Minor Project Report
“Prediction of behavior of Prosumers using Machine Learning”

Under the Guidance
of
Dr. Monika Bansal
&
Dr. Bhawna Rawat

Submitted By:

Krishna Goel, 02717711621, AIML-A

Anant Chauhan, 01517711621, AIML-A

Akanksha, 08017711621, AIML-B

DECLARATION

This is to certify that Minor Project Report titled "**Prediction of behavior of Prosumers using Machine Learning**",

for the Minor Project with AIDS451/AIML451/IIoT451 is submitted by **Krishna Goel (02717711621), Anant Chauhan (01517711621), Akanksha (08017711621)** in partial fulfillment of the requirement for the award of degree B.Tech. in Artificial Intelligence and Machine Learning/ Artificial Intelligence and Data Science/ Industrial Internet of Things, VIPS-TC, GGSIP University, Dwarka, Delhi. It comprises of our original work. The due affirmation has been made within the report for utilizing the referenced work.

Date:

Signature (Krishna Goel, 02717711621)

Signature (Anant Chauhan, 01517711621)

Signature (Akanksha, 08017711621)

Certificate by Supervisor

This is to certify that Minor Project Report titled "**Prediction of behavior of Prosumers using Machine Learning**" for the Minor Project with AIDS451/AIML451/IIoT451 is submitted by **Krishna Goel (02717711621), Anant Chauhan (01517711621), Akanksha (08017711621)** in partial fulfillment of the requirement for the award of degree B.Tech in Artificial Intelligence and Machine Learning/ Artificial Intelligence and Data Science/ Industrial Internet of Things, VIPS-TC, GGSIP University, Dwarka, Delhi. It is a record of the candidates own work carried out by them under my supervision. The matter embodied in this Report is original and has not been submitted for the award of any other degree.

Date: **(Signature)**

SUPERVISOR<NAME, DEPARTMENT>

<Signature of HOD> **<Signature of Programme Coordinator>**

Acknowledgment

We would like to express my heartfelt gratitude to all those who have contributed to the successful completion of our minor project report. First and foremost, we would like to extend our deepest appreciation to our supervisor, [Supervisor's Name], for their constant guidance, valuable insights, and unwavering support throughout this journey. We are also indebted to the staff and faculty members of VIPS-TC, GGSIPU, Dwarka, Delhi, whose expertise and cooperation were instrumental in enhancing our learning experience. Additionally, we are grateful to our fellow students who provided us with invaluable feedback and encouragement.

(Signature of the students)

Table of Contents

Introduction

The growing adoption of renewable energy sources, especially solar power, has led to the rise of energy prosumers—individuals who both produce and consume energy. This shift in the energy landscape presents new challenges for predicting energy production and consumption patterns, which are influenced by various factors such as weather conditions and energy prices. Accurately predicting these patterns is crucial for minimizing imbalance costs in energy grids and ensuring optimal energy utilization.

The global energy landscape is shifting towards decentralization, largely due to the increasing adoption of renewable energy sources like solar and wind power. This shift has led to the rise of "prosumers" — entities that not only consume energy but also produce it using installed renewable energy systems, such as solar panels. However, this prosumer behavior introduces significant variability in energy production and consumption, which poses challenges for energy grid operators in maintaining balance and minimizing associated costs. Predicting energy consumption and production accurately is crucial for grid stability, energy cost reduction, and optimizing the energy trade between prosumers and the grid.

Energy prosumers are individuals, businesses, or organizations that both consume and produce energy. This concept represents a shift from the traditional model where consumers simply purchase energy from utilities and rely on centralized power generation sources. Energy prosumers are actively involved in the energy ecosystem by generating their own electricity, typically through renewable energy sources like solar panels (or wind turbines, small-scale hydropower etc.). They also consume energy from the grid when their own generation is insufficient to meet their needs

This project aims to develop a forecasting model that can predict the energy behavior of Estonian prosumers who have installed solar panels. The availability of large datasets, including weather conditions, photovoltaic capacities, and energy prices, provides a solid foundation for building an accurate forecasting model.

Related Work

The related work highlights previous studies and competitions that have informed the project's approach to forecasting energy consumption and production:

- **Global Energy Forecasting Competitions:**
  - The Global Energy Forecasting Competition 2012 and other similar competitions have played a significant role in advancing methods for predicting energy patterns. These competitions have underscored the importance of using time-series data for forecasting tasks.
  - Insights gained from these events emphasize the relevance of incorporating factors such as weather data and seasonal patterns into predictive models to enhance accuracy.
- **Research Studies on Energy Forecasting:**
  - Prior studies have demonstrated the value of machine learning techniques for energy forecasting. The use of time-series methods, coupled with machine learning algorithms, has shown improvements in predicting energy consumption and production.
  - Specific techniques, such as the semi-parametric additive model for short-term load forecasting, have been documented to yield accurate results in studies published in journals like the IEEE Transactions on Power Systems.
- **Machine Learning Approaches:**
  - Algorithms such as XGBoost, known for its speed and ability to handle large datasets with missing values, have been cited as effective for energy forecasting.
  - The combination of tree-based methods with other techniques to handle non-linearities in the data has been found to improve model performance, especially in cases where renewable energy sources like solar are involved.

This provides a foundation for the current project, demonstrating the effectiveness of machine learning models, the incorporation of time-series data, and the consideration of external variables like weather in forecasting energy usage patterns.

Problem Statement & Objectives

Problem Statement

The problem this project addresses is the challenge of accurately predicting the energy behavior of prosumers—those who produce energy via solar panels while consuming energy from the grid. The unpredictable nature of weather conditions, combined with fluctuating energy prices, makes it difficult to balance energy supply and demand. Traditional methods struggle to accurately predict hourly energy production and consumption, leading to increased imbalance costs. The goal is to use machine learning techniques to create models that can predict these energy patterns with higher accuracy.

The primary objective of this project is to predict the energy consumption and production of prosumers in Estonia who have installed solar panels. By analyzing the available data — including weather patterns, energy prices, and photovoltaic capacities — the project aims to develop a robust forecasting model that can accurately predict energy patterns at hourly intervals. The key challenge lies in handling the inherent variability in renewable energy production due to factors like weather changes and the non-linear nature of prosumer behavior.

The number of prosumers is rapidly increasing, associated with higher energy imbalance - increased operational costs, potential grid instability, and inefficient use of energy resources.

The goal of the competition is to create an energy prediction model of prosumers to reduce energy imbalance costs. If solved, it would reduce the imbalance costs, improve the reliability of the grid, and make the integration of prosumers into the energy system more efficient and sustainable. Moreover, it could potentially incentivize more consumers to become prosumers and thus promote renewable energy production and use.

Objectives

- **Data Availability:** The dataset provided by the competition is rich in features, including weather data, energy prices, and photovoltaic capacity, making it feasible to build a predictive model. The data is time-stamped on an hourly basis, which is ideal for time-series forecasting.
- **Machine Learning Tools:** The availability of libraries such as XGBoost, Scikit-learn, and Pandas in Python makes the task of building and testing machine learning models feasible. XGBoost, in particular, is well-suited for this task because of its speed, accuracy, and handling of time-series data with missing values.
- **Computational Resources:** Modern personal computers with at least 16GB of RAM and a multi-core processor should be sufficient to handle the dataset and run the model. For faster computation and hyperparameter tuning, *GPU acceleration* can be leveraged.
- **Feasibility of Implementation:** Given the availability of well-documented libraries and tools, combined with the structured dataset, this project is practically feasible. The primary challenge will be tuning the machine learning model to achieve optimal accuracy, but this is manageable through iterative testing and cross-validation
- A feasibility study confirms that similar projects using weather and energy data have led to significant improvements in grid management and energy cost reduction. By predicting energy usage and production, grid operators can better plan for energy demand and supply imbalances, resulting in more efficient energy distribution. Given the rich dataset and the availability of machine learning tools, this project is both feasible and impactful.

Project Analysis and Design

Hardware Requirements:

- **Computer:** A computer with a modern processor (Intel i5 or equivalent) and at least 8GB of RAM to run SQL and Power BI efficiently.
- **Storage:** Adequate storage capacity (minimum 256GB SSD) to handle temporary files and project data.

Software requirements:

- **Programming Language:**
    - Python 3.x: Python will be the primary language used for data analysis, model development, and evaluation.
- **Libraries and Dependencies:**
    - **XGBoost:** The primary library for model implementation
    - **Pandas and NumPy:** For data manipulation and handling
    - **Scikit-learn:** For data preprocessing and model evaluation
    - **Matplotlib and Seaborn:** For data visualization and feature analysis
    - **Jupyter Notebook/Google Colab/Kaggle Notebooks:** For interactive development and testing.
    - **Kaggle API:** For data access and leaderboard submission.
- **Development Environment:**
    - **Anaconda (optional):** A Python distribution with pre-installed libraries
    - **Kaggle Notebooks or Google Colab:** for cloud-based computation, eliminating the need for high-end local hardware

Proposed work and Methodology Adopted

Proposed Work:

The project aims to develop a machine learning-based forecasting model that predicts the energy consumption and production patterns of prosumers in Estonia who use solar panels. The goal is to improve the accuracy of hourly energy predictions by leveraging a variety of factors such as weather conditions, energy prices, and photovoltaic capacities. The model will help in balancing energy supply and demand more efficiently, thereby reducing imbalance costs for energy grid operators.

Methodology Adopted:

- **Data Collection and Preprocessing:**
  - The primary dataset used for this project contains hourly time-stamped data, which includes weather conditions, energy prices, and photovoltaic capacity information.
  - Data preprocessing steps involve cleaning the dataset, handling missing values, and transforming the data into a suitable format for time-series analysis.
- **Feature Engineering:**
  - Important features, such as weather parameters (temperature, solar radiation, cloud cover), historical energy consumption, and production values, are extracted and used to enhance the model's predictive power.
  - Time-dependent features like hour of the day, day of the week, and seasonality are included to capture periodic variations in energy patterns.
- **Model Selection and Training:**
  - XGBoost is chosen as the primary algorithm due to its efficiency and accuracy in handling large datasets with non-linear relationships.
  - Other machine learning techniques, such as Random Forests and Support Vector Machines (SVM), may also be considered for comparison to ensure the best possible performance.

- **Hyperparameter Tuning:**
    - The model's hyperparameters are optimized through techniques like grid search or random search to improve prediction accuracy.
    - Cross-validation is employed during this process to ensure the model generalizes well to unseen data.
- **Model Evaluation and Testing:**
    - The model's performance is assessed using metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared to measure its accuracy in predicting energy consumption and production.
    - The evaluation involves testing the model on a separate test dataset to measure its real-world applicability.
- **Deployment and Implementation:**
    - Once the model is fine-tuned and tested, it can be deployed for practical use in predicting the energy behavior of prosumers.
    - The predictions can help energy grid operators plan for energy demand and supply more efficiently, thus reducing costs associated with grid imbalances.

Conclusion

This project demonstrates the feasibility of using machine learning to predict the energy consumption and production behaviors of prosumers who rely on renewable energy sources, specifically solar power. By leveraging a robust dataset with 59 features—such as 'county,' 'is_business,' 'product_type,' 'installed_capacity_client,' weather-related attributes (e.g., 'temperature_h_mean,' 'cloudcover_total_f_mean,' 'direct_solar_radiation_h_mean'), energy prices (e.g., 'euros_per_mwh_electricity'), and past energy consumption patterns (e.g., 'target_2_days_ago' to 'target_15_days_ago')—the model shows promise in accurately forecasting hourly energy patterns.

Through iterative model tuning and validation, early stopping was applied at the best iteration #202, achieving a Mean Absolute Error (MAE) of 92.59 and a Mean Absolute Percentage Error (MAPE) of 6.16% on the validation set. These metrics indicate a high level of accuracy, supporting the model's effectiveness in predicting energy behaviors.

This predictive capability is essential for reducing imbalance costs and enhancing grid stability as the energy grid becomes increasingly decentralized with the rise of renewable energy prosumers.

The chosen methodology, which includes comprehensive feature engineering and time-series forecasting techniques, effectively addresses non-linear relationships and the impact of external variables on energy production and consumption. While challenges remain, including handling data variability and ensuring model generalizability, the results underscore the value of machine learning in supporting sustainable energy grid management.

Future scope of Work

The project has shown promising results in forecasting energy consumption and production for prosumers using solar panels, but there are several avenues for further development and enhancements.

- **Incorporation of Additional Data Sources:**
  - Integrating more data sources, such as real-time energy price forecasts, demand response signals, and energy storage data, could enhance the model's accuracy and applicability.
  - Including more granular weather data, such as localized temperature, humidity, wind speed, and cloud cover variations, can improve the precision of energy production forecasts, particularly for solar power generation.
- **Real-Time Forecasting and Dynamic Model Updates:**
  - Implementing a real-time forecasting system that continuously updates predictions based on new incoming data could make the model more adaptive to changing conditions.
  - Techniques like online learning, where the model is updated incrementally as new data arrives, can be explored to maintain accuracy without needing to retrain the model from scratch.
- **Scaling the Model for Larger Geographical Areas:**
  - Expanding the model to forecast energy patterns for larger regions, including multiple cities or countries, would be valuable for regional grid management and planning.
  - This would involve handling more diverse data sources and dealing with the increased complexity of energy patterns across different geographical locations.
- **Incorporation of Energy Storage Systems and Electric Vehicles:**
  - Future models could consider the impact of energy storage systems (e.g., batteries) and electric vehicles (EVs) on energy consumption and production patterns. Predicting when prosumers charge their batteries or EVs could refine forecasts and optimize grid management.

- Integration with Smart Grid Technologies:
    - Integrating the model with smart grid technologies for real-time monitoring and automated control can enable more efficient demand response strategies and enhance grid stability.
    - The model could be used to provide insights for load balancing, peak shaving, and demand-side management in smart grid environments.

References

1. Hong, T., Pinson, P., & Fan, S. (2016). "Global Energy Forecasting Competition 2012 and beyond." International Journal of Forecasting, 32(3), 596-608. DOI: [10.1016/j.ijforecast.2015.11.014](https://doi.org/10.1016/j.ijforecast.2015.11.014)

2. Fan, S., & Hyndman, R. J. (2012). "Short-term load forecasting based on a semi-parametric additive model." IEEE Transactions on Power Systems, 27(1), 134-141. DOI:[10.1109/TPWRS.2011.2162082](https://doi.org/10.1109/TPWRS.2011.21620.

3. Chen, T., & Guestrin, C. (2016). "XGBoost: A scalable tree boosting system." In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge and Data Mining, pp. [10.1145/2939672.2939785](https://doi.org/10.1145/2939672.2939785) 785-794.DOI:

4. Li, L., Yu, J., & Yang, Z. (2015). "Renewable energy forecasting based on time-series method and machine learning." Renewable and Sustainable Energy Reviews, 52, 273-284. DOI: [10.1016/j.rser.2015.07.078](https://doi.org/10.1016/j.rser.2015.07.078)

Code

```
!pip install xgboost -U
!pip install colorama
#General
import pandas as pd
import numpy as np
import json
# Visualization
import seaborn as sns
import matplotlib.pyplot as plt
from colorama import Fore, Style, init;

# Modeling
import xgboost as xgb
import lightgbm as lgb
import torch

# Geolocation
from geopy.geocoders import Nominatim

# Options
pd.set_option('display.max_columns', 100)
DEBUG = False
if torch.cuda.is_available():
    device = 'cuda'
else:
    device = 'cpu'
# Helper functions
def display_df(df, name):
    '''Display df shape and first row '''
    PrintColor(text = f'{name} data has {df.shape[0]} rows and
{df.shape[1]} columns. \n ===> First row:')
    display(df.head(1))

# Color printing
def PrintColor(text:str, color = Fore.BLUE, style = Style.BRIGHT):
    '''Prints color outputs using colorama of a text string'''
    print(style + color + text + Style.RESET_ALL);
# Read CSVs and parse relevant date columns
train = pd.read_csv("/content/train.csv")
client = pd.read_csv("/content/client.csv")
historical_weather = pd.read_csv("/content/historical_weather.csv")
forecast_weather = pd.read_csv("/content/forecast_weather.csv")
electricity = pd.read_csv("/content/electricity_prices.csv")
gas = pd.read_csv("/content/gas_prices.csv")
# Location from https://www.kaggle.com/datasets/michaelo/fabiendaniels-
mapping-locations-and-county-codes/data
location = (pd.read_csv("/content/county_lon_lats.csv")
            .drop(columns = ["Unnamed: 0"])
           )
display_df(train, 'train')
display_df(client, 'client')
display_df(historical_weather, 'historical weather')
display_df(forecast_weather, 'forecast weather')
```

```
display_df(electricity, 'electricity prices')
display_df(gas, 'gas prices')
display_df(location, 'location data')
import os
import json
import pandas as pd

DATA_DIR = '/kaggle/input/predict-energy-behavior-of-prosumers/'

# Check if the file exists
file_path = os.path.join(DATA_DIR, 'county_id_to_name_map.json')
if not os.path.exists(file_path):
    # If not found in the default location, try searching in the current
directory
    file_path = 'county_id_to_name_map.json'
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"The file 'county_id_to_name_map.json'
was not found in '{DATA_DIR}' or the current directory.")

# See county codes
with open(file_path) as f:
    county_codes = json.load(f)
pd.DataFrame(county_codes, index=[0])
class FeatureProcessorClass():
    def __init__(self):
        # Columns to join on for the different datasets
        self.weather_join = ['datetime', 'county', 'data_block_id']
        self.gas_join = ['data_block_id']
        self.electricity_join = ['datetime', 'data_block_id']
        self.client_join = ['county', 'is_business', 'product_type',
'data_block_id']

        # Columns of latitude & longitude
        self.lat_lon_columns = ['latitude', 'longitude']

        # Aggregate stats
        self.agg_stats = ['mean'] #, 'min', 'max', 'std', 'median']

        # Categorical columns (specify for XGBoost)
        self.category_columns = ['county', 'is_business', 'product_type',
'is_consumption', 'data_block_id']

    def create_new_column_names(self, df, suffix, columns_no_change):
        '''Change column names by given suffix, keep columns_no_change,
and return back the data'''
        df.columns = [col + suffix
                      if col not in columns_no_change
                      else col
                      for col in df.columns
                      ]
        return df

    def flatten_multi_index_columns(self, df):
        df.columns = ['_'.join([col for col in multi_col if len(col)>0])
                      for multi_col in df.columns]
        return df
```

```
    def create_data_features(self, data):
        '''📊Create features for main data (test or train) set📊'''
        # To datetime
        data['datetime'] = pd.to_datetime(data['datetime'])

        # Time period features
        data['date'] = data['datetime'].dt.normalize()
        data['year'] = data['datetime'].dt.year
        data['quarter'] = data['datetime'].dt.quarter
        data['month'] = data['datetime'].dt.month
        data['week'] = data['datetime'].dt.isocalendar().week
        data['hour'] = data['datetime'].dt.hour

        # Day features
        data['day_of_year'] = data['datetime'].dt.day_of_year
        data['day_of_month']  = data['datetime'].dt.day
        data['day_of_week'] = data['datetime'].dt.day_of_week
        return data

    def create_client_features(self, client):
        '''💼 Create client features 💼'''
        # Modify column names - specify suffix
        client = self.create_new_column_names(client,
                                              suffix='_client',
                                              columns_no_change =
self.client_join
                                             )
        return client

    def create_historical_weather_features(self, historical_weather):
        '''⏳🌤️ Create historical weather features 🌤️⏳'''

        # To datetime
        historical_weather['datetime'] =
pd.to_datetime(historical_weather['datetime'])

        # Add county
        historical_weather[self.lat_lon_columns] =
historical_weather[self.lat_lon_columns].astype(float).round(1)
        historical_weather = historical_weather.merge(location, how =
'left', on = self.lat_lon_columns)

        # Modify column names - specify suffix
        historical_weather =
self.create_new_column_names(historical_weather,
                                                     suffix='_h',
                                                     columns_no_chang
e = self.lat_lon_columns + self.weather_join
                                                     )

        # Group by & calculate aggregate stats
        agg_columns = [col for col in historical_weather.columns if col
not in self.lat_lon_columns + self.weather_join]
        agg_dict = {agg_col: self.agg_stats for agg_col in agg_columns}
        historical_weather =
historical_weather.groupby(self.weather_join).agg(agg_dict).reset_index()
```

```
        # Flatten the multi column aggregates
        historical_weather =
self.flatten_multi_index_columns(historical_weather)

        # Test set has 1 day offset for hour<11 and 2 day offset for
hour>11
        historical_weather['hour_h'] =
historical_weather['datetime'].dt.hour
        historical_weather['datetime'] = (historical_weather
                                                .apply(lambda x:
                                                       x['datetime'] +
pd.DateOffset(1)
                                                       if x['hour_h']< 11
                                                       else x['datetime'] +
pd.DateOffset(2),
                                                       axis=1)
                                                )

        return historical_weather

    def create_forecast_weather_features(self, forecast_weather):
        '''🔮🌤️ Create forecast weather features 🌤️🔮'''

        # Rename column and drop
        forecast_weather = (forecast_weather
                            .rename(columns = {'forecast_datetime':
'datetime'})
                            .drop(columns = 'origin_datetime') # not
needed
                           )

        # To datetime
        forecast_weather['datetime'] =
(pd.to_datetime(forecast_weather['datetime'])
                                        .dt
                                        .tz_localize(None)
                                       )

        # Add county
        forecast_weather[self.lat_lon_columns] =
forecast_weather[self.lat_lon_columns].astype(float).round(1)
        forecast_weather = forecast_weather.merge(location, how = 'left',
on = self.lat_lon_columns)

        # Modify column names - specify suffix
        forecast_weather = self.create_new_column_names(forecast_weather,
                                                        suffix='_f',
                                                        columns_no_change
= self.lat_lon_columns + self.weather_join
                                                        )

        # Group by & calculate aggregate stats
        agg_columns = [col for col in forecast_weather.columns if col not
in self.lat_lon_columns + self.weather_join]
        agg_dict = {agg_col: self.agg_stats for agg_col in agg_columns}
```

```
        forecast_weather =
forecast_weather.groupby(self.weather_join).agg(agg_dict).reset_index()

        # Flatten the multi column aggregates
        forecast_weather =
self.flatten_multi_index_columns(forecast_weather)
        return forecast_weather

    def create_electricity_features(self, electricity):
        ''' ⚡ Create electricity prices features ⚡ '''
        # To datetime
        electricity['forecast_date'] =
pd.to_datetime(electricity['forecast_date'])

        # Test set has 1 day offset
        electricity['datetime'] = electricity['forecast_date'] +
pd.DateOffset(1)

        # Modify column names - specify suffix
        electricity = self.create_new_column_names(electricity,
                                                   suffix='_electricity',
                                                   columns_no_change =
self.electricity_join
                                                  )
        return electricity

    def create_gas_features(self, gas):
        ''' ⛽ Create gas prices features ⛽ '''
        # Mean gas price
        gas['mean_price_per_mwh'] = (gas['lowest_price_per_mwh'] +
gas['highest_price_per_mwh'])/2

        # Modify column names - specify suffix
        gas = self.create_new_column_names(gas,
                                           suffix='_gas',
                                           columns_no_change =
self.gas_join
                                          )
        return gas

    def __call__(self, data, client, historical_weather, forecast_weather,
electricity, gas):
        '''Processing of features from all datasets, merge together and
return features for dataframe df '''
        # Create features for relevant dataset
        data = self.create_data_features(data)
        client = self.create_client_features(client)
        historical_weather =
self.create_historical_weather_features(historical_weather)
        forecast_weather =
self.create_forecast_weather_features(forecast_weather)
        electricity = self.create_electricity_features(electricity)
        gas = self.create_gas_features(gas)

        # 🔗 Merge all datasets into one df 🔗
        df = data.merge(client, how='left', on = self.client_join)
```

```
        df = df.merge(historical_weather, how='left', on =
self.weather_join)
        df = df.merge(forecast_weather, how='left', on =
self.weather_join)
        df = df.merge(electricity, how='left', on = self.electricity_join)
        df = df.merge(gas, how='left', on = self.gas_join)

        # Change columns to categorical for XGBoost
        df[self.category_columns] =
df[self.category_columns].astype('category')
        return df
def create_revealed_targets_train(data, N_day_lags):
    '''🎯 Create past revealed_targets for train set based on number of
day lags N_day_lags 🎯 '''
    original_datetime = data['datetime']
    revealed_targets = data[['datetime', 'prediction_unit_id',
'is_consumption', 'target']].copy()

    # Create revealed targets for all day lags
    for day_lag in range(2, N_day_lags+1):
        revealed_targets['datetime'] = original_datetime +
pd.DateOffset(day_lag)
        data = data.merge(revealed_targets,
                          how='left',
                          on = ['datetime', 'prediction_unit_id',
'is_consumption'],
                          suffixes = ('', f'_{day_lag}_days_ago')
                         )
    return data
df
# Remove columns for features
no_features = ['date',
               'latitude',
               'longitude',
               'data_block_id',
               'row_id',
               'hours_ahead',
               'hour_h',
              ]

remove_columns = [col for col in df.columns for no_feature in no_features
if no_feature in col]
remove_columns.append(target)
features = [col for col in df.columns if col not in remove_columns]
PrintColor(f'There are {len(features)} features: {features}')
#### Create single fold split ######
# Remove empty target row
target = 'target'
df = df[df[target].notnull()].reset_index(drop=True)

train_block_id = list(range(0, 600))

tr = df[df['data_block_id'].isin(train_block_id)] # first 600
data_block_ids used for training
val = df[~df['data_block_id'].isin(train_block_id)] # rest data_block_ids
used for validation
```

```
clf.fit(X = tr[features],
        y = tr[target],
        eval_set = [(tr[features], tr[target]), (val[features],
val[target])],
        verbose=True #False #True
       )
PrintColor(f'Early stopping on best iteration #{clf.best_iteration} with
MAE error on validation set of {clf.best_score:.2f}')
debug = False
n_rows_debug = 100000
import pandas as pd
import numpy as np
from datetime import date, datetime, timedelta
import matplotlib.pyplot as plt
import math
import collections
import time
from copy import deepcopy
import seaborn as sns
import os, gc
from tqdm import tqdm
import re

""" SCIKIT-LEARN """
from sklearn import preprocessing
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix, classification_report,
mean_squared_error
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, confusion_matrix,
accuracy_score, mean_squared_error, roc_auc_score, roc_curve
from sklearn.model_selection import GroupKFold, StratifiedKFold, KFold,
TimeSeriesSplit
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestRegressor

from collections import Counter, defaultdict

from xgboost import XGBRegressor
from catboost import CatBoostRegressor
import lightgbm as lgb
import joblib

pd.set_option('display.float_format', lambda x: '%.5f' % x) # No scientic
notation
pd.set_option('display.max_columns', 100)
train = pd.read_csv("/content/train.csv")
if debug :
    print("DEBUG IS ON !")
    train = train.sample(n=n_rows_debug, random_state=12)

# Show
print(train.shape)
train.head(2)
# Print datetimes in the train dataset (2021-09-01 -> 2023-05-31)
train['datetime'].value_counts().to_frame().sort_index()
```

```
# Show target distribution
display(train['target'].describe())
train['target'].hist(bins=100)
electricity_prices = pd.read_csv("/content/electricity_prices.csv")

# Date processing + features extraction
for date_col in ['forecast_date', 'origin_date'] :
    electricity_prices = process_date(electricity_prices, date_col)

# Show
print(electricity_prices.shape)
electricity_prices.head(2)
gas_prices = pd.read_csv("/content/gas_prices.csv")

# Date processing + features extraction
for date_col in ['forecast_date', 'origin_date'] :
    gas_prices = process_date(gas_prices, date_col)

# Show
print(gas_prices.shape)
gas_prices.head(2)
client = pd.read_csv("/content/client.csv")

# Rename
client = client.rename(columns = {'date' : 'datetime'})

# Date processing + features extraction
for date_col in ['datetime'] :
    client = process_date(client, date_col)

# Fill NaN
client.fillna(0, inplace=True)

# Show
print(client.shape)
client.head(2)
location = pd.read_csv("/content/county_lon_lats.csv").drop(columns =
["Unnamed: 0"])

# Convert to int to avoid float imprecision
for k in ['latitude', 'longitude'] :
    location[k] = (10*location[k]).astype(int)

# Show
print(location.shape)
location.head(2)
%%time

historical_weather = pd.read_csv("/content/historical_weather.csv")

# Date processing + features extraction
for date_col in ['datetime'] :
    historical_weather = process_date(historical_weather, date_col)

# Reduce memory usage to avoid OOM (Out OF Memory error)
#historical_weather = reduce_memory_usage(historical_weather,
print_info=True)
```

```
# Show
print(historical_weather.shape)
historical_weather.head(2)
%%time

forecast_weather = pd.read_csv("/content/forecast_weather.csv")

# Date processing + features extraction
for date_col in ['forecast_datetime'] :
    forecast_weather = process_date(forecast_weather, date_col)

# Reduce memory usage to avoid OOM (Out OF Memory error)
#forecast_weather = reduce_memory_usage(forecast_weather, print_info=True)

# Show
print(forecast_weather.shape)
forecast_weather.head(2)

%%time

lgbm_params['n_estimators'] = best_iter

Feature_Imp = None

# Train and test
xtr, ytr = df[feats], df['target']
print(f"Train : {len(xtr)} rows.")

# Fit model
model = lgb.LGBMRegressor(**lgbm_params)
fit_params['eval_set'] = [(xtr, ytr)]
model.fit(xtr, ytr, **fit_params, callbacks = [lgb.log_evaluation(50)])

# Save model
joblib.dump(model, f"lgbm_model.pkl")

# Features Importance
Feature_Imp = pd.DataFrame(sorted(zip(model.feature_importances_, feats)),
columns=['Value','Feature'])
Feature_Imp['Value'] = 100* (Feature_Imp['Value'] /
Feature_Imp['Value'].max()) # Normalisation
Feature_Imp = Feature_Imp.sort_values(by='Value',
ascending=False).reset_index(drop=True)
# Features Importance
if len(Feature_Imp) > 90 : plt.figure(figsize=(7, 15))
elif len(Feature_Imp) > 60 : plt.figure(figsize=(7, 12))
elif len(Feature_Imp) > 30 : plt.figure(figsize=(7, 10))
else :
    plt.figure(figsize=(5, 5))
sns.barplot(x="Value", y="Feature", data=Feature_Imp.head(100))
plt.title('Features Importance')
plt.show()
# Most important features
Feature_Imp.Feature.tolist()[:20]
# Less important features
Feature_Imp.Feature.tolist()[::-1][:10]
```

```
import pandas as pd
from pathlib import Path
import json
from geopy.geocoders import Nominatim

pd.set_option('display.max_columns', 500)
files = list(Path("../input/predict-energy-behavior-of-
prosumers/").glob("*.csv"))
for file in files:
    print(f"creates: '{file.stem}'")
    globals()[file.stem] = pd.read_csv(file)
f = open('/content/county_id_to_name_map.json')
county_codes = json.load(f)
county_codes
name_mapping = {
    "valga": "valg",
    "põlva": "põlv",
    "jõgeva": "jõgev",
    "rapla": "rapl",
    "järva": "järv"
}

for i, coords in  forecast_weather[["latitude",
"longitude"]].drop_duplicates().iterrows():

    lat, lon = coords["latitude"], coords["longitude"]

    geoLoc = Nominatim(user_agent="GetLoc")

    # passing the coordinates
    locname = geoLoc.reverse(f"{lat}, {lon}")   # lat, lon
    if locname is None: continue

    location = locname.raw["address"]
    if location["country"] == "Eesti":
        county = location['county'].split()[0].lower()
        county = name_mapping.get(county, county)
        print(f"county: '{county}', county code:",
parsed_counties[county], (lat, lon))
TOP = 20
importance_data = pd.DataFrame({'name': clf.feature_names_in_,
'importance': clf.feature_importances_})
importance_data = importance_data.sort_values(by='importance',
ascending=False)

fig, ax = plt.subplots(figsize=(8,4))
sns.barplot(data=importance_data[:TOP],
            x = 'importance',
            y = 'name'
        )
patches = ax.patches
count = 0
for patch in patches:
    height = patch.get_height()
    width = patch.get_width()
```

```
    perc =
100*importance_data['importance'].iloc[count]#100*width/len(importance_dat
a)
    ax.text(width, patch.get_y() + height/2, f'{perc:.1f}%')
    count+=1

plt.title(f'The top {TOP} features sorted by importance')
plt.show()

!pip install catboost
import polars as pl

def timestamp_UTC_conversion(train, date_col) :
    """  """

    # To avoid bugs
    if date_col == 'date' :
        train = train.rename(columns = {'date' : 'date_'})
        date_col = 'date_'

    # Transform
    dt_transforms = [
                     #pl.col(date_col).str.to_datetime(time_zone='UTC'), #
UTC conversion to avoid different timezones
                      (pl.col(date_col).str.to_datetime().dt.date()).alias(
'date'), #  Extract date
                      pl.col(date_col).str.to_datetime().dt.time().alias('t
ime'), # Extract time
                    ]

    # Apply transform
    return  (pl.from_pandas(train)
               .with_columns((dt_transforms))
               .to_pandas()
             )
def process_date(train, date_col, prefixe = '', extract_features = False)
:

    # Force to str (to avoid datetime error)
    train[date_col] = train[date_col].astype(str)

    # Polar UTC conversion
    train = timestamp_UTC_conversion(train, date_col)

    # Convert to string
    train['date'] = train['date'].astype(str)
    train['time'] = train['time'].astype(str)
    train[date_col] = train['date'] + ' ' + train['time']

    # ----------------------------------------------------
    # FEATURES EXTRACTION

    # Return
    if not(extract_features) :
        return train

    # Date features
```

```
    train['year']  = (train['date'].apply(lambda x :
x[:4]).astype(int)).astype(int)
    train['month'] = train['date'].apply(lambda x : x[5:7]).astype(int)
    train['day']   = train['date'].apply(lambda x : x[8:10]).astype(int)

    # Day of week
    train['dayofweek'] = train[['year', 'month', 'day']].apply(lambda row
: datetime(row['year'], row['month'], row['day']).weekday(), axis=1)

    # Time features
    train['hour']    = train['time'].apply(lambda x : x[:2]).astype(int)
    train['minutes'] = train['time'].apply(lambda x : x[3:5]).astype(int)
    train['seconds'] = train['time'].apply(lambda x : x[6:7]).astype(int)

    # Rename
    if len(prefixe) > 0 :
        cols = ['year', 'month', 'day', 'hour', 'minutes', 'seconds']
        train = train.rename(columns = {k : f"{prefixe}_{k}" for k in
cols})

    # Drop column
    #train.drop(columns = [date_col], inplace=True)

    # Return
    return train
def reduce_memory_usage(df, print_info=False):
    """ iterate through all the columns of a dataframe and modify the data
type
        to reduce memory usage.
    """

    if print_info :
        print('*'*50)
        start_mem = df.memory_usage().sum() / 1024**2
        print('Memory before : {:.2f} MB'.format(start_mem))

    for col in df.columns:
        col_type = df[col].dtype

        if col_type != object:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max <
np.iinfo(np.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max <
np.iinfo(np.int16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max <
np.iinfo(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max <
np.iinfo(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
```

```
                if c_min > np.finfo(np.float16).min and c_max <
np.finfo(np.float16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max <
np.finfo(np.float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)
        else:
            df[col] = df[col].astype('category')

    if print_info :
        end_mem = df.memory_usage().sum() / 1024**2
        print('Memory after  : {:.2f} MB'.format(end_mem))
        print('Decreased by  : {:.1f}%'.format(100 * (start_mem - end_mem)
/ start_mem))
        print('*'*50 + '\n')

    return df
%%time

# Date processing + features extraction
for date_col in ['datetime'] :
    train = process_date(train, date_col, extract_features = True)

# Show
print(train.shape)
train.head(2)
N_TARGETS_REVEALED = 14
for day_shift in range(2, N_TARGETS_REVEALED+3) :

    # Add previous targets
    train['data_block_id_shifted'] = train['data_block_id'] + day_shift
    train = pd.merge(train,
                     train[["county", "is_business", "product_type",
"is_consumption", "time", "data_block_id_shifted",
"target"]].rename(columns = {"data_block_id_shifted":"data_block_id",


 "target" : f"target_revealed_{day_shift}days_ago",


}),
                     how = 'left',
                     on = ["county", "is_business", "product_type",
"is_consumption", "time", "data_block_id"],
                     )

    # Fill NaN with -1
    mask = train[f"target_revealed_{day_shift}days_ago"].isna()
    train.loc[mask, f"target_revealed_{day_shift}days_ago"] =
train.loc[mask, "target"]

# Drop useless column
train.drop(columns = ['data_block_id_shifted'], inplace=True)

# Show
```

```
print(train.shape)
train.tail(2)
county = 15
is_business = 1
product_type = 3
is_consumption = 1
df = train.copy()

# Show
print(df.shape)
df.head(2)
%%time

# Shift data_clock_id (during the submission phase, the data are available
2 days ago)
electricity_prices['data_block_id_shifted'] =
electricity_prices['data_block_id'] + 2

# Join
df = pd.merge(df,
              electricity_prices[["time", "data_block_id_shifted",
"euros_per_mwh"]].rename(columns =
{"data_block_id_shifted":"data_block_id"}),
              how = 'left',
              on = ["time", "data_block_id"],
             )

# Show
print(df.shape)
df.head(2)
%%time

# Shift data_clock_id (during the submission phase, the data are available
2 days ago)
gas_prices['data_block_id_shifted'] = gas_prices['data_block_id'] + 2

# Join
cols = ['data_block_id_shifted', 'time', 'lowest_price_per_mwh',
'highest_price_per_mwh']
df = df.merge(gas_prices[cols].rename(columns = {'data_block_id_shifted' :
'data_block_id'}),
              how='left',
              on=["time", "data_block_id"],
              )

# Show
print(df.shape)
df.head(2)
def process_weather_info(h, location=location) :

    # Drop duplicates
    h = h.drop_duplicates().reset_index(drop=True)

    # Convert to int to avoid float imprecision
    for k in ['latitude', 'longitude'] :
        h[k] = (10*h[k]).astype(int)
```

```
    # Add location
    h = pd.merge(h, location, how='left', on=['latitude', 'longitude'])

    # Fill NaN and force int
    h['county'] = h['county'].fillna(-1).astype(int)

    # Return
    return h
def process_weather_info(h, location=location):
    # Drop duplicates
    h = h.drop_duplicates().reset_index(drop=True)

    # Convert to int to avoid float imprecision, handling NaN values
    for k in ['latitude', 'longitude']:
        # Replace NaN with 0 before multiplying and converting to int
        h[k] = h[k].fillna(0)  # Fill NaN with 0
        h[k] = (10 * h[k]).astype(int)

    # Add location
    h = pd.merge(h, location, how='left', on=['latitude', 'longitude'])

    # Fill NaN and force int
    h['county'] = h['county'].fillna(-1).astype(int)

    # Return
    return h
%%time

# Shift datetime
historical_weather['datetime_shifted'] =
(pd.to_datetime(historical_weather['datetime'].astype(str)) +
pd.Timedelta(days=1, hours=13)).astype(str)

# Join
df = df.merge(historical_weather.drop(columns =
['datetime']).rename(columns = {'datetime_shifted' : 'datetime'}),
              how='left',
              on=['county', 'datetime'],
             )

# Show
print(df.shape)
df.head(2)
# Add location
forecast_weather = process_weather_info(forecast_weather)

# Show
print(forecast_weather.shape)
forecast_weather.head(2)
# Aggregate information over latitude/longitude
dict_agg = {'temperature' : ['min', 'mean', 'max', 'std'],
            'dewpoint' : ['min', 'mean', 'max', 'std'],
            'cloudcover_high' : ['min', 'mean', 'max', 'std'],
            'cloudcover_low' : ['min', 'mean', 'max', 'std'],
            'cloudcover_mid' : ['min', 'mean', 'max', 'std'],
            'cloudcover_total' : ['min', 'mean', 'max', 'std'],
            '10_metre_u_wind_component' : ['min', 'mean', 'max', 'std'],
```

```
            '10_metre_v_wind_component' : ['min', 'mean', 'max', 'std'],
            'direct_solar_radiation' : ['min', 'mean', 'max', 'std'],
            'surface_solar_radiation_downwards' : ['min', 'mean', 'max',
'std'],
            'snowfall' : ['min', 'mean', 'max', 'std'],
            'total_precipitation' : ['min', 'mean', 'max', 'std'],
           }

# Groupby
keys = ['county', 'forecast_datetime']
forecast_weather =
forecast_weather.groupby(keys).agg(dict_agg).reset_index()

# Flatten columns names
forecast_weather.columns = ['_'.join([xx for xx in x if len(xx)>0]) for x
in forecast_weather.columns]
forecast_weather.columns = [x + '_f' if x not in keys else x for x in
forecast_weather.columns]

# Show
print(forecast_weather.shape)
forecast_weather.head(2)
%%time

# Join
df = df.merge(forecast_weather.rename(columns = {'forecast_datetime' :
'datetime'}),
              how='left',
              on=['county', 'datetime'],
             )

# Fill NaN
df = df.sort_values(by=['datetime']).reset_index(drop=True)
for k in df :
    if k.endswith('_f') :
        df[k] = df[k].ffill().bfill()

# Show
print(df.shape)
df.head(2)
# Sort
df = df.sort_values(by=['datetime']).reset_index(drop=True)

# Fill NaN
df.fillna(0, inplace=True)

# Show
print(df.shape)
df.head(2)
del historical_weather, forecast_weather
_ = gc.collect()
def create_df(df, client, historical_weather, forecast_weather,
              electricity_prices, gas_prices, sample_prediction) :

    df = pd.merge(df,
                  electricity_prices[["time", "euros_per_mwh"]],
                  how = 'left',
```

```
                on = ["time"],
               )
    # Join
    cols = ['time', 'lowest_price_per_mwh', 'highest_price_per_mwh']
    df = df.merge(gas_prices[cols],
                how='left',
                on=["time"],
                )

    # Join
    df = df.merge(client.drop(columns = ['datetime', 'date', 'time']),
                how='left',
                on=['county', 'is_business', 'product_type'],
                )

    # Aggregate information over latitude/longitude
    dict_agg = {'temperature' : ['min', 'mean', 'max', 'std'],
                'dewpoint' : ['min', 'mean', 'max', 'std'],
                'rain' : ['min', 'mean', 'max', 'std'],
                'snowfall' : ['min', 'mean', 'max', 'std'],
                'surface_pressure' : ['min', 'mean', 'max', 'std'],
                'cloudcover_total' : ['min', 'mean', 'max', 'std'],
                'cloudcover_low' : ['min', 'mean', 'max', 'std'],
                'cloudcover_mid' : ['min', 'mean', 'max', 'std'],
                'cloudcover_high' : ['min', 'mean', 'max', 'std'],
                'windspeed_10m' : ['min', 'mean', 'max', 'std'],
                'winddirection_10m' : ['min', 'mean', 'max', 'std'],
                'shortwave_radiation' : ['min', 'mean', 'max', 'std'],
                'direct_solar_radiation' : ['min', 'mean', 'max', 'std'],
                'diffuse_radiation' : ['min', 'mean', 'max', 'std'],
               }

    # Groupby
    keys = ['county', 'datetime']
    historical_weather =
historical_weather.groupby(keys).agg(dict_agg).reset_index()

    # Flatten columns names
    historical_weather.columns = ['_'.join([xx for xx in x if len(xx)>0])
for x in historical_weather.columns]
    historical_weather.columns = [x + '_h' if x not in keys else x for x
in historical_weather.columns]

    # Shift datetime
    historical_weather['datetime'] =
(pd.to_datetime(historical_weather['datetime'].astype(str)) +
pd.Timedelta(days=1, hours=13)).astype(str)

    # Join
    df = df.merge(historical_weather,
                how='left',
                on=['county', 'datetime'],
               )

    # Aggregate information over datetime
    dict_agg = {'temperature' : ['min', 'mean', 'max', 'std'],
                'dewpoint' : ['min', 'mean', 'max', 'std'],
```

```
                'cloudcover_high' : ['min', 'mean', 'max', 'std'],
                'cloudcover_low' : ['min', 'mean', 'max', 'std'],
                'cloudcover_mid' : ['min', 'mean', 'max', 'std'],
                'cloudcover_total' : ['min', 'mean', 'max', 'std'],
                '10_metre_u_wind_component' : ['min', 'mean', 'max',
'std'],
                '10_metre_v_wind_component' : ['min', 'mean', 'max',
'std'],
                'direct_solar_radiation' : ['min', 'mean', 'max', 'std'],
                'surface_solar_radiation_downwards' : ['min', 'mean',
'max', 'std'],
                'snowfall' : ['min', 'mean', 'max', 'std'],
                'total_precipitation' : ['min', 'mean', 'max', 'std'],
               }

    # Groupby
    keys = ['county', 'forecast_datetime']
    forecast_weather =
forecast_weather.groupby(keys).agg(dict_agg).reset_index()

    # Flatten columns names
    forecast_weather.columns = ['_'.join([xx for xx in x if len(xx)>0])
for x in forecast_weather.columns]
    forecast_weather.columns = [x + '_f' if x not in keys else x for x in
forecast_weather.columns]

    # Join
    df = df.merge(forecast_weather.rename(columns = {'forecast_datetime' :
'datetime'}),
                  how='left',
                  on=['county', 'datetime'],
                 )

    # Fill NaN
    df = df.sort_values(by=['datetime']).reset_index(drop=True)
    for k in df :
        if k.endswith('_f') :
            df[k] = df[k].ffill().bfill()

    # Sort
    #df = df.sort_values(by=['datetime']).reset_index(drop=True)

    # Fill NaN
    df.fillna(0, inplace=True)

    # Return
    return df
forbidden_cols = ['target',
                  'datetime',
                  'date',
                  'row_id',
                  'data_block_id',
                  'prediction_unit_id',

                  # Useless feats
                  'minutes',
                  'snowfall_min_h',
```

```
                    'rain_min_h',
                    'seconds',
                    'highest_price_per_mwh',
                    'lowest_price_per_mwh',
                    'snowfall_max_h',
                    'snowfall_min_f',
                    'cloudcover_mid_min_f',
                    'cloudcover_high_min_f',
                   ]
numeric_cols   = df.select_dtypes(include=np.number).columns.tolist()

feats = [x for x in numeric_cols if x not in forbidden_cols]

print(f"{len(feats)} features.")
df[feats].head(1)
# Fewer feats
if False :
    # Fewer feats
    feats = ['day',
             'month',
             'euros_per_mwh',
             'hour',
             'dayofweek',
             'county',
             'target_revealed_6days_ago',
             'target_revealed_5days_ago',
             'target_revealed_4days_ago',
             'target_revealed_3days_ago',
             'target_revealed_2days_ago',
             'year',
             'is_consumption',
             'is_business',
             'product_type',
             'eic_count',
             'installed_capacity',
             ]

print(f"{len(feats)} features.")
df[feats].head(1)
lgbm_params = {'boosting_type': 'gbdt',
               'objective': 'regression',
               'metric' : 'mean_absolute_error',
               'importance_type': 'split',
               'learning_rate': 0.08,
               'n_estimators': 1000,
               'max_depth': -1,
               'min_child_samples': 120,
               'num_leaves': 250,
               'colsample_bytree': 0.85,
               'subsample': 0.85,
               'reg_alpha': 2,
               'reg_lambda': 1,
               'n_jobs': 2,
               'random_state': 12,
              }

fit_params = {"eval_metric" : 'mean_absolute_error',
```

```
              #"eval_set" : [(xtr, ytr), (xte, yte)],
              "eval_names": ['train', 'test'],
              "categorical_feature": 'auto'}
%%time

Iterations = []

for date_limit in ['2023-01-01 00:00:00',
                   '2023-02-01 00:00:00',
                   '2023-03-01 00:00:00',
                   '2023-04-01 00:00:00',
                   #'2023-05-01 00:00:00',
                  ] :

    # Train and test
    mask_train = (df['datetime'] <= date_limit)

    # Check if mask_train results in empty test set
    if not (~mask_train).any():  # Check if any values in ~mask_train are
True
        print(f"Skipping date_limit {date_limit} as it results in an empty
test set.")
        continue

    xtr, ytr = df.loc[mask_train, feats], df.loc[mask_train, 'target']
    xte, yte = df.loc[~mask_train, feats], df.loc[~mask_train, 'target']

    # Fit model
    model = lgb.LGBMRegressor(**lgbm_params)
    fit_params['eval_set'] = [(xtr, ytr), (xte, yte)]
    model.fit(xtr, ytr, **fit_params, callbacks = [lgb.early_stopping(15,
verbose=False),
                                                   lgb.log_evaluation(0)])

    # Add number of iterations
    Iterations.append(model.best_iteration_)

    # Predictions
    preds = np.clip(model.predict(xte), 0, 15000)

    # Print score
    print(f"{date_limit[:10]} :")
    print(f"Train/Test : {len(xtr)}/{len(xte)} rows.")
    print(f"Iterations : {Iterations[-1]}.")
    print(f"Score      : {round(mean_absolute_error(yte, preds), 2)}.")
    print()
# The number of iterations we will use for our training (trained on all
data)
print(Iterations)

# Check if Iterations is empty before calculating median
if Iterations:
    best_iter = int(np.median(Iterations))
else:
    best_iter = 1000  # Assign a default value if Iterations is empty

best_iter
```

```
# Plot RMSE
results = clf.evals_result()
train_mae, val_mae = results["validation_0"]["mae"],
results["validation_1"]["mae"]
x_values = range(0, len(train_mae))
fig, ax = plt.subplots(figsize=(8,4))
ax.plot(x_values, train_mae, label="Train MAE")
ax.plot(x_values, val_mae, label="Validation MAE")
ax.legend()
plt.ylabel("MAE Loss")
plt.title("XGBoost MAE Loss")
plt.show()
importance_data[importance_data['importance']<0.0005].name.values
def create_revealed_targets_test(data, previous_revealed_targets,
N_day_lags):
    '''🎯 Create new test data based on previous_revealed_targets and
N_day_lags 🎯 '''
    for count, revealed_targets in enumerate(previous_revealed_targets) :
        day_lag = count + 2

        # Get hour
        revealed_targets['hour'] =
pd.to_datetime(revealed_targets['datetime']).dt.hour

        # Select columns and rename target
        revealed_targets = revealed_targets[['hour', 'prediction_unit_id',
'is_consumption', 'target']]
        revealed_targets = revealed_targets.rename(columns = {"target" :
f"target_{day_lag}_days_ago"})

        # Add past revealed targets
        data = pd.merge(data,
                        revealed_targets,
                        how = 'left',
                        on = ['hour', 'prediction_unit_id',
'is_consumption'],
                       )

    # If revealed_target_columns not available, replace by nan
    all_revealed_columns = [f"target_{day_lag}_days_ago" for day_lag in
range(2, N_day_lags+1)]
    missing_columns = list(set(all_revealed_columns) - set(data.columns))
    data[missing_columns] = np.nan

    return data
from competition import make_env

__all__ = ['make_env']
!pip install enefit-competition-package
!pip install meteostat
import holidays # For working with holiday data
import numpy as np
import pandas as pd
import seaborn as sns
import xgboost as xgb
```

```
from datetime import datetime
from meteostat import Stations, Daily # For working with weather data
from xgboost import plot_importance
## Features Functions

def create_datetime_features(df):
    """
    Creates time series features from datetime index
    Adapted from: https://www.kaggle.com/code/robikscube/tutorial-time-
series-forecasting-with-xgboost#Look-at-Worst-and-Best-Predicted-Days
    """
    df['date'] = df.index
    df['hour'] = df['date'].dt.hour
    df['dayofweek'] = df['date'].dt.dayofweek
    df['quarter'] = df['date'].dt.quarter
    df['month'] = df['date'].dt.month
    df['year'] = df['date'].dt.year
    df['dayofyear'] = df['date'].dt.dayofyear
    df['dayofmonth'] = df['date'].dt.day
    df['weekofyear'] = df['date'].dt.isocalendar().week
    df['date'] = pd.to_datetime(df.index.date)
    return df

def get_holiday_features(df, country_code='US'):
    """
    Creates holiday features from datetime index
    """
    year_range = list(range(min(df.index.year), max(df.index.year) + 1))
    country_holidays = holidays.country_holidays(
        country_code,
        years=year_range,
        observed=False
    )
    holiday_df = pd.DataFrame(country_holidays.items())
    holiday_df.columns = ['date', 'holiday']
    holiday_df['date'] = pd.to_datetime(holiday_df['date'])
    return holiday_df

def get_weather_features(df, lat, lon):
    """
    Creates weather features based on latitude and longitude
    """
    room_temperature = 20  # Weather data is in Celsius
    start = min(df.index)
    end = max(df.index)

    stations = Stations()
    stations = stations.nearby(lat, lon)
    station = stations.fetch(1)

    # Get daily data
    weather_data = Daily(station['wmo'][0], start, end)
    weather_data = weather_data.fetch()
    weather_data = weather_data.dropna(axis=1)
    weather_data = weather_data.reset_index()
    weather_data['tmin_abs_diff_from_room_temperature'] =
abs(weather_data['tmin'] - room_temperature)
```

```
    weather_data['tmax_abs_diff_from_room_temperature'] =
abs(weather_data['tmax'] - room_temperature)
    weather_data = weather_data.rename(columns ={'time': 'date'})

    return weather_data
## Custom Evaluation Functions
def mean_absolute_percentage_error(y_true, y_pred):
    """
    Calculates MAPE given y_true and y_pred
    From: https://www.kaggle.com/code/robikscube/tutorial-time-series-
forecasting-with-xgboost#Look-at-Worst-and-Best-Predicted-Days
    """
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
df = pd.read_csv('/content/AEP_hourly.csv', index_col=[0],
parse_dates=[0])
data = create_datetime_features(df)
holiday_features = get_holiday_features(data)
weather_features = get_weather_features(data, 41.9833, -87.9167) # Chicago
O'Hare Airport, IL
data['datetime'] = data.index
data = data.merge(holiday_features, how='left', on='date')
data['holiday'] = data['holiday'].fillna('')
data = pd.get_dummies(data)
data = data.merge(weather_features, how='left', on='date')
# Check the existing column names in your DataFrame
print(data.columns)

# If 'COMED_MW' is not present, identify the correct column name
# For example, if the column is named 'AEP_MW' in your data:
correlation_matrix = data[['AEP_MW', 'tmin', 'tmax', 'prcp', 'snow',
'wspd', 'tmin_abs_diff_from_room_temperature',
'tmax_abs_diff_from_room_temperature']].corr()

# Alternatively, if the column name is slightly different (e.g., due to
typos):
# 1. Correct the column name in the code
# 2. Or rename the column in the DataFrame if needed:
#   data = data.rename(columns={'old_column_name': 'COMED_MW'})
#   Replace 'old_column_name' with the actual incorrect name

# If you are sure that the column should exist, double-check the data
loading and processing steps.
# There might be an error in how you are creating or manipulating the
DataFrame.
sns.heatmap(correlation_matrix)
# Check the existing column names in your DataFrame
print(train.columns)

# If 'COMED_MW' is not present, identify the correct column name
# For example, if the column is named 'AEP_MW' in your data:
train_drop_cols = ['date', 'datetime', 'AEP_MW']  # Replace 'AEP_MW' with
the actual column name if different

# Alternatively, if the column name is slightly different (e.g., due to
typos):
# 1. Correct the column name in the code
```

```
# 2. Or rename the column in the DataFrame if needed:
#   train = train.rename(columns={'old_column_name': 'COMED_MW'})
#   Replace 'old_column_name' with the actual incorrect name

X_train = train.drop(columns=train_drop_cols, errors='ignore') # Add
errors='ignore' to avoid KeyError if a column is not found
y_train = train['AEP_MW']  # Replace 'AEP_MW' with the actual column name
if different
X_test = test.drop(columns=train_drop_cols, errors='ignore') # Add
errors='ignore' to avoid KeyError if a column is not found
y_test = test['AEP_MW']  # Replace 'AEP_MW' with the actual column name if
different
reg = xgb.XGBRegressor(
    n_estimators=1000,
    early_stopping_rounds=50
)
reg.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
       verbose=False)
_ = plot_importance(reg, height=0.9)
_ = plot_importance(reg, height=0.9)
mean_absolute_percentage_error(y_true=y_test, y_pred=predictions)
train = data.loc[data['datetime'] <= split_date].copy()
test = data.loc[data['datetime'] > split_date].copy()
train_cols = ['hour', 'dayofweek', 'quarter', 'month', 'year',
       'dayofyear', 'dayofmonth']
X_train = train[train_cols]
# Replace 'target_column_name' with the actual name of your target column
target_column_name = 'AEP_MW'  # Example: Assuming 'AEP_MW' is your target
column
y_train = train[target_column_name]
X_test = test[train_cols]
y_test = test[target_column_name]

reg = xgb.XGBRegressor(
    n_estimators=1000,
    early_stopping_rounds=50
)
reg.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
       verbose=False)
predictions = reg.predict(X_test)
# Assuming mean_absolute_percentage_error is defined elsewhere in your
code
mean_absolute_percentage_error(y_true=y_test, y_pred=predictions)
```