12 12:00:00	174.00000	0	71	12	3	11	0	0	0	0
2018-09-23 09:00:00	124.00000	1	266	9	9	38	6	1		

```
2018-09-23 09:00:00           0              3
```

1.4.4 Part - 4

[18]:
```
hourly_cleaned_data.corr()
```

[18]:
```
                    Power  season_cluster   day_of_year   hour_of_day  \
Power            1.000000   -1.936409e-01 -1.148799e-01  3.563970e-01
season_cluster  -0.193641    1.000000e+00  8.660255e-01  1.124631e-16
day_of_year     -0.114880    8.660255e-01  1.000000e+00  2.032560e-16
hour_of_day      0.356397    1.124631e-16  2.032560e-16  1.000000e+00
month_of_year   -0.116155    8.667128e-01  9.965016e-01  2.125246e-16
week_of_year    -0.118093    8.525350e-01  9.766317e-01  1.862017e-16
day_of_week     -0.174130   -5.853355e-04  3.038703e-03  4.349527e-19
is_weekend      -0.219850   -1.111703e-03  1.492124e-03 -5.026728e-18
is_holiday      -0.042496   -4.388853e-02 -3.594564e-02  4.771997e-18
Power_cluster   -0.169593    4.743176e-02  3.452952e-02 -7.425914e-02

                month_of_year  week_of_year   day_of_week    is_weekend  \
Power           -1.161554e-01 -1.180928e-01 -1.741300e-01 -2.198503e-01
season_cluster   8.667128e-01  8.525350e-01 -5.853355e-04 -1.111703e-03
day_of_year      9.965016e-01  9.766317e-01  3.038703e-03  1.492124e-03
hour_of_day      2.125246e-16  1.862017e-16  4.349527e-19 -5.026728e-18
month_of_year    1.000000e+00  9.747871e-01  3.370378e-03  1.699359e-03
week_of_year     9.747871e-01  1.000000e+00  9.904108e-04  1.043934e-03
day_of_week      3.370378e-03  9.904108e-04  1.000000e+00  7.907625e-01
is_weekend       1.699359e-03  1.043934e-03  7.907625e-01  1.000000e+00
is_holiday      -2.492492e-02 -7.573929e-03 -6.445387e-03  9.226012e-03
Power_cluster    3.481173e-02  3.828590e-02  3.870197e-02  5.057541e-02

                  is_holiday  Power_cluster
Power           -4.249578e-02     -0.169593
season_cluster  -4.388853e-02      0.047432
day_of_year     -3.594564e-02      0.034530
hour_of_day      4.771997e-18     -0.074259
month_of_year   -2.492492e-02      0.034812
week_of_year    -7.573929e-03      0.038286
day_of_week     -6.445387e-03      0.038702
is_weekend       9.226012e-03      0.050575
is_holiday       1.000000e+00      0.001176
Power_cluster    1.176122e-03      1.000000
```

Suppression des variables **day__of__year** et **week__of__year** après analyse de la pertinence des variables et du coéfficient de correlation

[19]:
```
hourly_cleaned_data.drop(columns=["day_of_year","week_of_year"],inplace=True)
hourly_cleaned_data.corr()
```

[19]:
```
                    Power  season_cluster   hour_of_day  month_of_year  \
Power            1.000000   -1.936409e-01  3.563970e-01  -1.161554e-01
season_cluster -0.193641    1.000000e+00  1.124631e-16   8.667128e-01
hour_of_day     0.356397    1.124631e-16  1.000000e+00   2.125246e-16
month_of_year  -0.116155    8.667128e-01  2.125246e-16   1.000000e+00
day_of_week    -0.174130   -5.853355e-04  4.349527e-19   3.370378e-03
is_weekend     -0.219850   -1.111703e-03 -5.026728e-18   1.699359e-03
is_holiday     -0.042496   -4.388853e-02  4.771997e-18  -2.492492e-02
Power_cluster  -0.169593    4.743176e-02 -7.425914e-02   3.481173e-02

                 day_of_week    is_weekend    is_holiday  Power_cluster
Power          -1.741300e-01 -2.198503e-01 -4.249578e-02      -0.169593
season_cluster -5.853355e-04 -1.111703e-03 -4.388853e-02       0.047432
hour_of_day     4.349527e-19 -5.026728e-18  4.771997e-18      -0.074259
month_of_year   3.370378e-03  1.699359e-03 -2.492492e-02       0.034812
day_of_week     1.000000e+00  7.907625e-01 -6.445387e-03       0.038702
is_weekend      7.907625e-01  1.000000e+00  9.226012e-03       0.050575
is_holiday     -6.445387e-03  9.226012e-03  1.000000e+00       0.001176
Power_cluster   3.870197e-02  5.057541e-02  1.176122e-03       1.000000
```

[20]:
```
hourly_cleaned_data.isna().sum()
```

[20]:
```
Power             0
season_cluster    0
hour_of_day       0
month_of_year     0
day_of_week       0
is_weekend        0
is_holiday        0
Power_cluster     0
dtype: int64
```

1.5 V- Modélisation

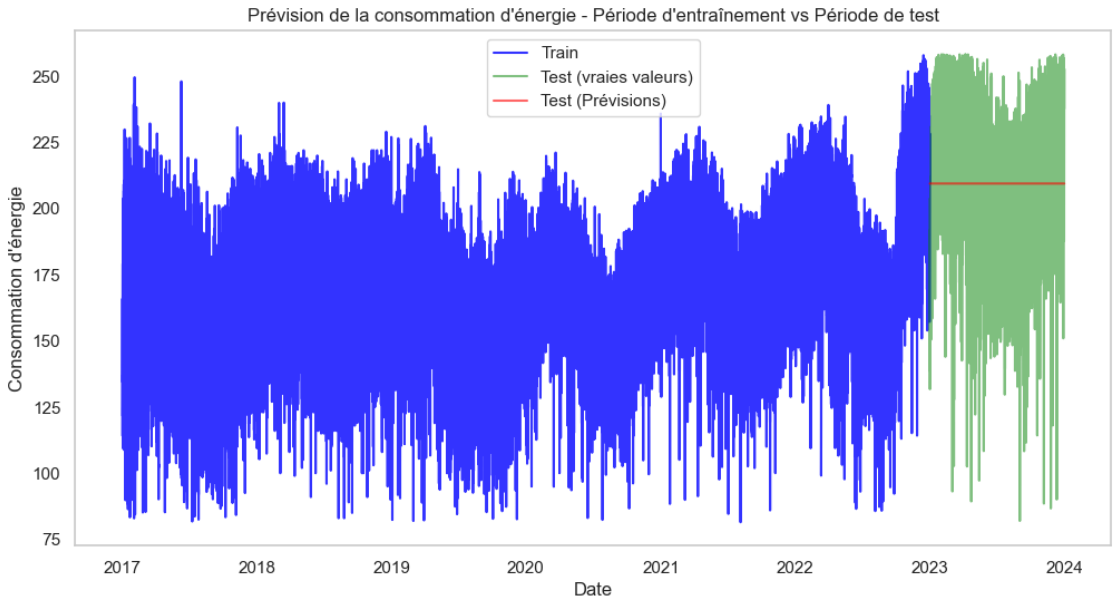
1.5.1 ARIMA

Nous allons d'abord utiliser le modèle ARIMA comme référence de base. Ses prévisions nous permettront de comparer les performances des modèles futurs, un peu à la manière d'un modèle naïf.

[21]:
```
from statsmodels.tsa.arima.model import ARIMA
train = hourly_cleaned_data[["Power"]][:'2022-12-31'].copy()
test = hourly_cleaned_data[["Power"]]['2023-01-01':].copy()
model = ARIMA(train, order=(1, 1, 1))
model_fit = model.fit()
y_pred = model_fit.forecast(steps=len(test))
mae_arima = mean_absolute_error(test['Power'], y_pred)
rmse_arima = mean_squared_error(test['Power'], y_pred, squared=False)
```

```
mape_arima = mean_absolute_percentage_error(test['Power'], y_pred)
r2_arima = r2_score(test['Power'], y_pred)
print(f"ARIMA \n ------ \n MAE: {mae_arima} \n RMSE: {rmse_arima} \n MAPE:␣
 ↪{mape_arima} \n R²: {r2_arima}")
# Visualisation des résultats
train_size = len(train)
plt.figure(figsize=(12, 6))
plt.plot(hourly_cleaned_data.index[:train_size],
         hourly_cleaned_data['Power'][:train_size],
         label='Train', color='blue', alpha=0.8)
plt.plot(hourly_cleaned_data.index[train_size:],
         test['Power'], label='Test (vraies valeurs)', color='green', alpha=0.5)
plt.plot(hourly_cleaned_data.index[train_size:],
         y_pred, label='Test (Prévisions)', color='red', alpha=0.6)
plt.title('Prévision de la consommation d\'énergie - Période d\'entraînement vs␣
 ↪Période de test')
plt.xlabel('Date')
plt.ylabel('Consommation d\'énergie')
plt.legend()
plt.grid()
plt.show()
```

```
ARIMA
 ------
 MAE: 22.80537492776746
 RMSE: 27.656944161868317
 MAPE: 0.11524235311289133
 R²: -0.0011729889130780435
```

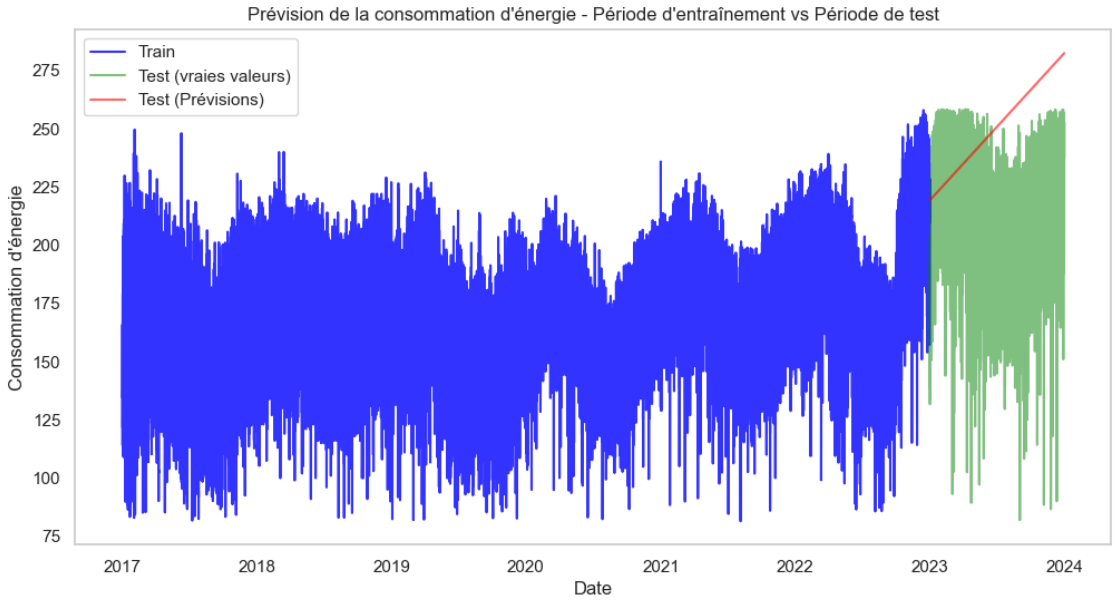


On peut clairement constater l'incapacité du modèle à s'ajuster correctement aux données et à capturer les tendances.

Nous allons maintenant tester des modèles un peu plus avancés, souvent utilisés en prévision, afin d'observer les résultats obtenus.

[22]:
```
from pygam import GAM, s
train = hourly_cleaned_data[["Power"]][:'2022-12-31'].copy()
test = hourly_cleaned_data[["Power"]]['2023-01-01':].copy()
train['time_index'] = range(len(train))
test['time_index'] = range(len(train), len(train) + len(test))
gam = GAM(s(0)).fit(train['time_index'], train['Power'])
y_pred = gam.predict(test['time_index'])
mae_gam = mean_absolute_error(test['Power'], y_pred)
rmse_gam = mean_squared_error(test['Power'], y_pred, squared=False)
mape_gam = mean_absolute_percentage_error(test['Power'], y_pred)
r2_gam = r2_score(test['Power'], y_pred)
print(f"GAM \n ------ \n MAE: {mae_gam} \n RMSE: {rmse_gam} \n MAPE: {mape_gam}␣
 ↪\n R²: {r2_gam}")
# Visualisation des résultats
train_size = len(train)
plt.figure(figsize=(12, 6))
plt.plot(hourly_cleaned_data.index[:train_size],
         hourly_cleaned_data['Power'][:train_size],
         label='Train', color='blue', alpha=0.8)
plt.plot(hourly_cleaned_data.index[train_size:],
         test['Power'], label='Test (vraies valeurs)', color='green', alpha=0.5)
plt.plot(hourly_cleaned_data.index[train_size:],
         y_pred, label='Test (Prévisions)', color='red', alpha=0.6)
plt.title('Prévision de la consommation d\'énergie - Période d\'entraînement vs␣
 ↪Période de test')
plt.xlabel('Date')
plt.ylabel('Consommation d\'énergie')
plt.legend()
plt.grid()
plt.show()
```

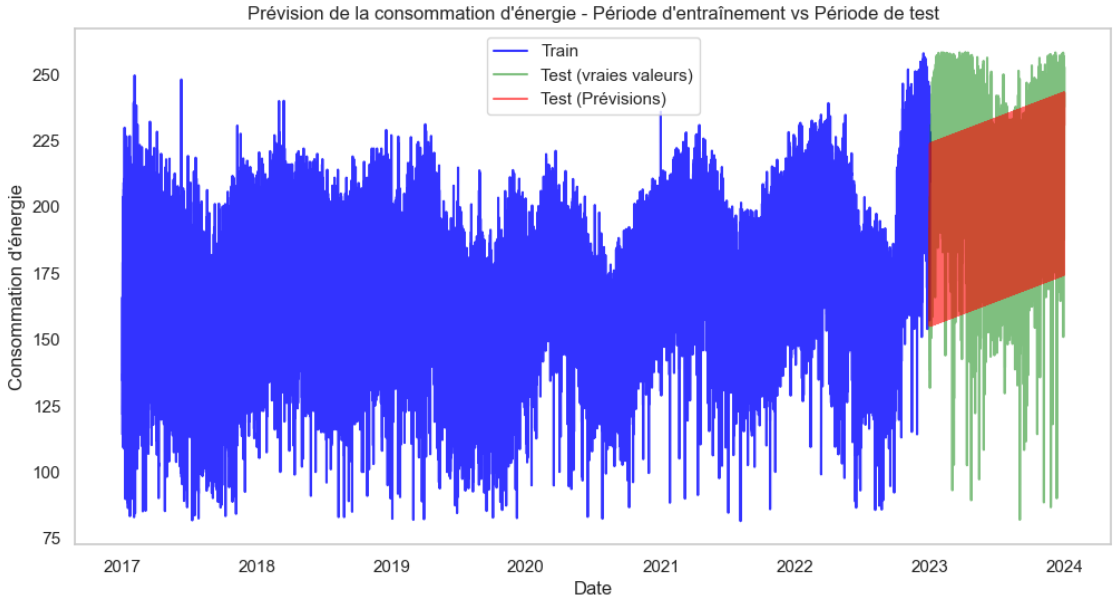
```
GAM
 ------
 MAE: 46.947070961491804
 RMSE: 55.461165949094415
 MAPE: 0.24698032519740004
 R²: -3.0260459676784697
```



[29]:
```
from statsmodels.tsa.holtwinters import ExponentialSmoothing
train = hourly_cleaned_data[["Power"]][:'2022-12-31'].copy()
test = hourly_cleaned_data[["Power"]]['2023-01-01':].copy()
hw_model = ExponentialSmoothing(train['Power'], trend='add', seasonal='add',␣
 ↪seasonal_periods=24)
hw_fit = hw_model.fit()
y_pred = hw_fit.forecast(len(test))
mae_smooth = mean_absolute_error(test['Power'], y_pred)
rmse_smooth = mean_squared_error(test['Power'], y_pred, squared=False)
mape_smooth = mean_absolute_percentage_error(test['Power'], y_pred)
r2_smooth = r2_score(test['Power'], y_pred)
print(f"ExponentialSmoothing \n ------ \n MAE: {mae_smooth} \n RMSE:␣
 ↪{rmse_smooth} \n MAPE: {mape_smooth} \n R²: {r2_smooth}")
# Visualisation des résultats
train_size = len(train)
plt.figure(figsize=(12, 6))
plt.plot(hourly_cleaned_data.index[:train_size],
         hourly_cleaned_data['Power'][:train_size],
         label='Train', color='blue', alpha=0.8)
plt.plot(hourly_cleaned_data.index[train_size:],
         test['Power'], label='Test (vraies valeurs)', color='green', alpha=0.5)
plt.plot(hourly_cleaned_data.index[train_size:],
         y_pred, label='Test (Prévisions)', color='red', alpha=0.6)
plt.title('Prévision de la consommation d\'énergie - Période d\'entraînement vs␣
 ↪Période de test')
plt.xlabel('Date')
plt.ylabel('Consommation d\'énergie')
```

```
plt.legend()
plt.grid()
plt.show()
```

```
ExponentialSmoothing
 ------
 MAE: 21.583079842397098
 RMSE: 27.078616640125997
 MAPE: 0.10397845130442579
 R²: 0.04025980030718501
```



[23]:
```
from neuralprophet import NeuralProphet
train = hourly_cleaned_data[:'2022-12-31'][["Power"]].copy()
test = hourly_cleaned_data['2023-01-01':][["Power"]].copy()
train_prophet = train.rename(columns={'Power': 'y'})
train_prophet["ds"] = train.index
test_prophet = test.rename(columns={'Power': 'y'})
test_prophet["ds"] = test.index
model = NeuralProphet(batch_size=128, epochs=30, normalize="auto")
model.fit(train_prophet, freq='H')
future = model.make_future_dataframe(df=train_prophet, periods=len(test))
forecast_np = model.predict(future)['yhat1'].iloc[-len(test):]
y_pred = forecast_np.values
mae_np = mean_absolute_error(test['Power'], y_pred)
rmse_np = mean_squared_error(test['Power'], y_pred, squared=False)
mape_np = mean_absolute_percentage_error(test['Power'], y_pred)
r2_np = r2_score(test['Power'], y_pred)
```

```
print(f"NeuralProphet \n ------ \n MAE: {mae_np} \n RMSE: {rmse_np} \n MAPE:␣
 ↪{mape_np} \n R²: {r2_np}")
# Visualisation des résultats
train_size = len(train)
plt.figure(figsize=(12, 6))
plt.plot(hourly_cleaned_data.index[:train_size],
         hourly_cleaned_data['Power'][:train_size],
         label='Train', color='blue', alpha=0.8)
plt.plot(hourly_cleaned_data.index[train_size:],
         test['Power'], label='Test (vraies valeurs)', color='green', alpha=0.5)
plt.plot(hourly_cleaned_data.index[train_size:],
         y_pred, label='Test (Prévisions)', color='red', alpha=0.6)
plt.title('Prévision de la consommation d\'énergie - Période d\'entraînement vs␣
 ↪Période de test')
plt.xlabel('Date')
plt.ylabel('Consommation d\'énergie')
plt.legend()
plt.grid()
plt.show()
```

```
WARNING - (NP.forecaster.fit) - When Global modeling with local normalization,
metrics are displayed in normalized scale.
INFO - (NP.df_utils._infer_frequency) - Major frequency H corresponds to 99.998%
of the data.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major
frequency - H
INFO - (NP.config.init_data_params) - Setting normalization to global as only
one dataframe provided for training.
```

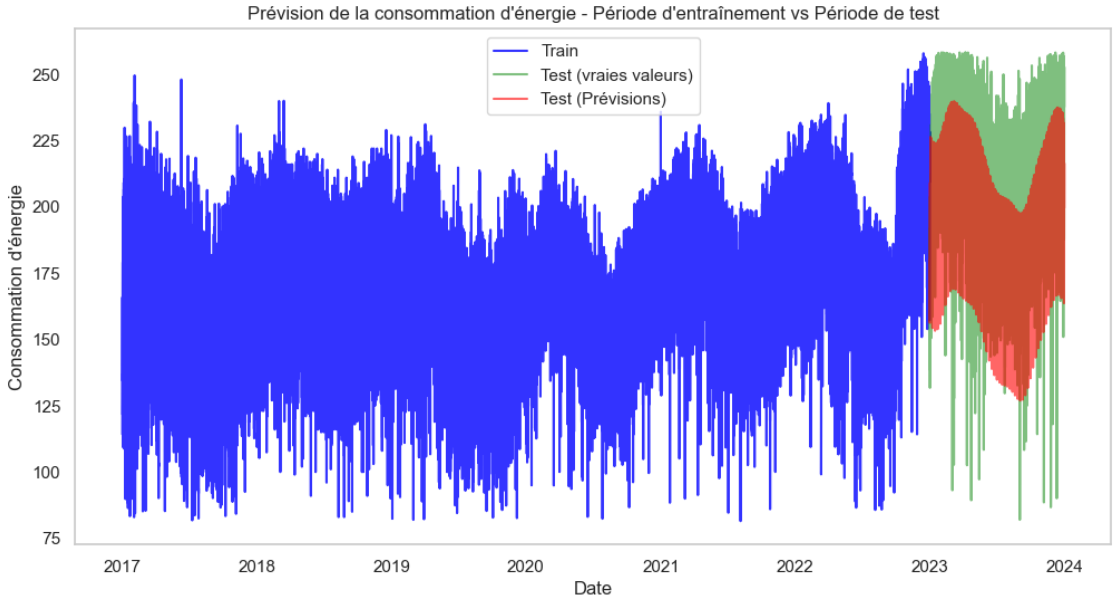
```
Finding best initial lr:   0%|          | 0/268 [00:00<?, ?it/s]
```

```
Training: 0it [00:00, ?it/s]
```

```
INFO - (NP.df_utils._infer_frequency) - Major frequency H corresponds to 99.998%
of the data.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major
frequency - H
INFO - (NP.df_utils.return_df_in_original_format) - Returning df with no ID
column
INFO - (NP.df_utils._infer_frequency) - Major frequency H corresponds to 99.989%
of the data.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major
frequency - H
INFO - (NP.df_utils._infer_frequency) - Major frequency H corresponds to 99.989%
of the data.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major
frequency - H
```

```
Predicting: 411it [00:00, ?it/s]
```

```
INFO - (NP.df_utils.return_df_in_original_format) - Returning df with no ID
column

NeuralProphet
 ------
 MAE: 25.192998244221446
 RMSE: 28.890365658724676
 MAPE: 0.11898955725680568
 R²: -0.09246319768814648
```



1.5.2 LSTM

Au regard de la complexité des données, il serait pertinent d'utiliser des modèles temporels plus sophistiqués, tels que les réseaux de neurones, notamment les LSTM, pour effectuer des prévisions à long terme.

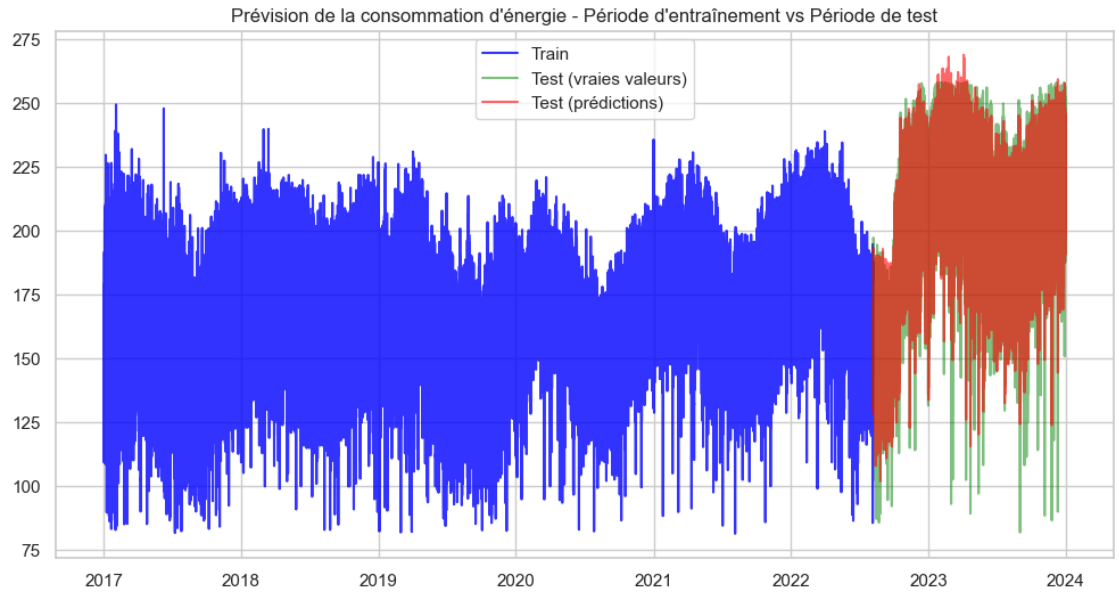
[24]:
```python
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

def create_dataset(data, look_back=1):
    X, y = [], []
    for i in range(len(data) - look_back - 1):
        a = data[i:(i + look_back)]
        X.append(a)
        y.append(data[i + look_back, 0])
    return np.array(X), np.array(y)
```

Entrainement du modèle LSTM sans variables exogènes dans un premier temps.

[25]:
```
# Normalisation
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(hourly_cleaned_data[['Power']])

# Data splitting
look_back = 24
X, y = create_dataset(scaled_data, look_back)
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], X_train.
 ↪shape[2]))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], X_test.shape[2]))

# Création du modèle LSTM
modele_lstm = Sequential()
modele_lstm.add(LSTM(50, activation='relu', input_shape=(look_back, X_train.
 ↪shape[2])))
modele_lstm.add(Dense(1))
modele_lstm.compile(optimizer='adam', loss='mean_absolute_error')
modele_lstm.fit(X_train, y_train, epochs=200, batch_size=32, verbose=0)

# Prévisions
y_pred = modele_lstm.predict(X_test)
y_pred = scaler.inverse_transform(y_pred)
y_test = scaler.inverse_transform(y_test.reshape(-1, 1))
mae_lstm = mean_absolute_error(y_test, y_pred)
rmse_lstm = mean_squared_error(y_test, y_pred, squared=False)
mape_lstm = mean_absolute_percentage_error(y_test, y_pred)
r2_lstm = r2_score(y_test, y_pred)
print(f"Long Short Term Memory - LSTM \n ------ \n MAE: {mae_lstm} \n RMSE:␣
 ↪{rmse_lstm} \n MAPE: {mape_lstm} \n R²: {r2_lstm}")

# Visualisation des résultats
plt.figure(figsize=(12, 6))
plt.plot(hourly_cleaned_data.index[:train_size],
         scaler.inverse_transform(scaled_data[:train_size, 0].reshape(-1, 1)),
         label='Train', color='blue', alpha=0.8)
plt.plot(hourly_cleaned_data.index[train_size:train_size + len(y_test)],
         y_test, label='Test (vraies valeurs)', color='green', alpha = 0.5)
plt.plot(hourly_cleaned_data.index[train_size:train_size + len(y_pred)],
         y_pred, label='Test (prédictions)', color='red', alpha=0.6)
plt.title('Prévision de la consommation d\'énergie - Période d\'entraînement vs␣
 ↪Période de test')
plt.legend()
```

```
plt.show()
```

```
384/384 [==============================] - 2s 5ms/step
Long Short Term Memory - LSTM
 ------
 MAE: 5.835236916475336
 RMSE: 9.49400067222835
 MAPE: 0.030600065049446623
 R²: 0.9164215079935419
```

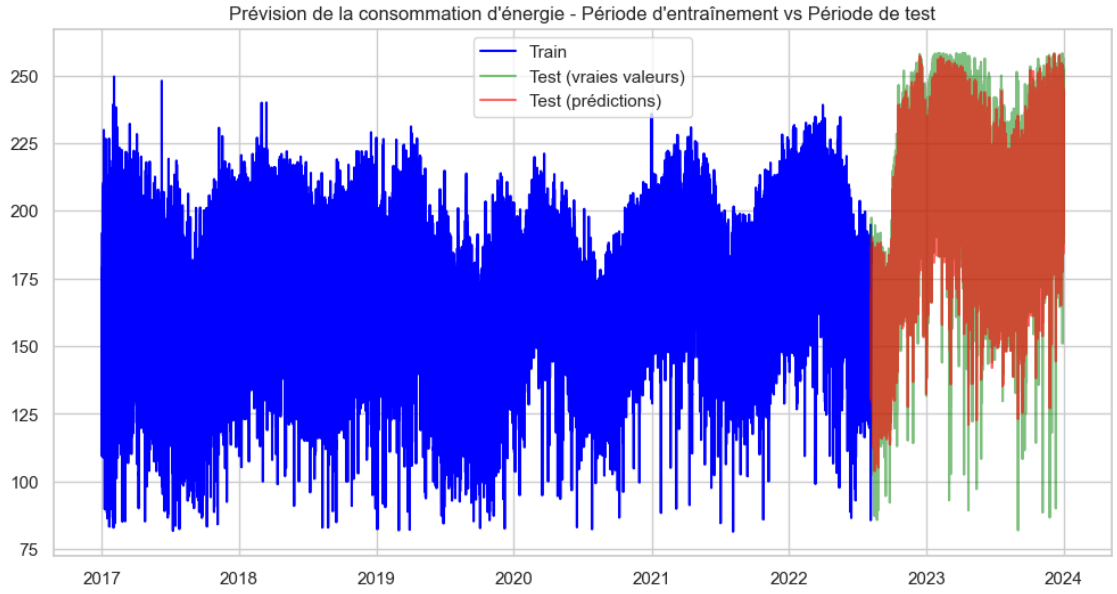


Entrainement du modèle avec toutes les variables exogènes crées et retenues après le **Feature Engineering**

[27]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
hourly_cleaned_data['Power_scaled'] = scaler.
 ↪fit_transform(hourly_cleaned_data[['Power']])
# Ajout des variables exogènes sans normalisation
scaled_data =␣
 ↪hourly_cleaned_data[['Power_scaled',"season_cluster","hour_of_day",
                                    "month_of_year","day_of_week","is_weekend",
                                    "is_holiday", "Power_cluster"]].values
look_back = 24
X, y = create_dataset(scaled_data, look_back)
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], X_train.
 ↪shape[2]))
```

```
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], X_test.shape[2]))

# modèle LSTM
modele_lstm_2 = Sequential()
modele_lstm_2.add(LSTM(50, activation='relu', input_shape=(look_back, X_train.
 ↪shape[2])))
modele_lstm_2.add(Dense(1))
modele_lstm_2.compile(optimizer='adam', loss='mean_absolute_error')
modele_lstm_2.fit(X_train, y_train, epochs=200, batch_size=32, verbose=0)
y_pred_2 = modele_lstm_2.predict(X_test)
y_pred_2 = scaler.inverse_transform(y_pred_2)
y_test_2 = scaler.inverse_transform(y_test.reshape(-1, 1))
mae_lstm_exovar = mean_absolute_error(y_test_2, y_pred_2)
rmse_lstm_exovar = mean_squared_error(y_test_2, y_pred_2, squared=False)
mape_lstm_exovar = mean_absolute_percentage_error(y_test_2, y_pred_2)
r2_lstm_exovar = r2_score(y_test_2, y_pred_2)
print(f"Long Short Term Memory - LSTM avec Variables exogènes \n ------ \n MAE:␣
 ↪{mae_lstm_exovar} \n RMSE: {rmse_lstm_exovar} \n MAPE: {mape_lstm_exovar} \n␣
 ↪R²: {r2_lstm_exovar}")
plt.figure(figsize=(12, 6))
plt.plot(hourly_cleaned_data.index[:train_size],
         scaler.inverse_transform(scaled_data[:train_size, 0].reshape(-1, 1)),
         label='Train', color='blue')
plt.plot(hourly_cleaned_data.index[train_size:train_size + len(y_test_2)],
         y_test_2, label='Test (vraies valeurs)', color='green', alpha = 0.5)
plt.plot(hourly_cleaned_data.index[train_size:train_size + len(y_pred_2)],
         y_pred_2, label='Test (prédictions)', color='red', alpha = 0.6)
plt.title('Prévision de la consommation d\'énergie - Période d\'entraînement vs␣
 ↪Période de test')
plt.legend()
plt.show()
```

```
384/384 [==============================] - 2s 4ms/step
Long Short Term Memory - LSTM avec Variables exogènes
 ------
 MAE: 6.165404121751082
 RMSE: 9.430227439791162
 MAPE: 0.03139722184320179
 R²: 0.9175405660532493
```



On peut constater que le modèle LSTM prévoit mieux les valeurs en l'absence de variables exogènes. Cela est probablement dû à un problème d'overfitting ou de biais présents dans les variables introduites dans le modèle.

Aperçu des performances des modèles entrainés

[30]:
```
model_perform_summary = pd.DataFrame({'MAE' : [mae_arima, mae_gam, mae_smooth,␣
↪mae_np, mae_lstm ],
                        'RMSE':[rmse_arima, rmse_gam, rmse_smooth, rmse_np,␣
↪rmse_lstm ] ,
                        'MAPE' : [mape_arima, mape_gam, mape_smooth, mape_np,␣
↪mape_lstm ],
                        'R²' : [r2_arima, r2_gam, r2_smooth, r2_np, r2_lstm ]
                       }, index = ['ARIMA', 'GAM', 'Exponetial Smoothing',␣
↪'Neural Prophet', 'LSTM'])
model_perform_summary.sort_values(ascending=False, by="MAE", inplace=True)
model_perform_summary
```

[30]:

	MAE	RMSE	MAPE	R²
GAM	46.947071	55.461166	0.246980	-3.026046
Neural Prophet	25.192998	28.890366	0.118990	-0.092463
ARIMA	22.805375	27.656944	0.115242	-0.001173
Exponetial Smoothing	21.583080	27.078617	0.103978	0.040260
LSTM	5.835237	9.494001	0.030600	0.916422

Conclusion

Après une analyse comparative des métriques des modèles entrainés, il ressort que le modèle LSTM est le meilleur

1.5.3 Sauvegarde du modèle sélectionné

[ ]:
```
import joblib
modele_lstm.save('modele_lstm.keras')
joblib.dump(scaler, 'scaler.pkl')
```

1.6 VI- Post processing

Création d'un DataFrame constitué des prédictions du modele et des valeurs réelles

[54]:
```
results_df = pd.DataFrame({
    'Date': hourly_cleaned_data.index[train_size:train_size + len(y_test)],
    'y_test': y_test.flatten(),
    'y_pred': y_pred.flatten(),
    'Mean_absolute_error' :  y_test.flatten() - y_pred.flatten()
})
results_df
```

[54]:

	Date	y_test	y_pred	Mean_absolute_error
0	2022-08-06 23:00:00	170.175	169.860229	0.314771
1	2022-08-07 00:00:00	157.775	159.553940	-1.778940
2	2022-08-07 01:00:00	141.206	146.014847	-4.808847
3	2022-08-07 02:00:00	138.359	131.728104	6.630896
4	2022-08-07 03:00:00	135.725	130.606400	5.118600
…	…	…	…	…
12259	2023-12-30 18:00:00	212.140	221.539886	-9.399886
12260	2023-12-30 19:00:00	233.900	229.518448	4.381552
12261	2023-12-30 20:00:00	243.220	250.171799	-6.951799
12262	2023-12-30 21:00:00	252.540	252.505829	0.034171
12263	2023-12-30 22:00:00	252.740	260.056183	-7.316183

[12264 rows x 4 columns]

Utilisation du modèle de regression linéaire

[56]:
```
from sklearn.linear_model import LinearRegression
from scipy.signal import savgol_filter
```

[57]:
```
X_error = results_df[['y_pred']]
y_error = results_df['Mean_absolute_error']
```

[58]:
```
# Entraînement du modèle de post-processing
error_model = LinearRegression()
error_model.fit(X_error, y_error)

# Prédiction de l'erreur
error_predictions = error_model.predict(results_df[['y_pred']])
```

```
# Ajustement des prédictions
adjusted_predictions = results_df['y_pred'] + error_predictions
results_df['y_pred_adjusted'] = adjusted_predictions

# Application du filtre de Savitzky-Golay sur les prédictions ajustées
smoothed_predictions = savgol_filter(results_df['y_pred_adjusted'],␣
 ↪window_length=5, polyorder=2)
results_df['y_pred_smoothed'] = smoothed_predictions
print("Adjusted mae (LinearRegression) : ",␣
 ↪mean_absolute_error(results_df['y_test'], results_df['y_pred_adjusted']))
print("Smoothed mae (LinearRegression) : ",␣
 ↪mean_absolute_error(results_df['y_test'], results_df['y_pred_smoothed']))
```

```
Adjusted mae (LinearRegression) :  6.918399272775867
Smoothed mae (LinearRegression) :  6.07279443000616
```

Utitilisation d'un modele plus avancé tel que catboost pour une meilleur prédictin

[59]:
```
from catboost import CatBoostRegressor
```

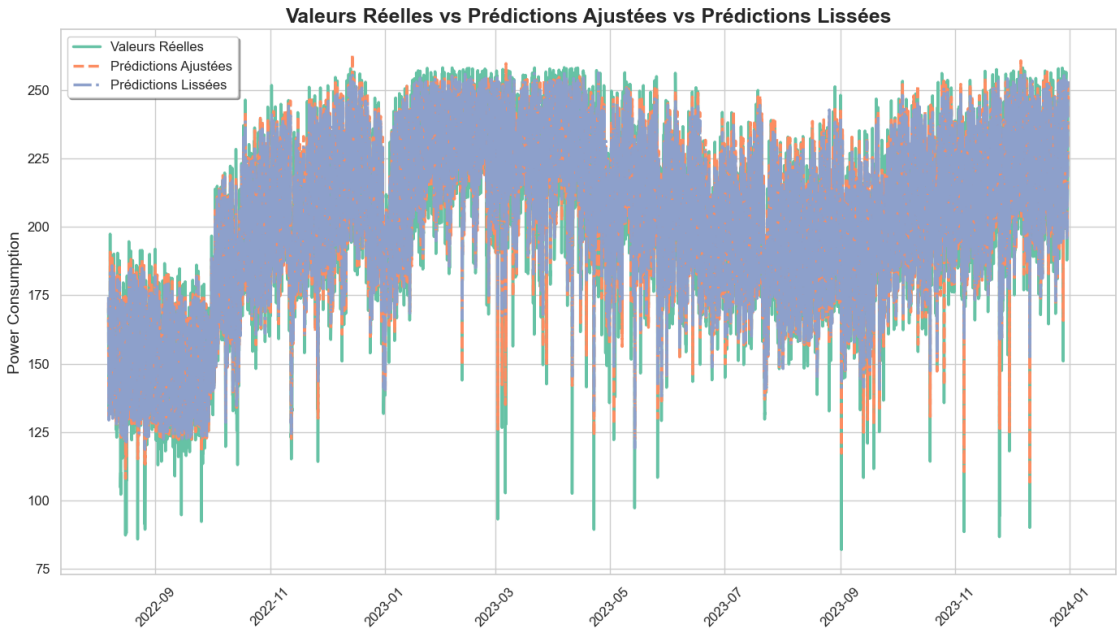
[60]:
```
error_model = CatBoostRegressor()
error_model.fit(X_error, y_error, verbose=100)
error_predictions = error_model.predict(results_df[['y_pred']])
adjusted_predictions = results_df['y_pred'] + error_predictions
results_df['y_pred_adjusted'] = adjusted_predictions
smoothed_predictions = savgol_filter(results_df['y_pred_adjusted'],␣
 ↪window_length=5, polyorder=2)
results_df['y_pred_smoothed'] = smoothed_predictions
print("Adjusted mae (CatBoostRegressor) : ",␣
 ↪mean_absolute_error(results_df['y_test'], results_df['y_pred_adjusted']))
print("Smoothed mae (CatBoostRegressor) :",␣
 ↪mean_absolute_error(results_df['y_test'], results_df['y_pred_smoothed']))
```

```
Learning rate set to 0.060839
0:      learn: 10.6275269       total: 184ms    remaining: 3m 3s
100:    learn: 10.4466637       total: 1.02s    remaining: 9.05s
200:    learn: 10.4229323       total: 1.74s    remaining: 6.91s
300:    learn: 10.4003232       total: 2.45s    remaining: 5.69s
400:    learn: 10.3844028       total: 3.4s     remaining: 5.08s
500:    learn: 10.3752591       total: 4.16s    remaining: 4.14s
600:    learn: 10.3691164       total: 5s       remaining: 3.32s
700:    learn: 10.3650229       total: 5.84s    remaining: 2.49s
800:    learn: 10.3621456       total: 6.69s    remaining: 1.66s
900:    learn: 10.3598024       total: 7.45s    remaining: 819ms
999:    learn: 10.3578016       total: 8.2s     remaining: 0us
Adjusted mae (CatBoostRegressor) :  6.737278357724498
Smoothed mae (CatBoostRegressor) : 5.9389204558009405
```

[ ]:
```
# Sauvegarder du modele catboost pour la correction des erreurs
joblib.dump(error_model, 'error_catboost_model.pkl')
```

[61]:
```
print(" Mean Absolute Error finale : ",␣
 ↪mean_absolute_error(results_df['y_test'], results_df['y_pred_smoothed']))
palette = sns.color_palette("Set2", 3)
plt.figure(figsize=(14, 8))
plt.plot(results_df['Date'], results_df['y_test'], label='Valeurs Réelles',␣
 ↪color=palette[0], linewidth=2.5)
plt.plot(results_df['Date'], results_df['y_pred_adjusted'], label='Prédictions␣
 ↪Ajustées', color=palette[1], linestyle='--', linewidth=2.5)
plt.plot(results_df['Date'], results_df['y_pred_smoothed'], label='Prédictions␣
 ↪Lissées', color=palette[2], linestyle='-.', linewidth=2.5)
plt.title('Valeurs Réelles vs Prédictions Ajustées vs Prédictions Lissées',␣
 ↪fontsize=18, fontweight='bold')
plt.ylabel('Power Consumption', fontsize=14)
plt.legend(loc='best', fontsize=12, frameon=True, shadow=True, fancybox=True)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.show()
```

```
Mean Absolute Error finale :  5.9389204558009405
```



FIN