Final Project for Machine Learning Class

Instructor: Elias Houstis

ECE 461: Machine Learning for Data Science and Analysis

Authors:
Organtzoglou Evangelos
Charalambos Malakoudis
Christodoulos Zerdalis

Date: 16/1/2025

Contents

1 Abstract

In this final project, we explore advanced machine learning techniques to predict energy use in appliances using a dataset of sensor readings and environmental variables. Inspired by the study "Data Driven Prediction Models of Energy Use of Appliances in a Low-Energy House," this project builds on the methods and findings of the paper. The dataset includes indoor environmental mea such as temperature and humidity from wireless sensors, external weather data from a nearby airport station, and appliance energy consumption metrics.

This project emphasizes data preprocessing and feature selection to optimize model performance and identify the most critical predictors of energy consumption. By analyzing and ranking feature importance, key insights are gained into the influence of factors such as atmospheric pressure, indoor environmental conditions, and time-of-day effects. We apply various machine learning models to establish predictive accuracy, with an emphasis on cross-validation to ensure reliability.

Through comprehensive experimentation, this study aims to evaluate the ability of these predictive models to provide actionable insights into energy use patterns in residential settings. The outcomes have implications for optimizing appliance usage, improving energy efficiency, and informing building design strategies. Ultimately, the findings aim to bridge the gap between theoretical modeling and practical applications in energy management and sustainability.

2 Introduction

2.1 Problem Statement

In the context of growing global energy demand, predicting residential appliance energy use is a key challenge. Appliances represent a significant portion of household energy consumption, impacting costs and contributing to environmental issues like carbon emissions.

Time series forecasting is particularly suited to this problem as energy usage patterns depend on temporal factors such as daily routines and weather conditions. By leveraging historical data, we can uncover these trends and make accurate predictions to improve energy efficiency.

2.2 Objectives

The primary goal of this project is to forecast the energy consumption of household appliances based on sensor and weather data. By leveraging features such as indoor temperature, humidity, and external weather conditions, the objective is to develop accurate models that can predict energy usage trends over time. These predictions are intended to provide insights for optimizing energy management in residential settings.

A key focus of this project is the implementation and evaluation of advanced machine learning techniques, including deep learning models and transformers. Transformers, known for their exceptional performance in sequence-to-sequence tasks, are explored in this context for their potential to capture temporal dependencies and complex relationships within time-series data. This approach aims to push the boundaries of traditional forecasting methods and assess the suitability of state-of-the-art techniques for energy prediction tasks.

2.3 Scope and Limitations

This study is limited to predicting aggregate energy consumption in a single residential household using a finite dataset. The analysis relies on sensor and weather data collected over a few months, which may not capture long-term or seasonal variations.

The computational demands of training deep learning models, particularly transformers, are a key constraint. Despite using a normal student laptop, resource limitations may restrict extensive experimentation and fine-tuning. Additionally, the weather data from a distant station might introduce minor inaccuracies, potentially impacting prediction accuracy.

3 Literature Review

Next , we reviewed articles related to these models:

- **Classical time series models (ARIMA, ETS)**
- **Machine learning models (XGBoost, Random Forest)**
- **Deep learning models (LSTMs, GRU, CNNs)**
- **Transformer-based approaches (e.g., Informer, Time Series Transformer)**

The links to the work we reviewed are cited at the end of this report. After the professor's latest ananouncment on the Piazza platform and our research with these articles , we concluded that the models we had to use are the ones shown below.

4 Data Collection and Preprocessing

First we create a new `Jupyter Notebook` on whuch we are going to implement our preprocessing of the dataset.

After loading the file using the `Pandas` library in `Python`using this basic code:

```
    import pandas as pd
energy_data = pd.read_csv('C:\our_file_path_to_the_dataset\energydata_complet
print(energy_data.head())
```

As we can see, our chosen dataset consists of 29 columns. Let's see what each column represents:

1. **Date**: Timestamp of each measurement
2. **Appliances**: Energy use of appliances (in Wh)
3. **lights**: Energy use of lights (in Wh)
4. **T1 ... T9**: Temperature readings in different areas

5. **RH_1 ... RH_9**: Relative humidity in different areas
6. **T_out**: Outdoor temperature
7. **Press_mm_hg**: Pressure in mmHg
8. **RH_out**: Outdoor relative humidity
9. **Windspeed**: Outdoor windspeed
10. **Visibility**: Outdoor visibility
11. **Tdewpoint**: Dew point temperature
12. **rv1 and rv2**: Random variables for anonymization

Dataset Description

The data represents energy consumption and environmental variables collected from a low-energy house in Stambruges, Belgium. This house adheres to passive house standards, incorporating energy-efficient features like triple-glazed windows, high-efficiency ventilation, and low air leakage. **Key details:**

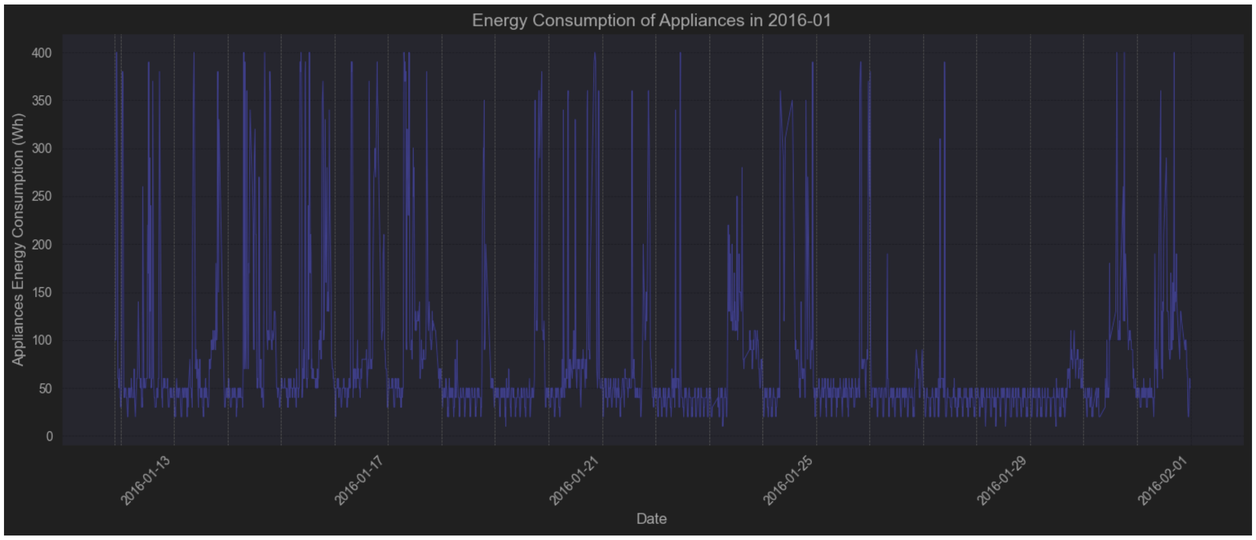
1. **Source**: Data was collected using M-BUS energy counters for appliances and lights, a ZigBee wireless sensor network for temperature and humidity, and weather data from a nearby airport station (Chièvres Airport).
2. **Variables**:
   - **Energy**: Consumption by appliances (primary focus) and lights.
   - **Indoor Environment**: Temperature and humidity from multiple rooms (e.g., kitchen, living room, laundry).
   - **Outdoor Environment**: Temperature, humidity, wind speed, pressure, visibility, and dew point from the weather station.
3. **Time Granularity**: 10-minute intervals.
4. **Duration**: Approximately 4.5 months, covering 137 days.

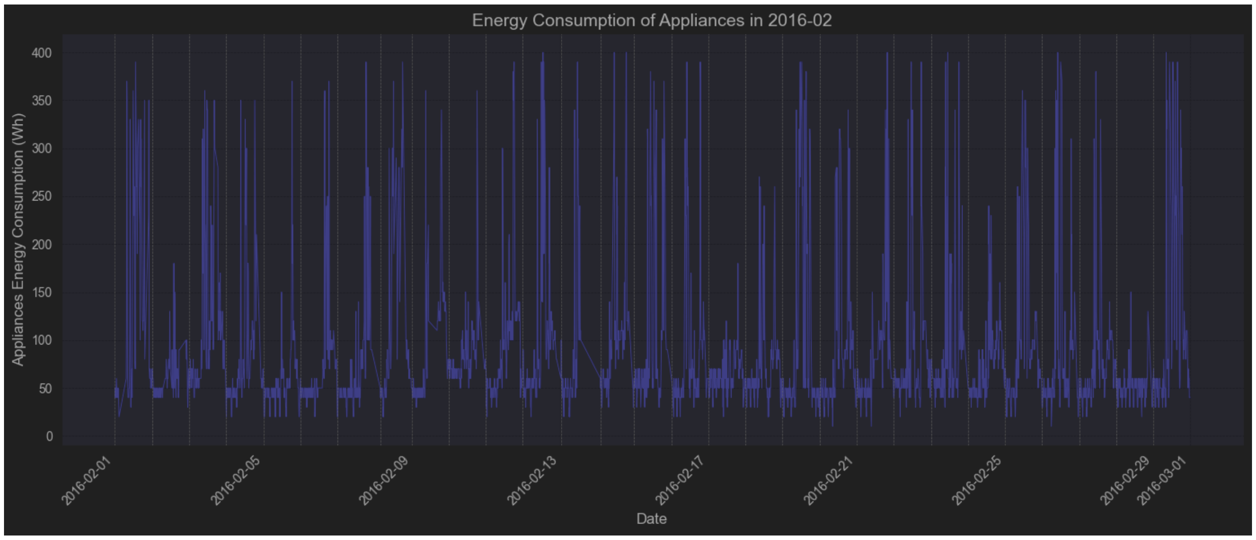
4.1 We are now moving to the Cleaning process:

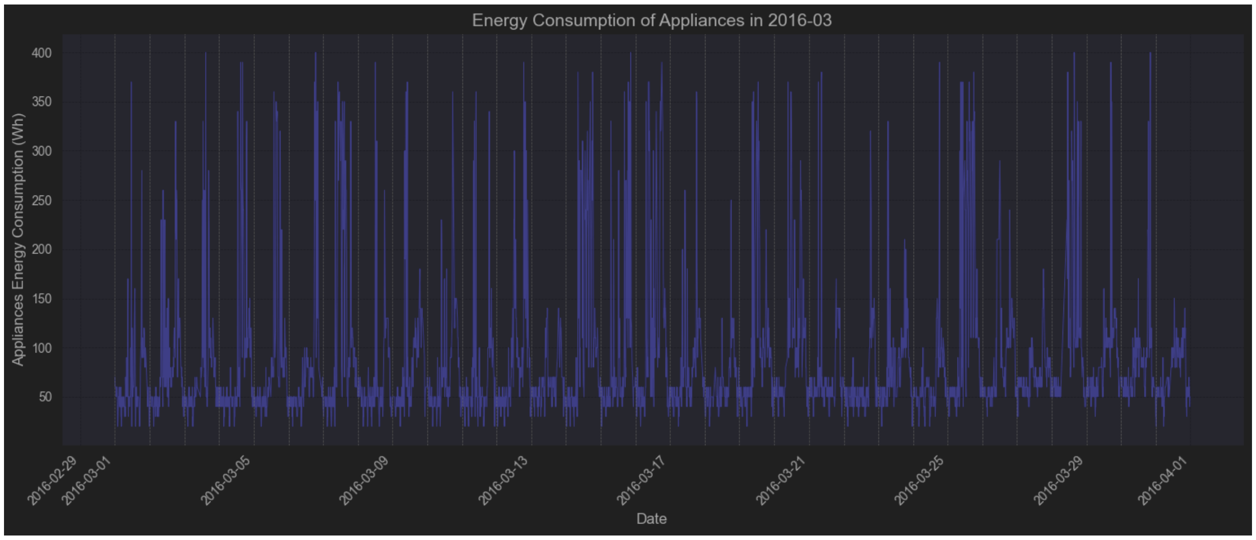
```
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import adfuller
from pandas.plotting import lag_plot
import numpy as np
```

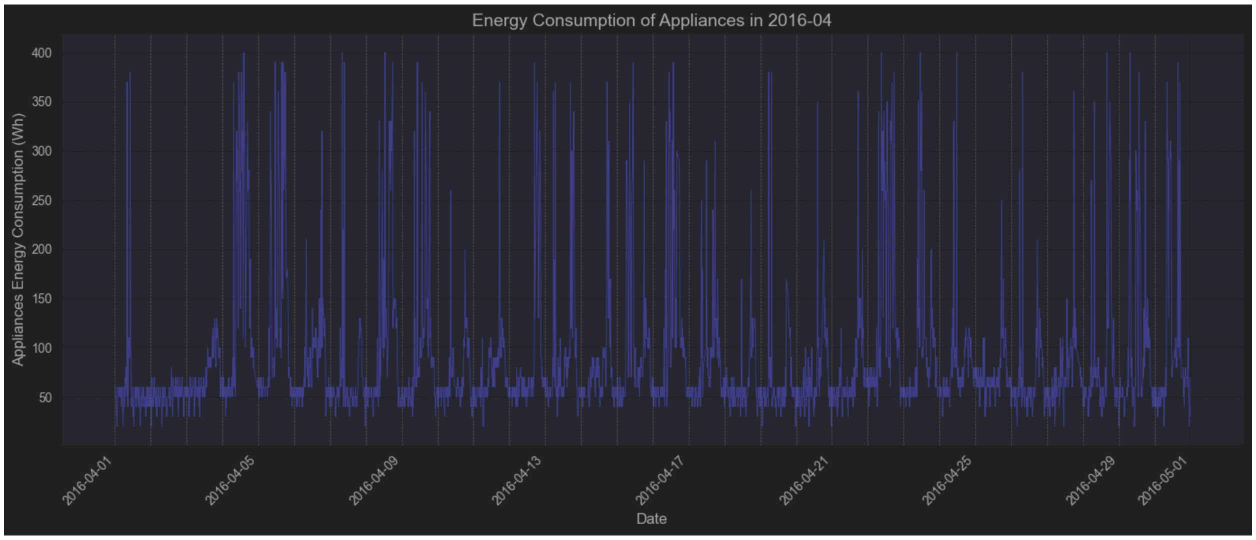
```
from sklearn.preprocessing import StandardScaler

# Convert 'date' to datetime and set it as the index
energy_data['date'] = pd.to_datetime(energy_data['date'])
energy_data.set_index('date', inplace=True)

# Check for missing values and forward fill since we are dealing
   with time series
# data and 10 minute intervals the most logical halding method is
   using the last
# known value
if energy_data.isnull().sum().sum() > 0:
    energy_data.fillna(method='ffill', inplace=True)

# Handle outliers using z-score with a threshold of 3 standard
   deviations which
# is based on the empirical rule. Rows that exceed the threshold are
    removed
z_scores = np.abs((energy_data - energy_data.mean()) / energy_data.
   std())
energy_data = energy_data[(z_scores < 3).all(axis=1)]

# Scaling the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(energy_data)
scaled_data = pd.DataFrame(scaled_data, columns=energy_data.columns,
    index=energy_data.index)

# Trend Visualization Split by Month with Daily Lines
print("Trend Visualization:")
months = energy_data.index.to_period("M").unique()  # Extract unique
    months from the index

for month in months:
    plt.figure(figsize=(14, 6))
    month_data = energy_data[month.start_time:month.end_time]['
       Appliances']
    month_data.plot(color='blue', linewidth=0.7, alpha=0.8)
    for day in month_data.index.day.unique():
        plt.axvline(x=month_data.index[month_data.index.day == day
           ][0], color='gray', linestyle='--', linewidth=0.5, alpha
           =0.7)
    plt.title(f'Energy Consumption of Appliances in {month}',
       fontsize=14)
    plt.xlabel('Date', fontsize=12)
    plt.ylabel('Appliances Energy Consumption (Wh)', fontsize=12)
    plt.grid(visible=True, linestyle='--', linewidth=0.5)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

# Additional Analysis: Weekday vs. Weekend
# Add a 'day_type' column to classify weekdays and weekends
energy_data['day_of_week'] = energy_data.index.dayofweek
energy_data['day_type'] = energy_data['day_of_week'].apply(lambda x:
    'Weekend' if x >= 5 else 'Weekday')

# Group by day type and calculate average consumption
avg_consumption = energy_data.groupby('day_type')['Appliances'].mean
   ()

# Plot weekday vs. weekend average consumption
print("Weekday vs Weekend average:")
plt.figure(figsize=(8, 6))
avg_consumption.plot(kind='bar', color=['orange', 'blue'], alpha
   =0.8)
plt.title('Average Energy Consumption: Weekday vs Weekend', fontsize
   =14)
plt.ylabel('Average Appliances Energy Consumption (Wh)', fontsize
   =12)
plt.xlabel('Day Type', fontsize=12)
plt.xticks(rotation=0)
plt.grid(visible=True, linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()


# Seasonal Decomposition without Residuals
print("Seasonal:")
seasonal_decompose_result = seasonal_decompose(energy_data['
   Appliances'], model='additive', period=144)
plt.figure(figsize=(14, 10))
plt.subplot(3, 1, 1)
plt.plot(seasonal_decompose_result.trend, label='Trend', color='blue
   ', alpha=0.8)
plt.title('Trend Component')
plt.grid(visible=True, linestyle='--', linewidth=0.5)
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(seasonal_decompose_result.seasonal, label='Seasonality',
   color='orange', alpha=0.8)
plt.title('Seasonal Component')
plt.grid(visible=True, linestyle='--', linewidth=0.5)
plt.legend()

plt.tight_layout()
plt.show()

# Autocorrelation Plot
print("Autocorrelation plot:")
plt.figure(figsize=(12, 6))
plot_acf(energy_data['Appliances'], lags=50, title="Autocorrelation
    of Appliances Energy Consumption")
plt.grid(visible=True, linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()

# Stationarity Check (ADF Test)
print("Stationarity check:")
adf_test = adfuller(energy_data['Appliances'])
print("ADF Test Results:")
print(f"ADF Statistic: {adf_test[0]}")
print(f"p-value: {adf_test[1]}")
print("Critical Values:", adf_test[4])
```

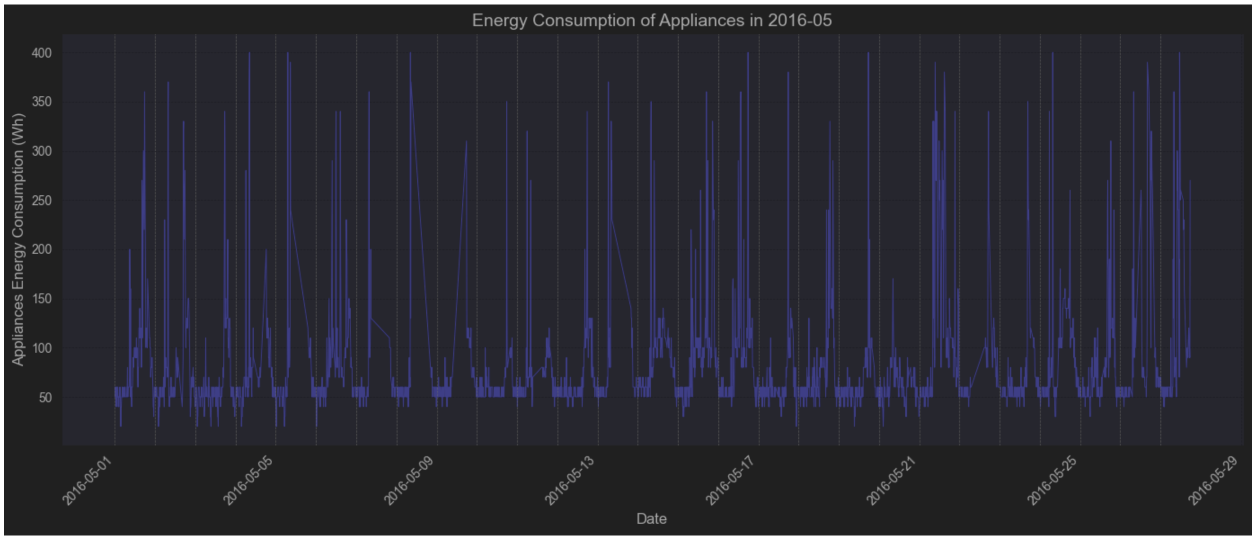
Following are the produced plots:

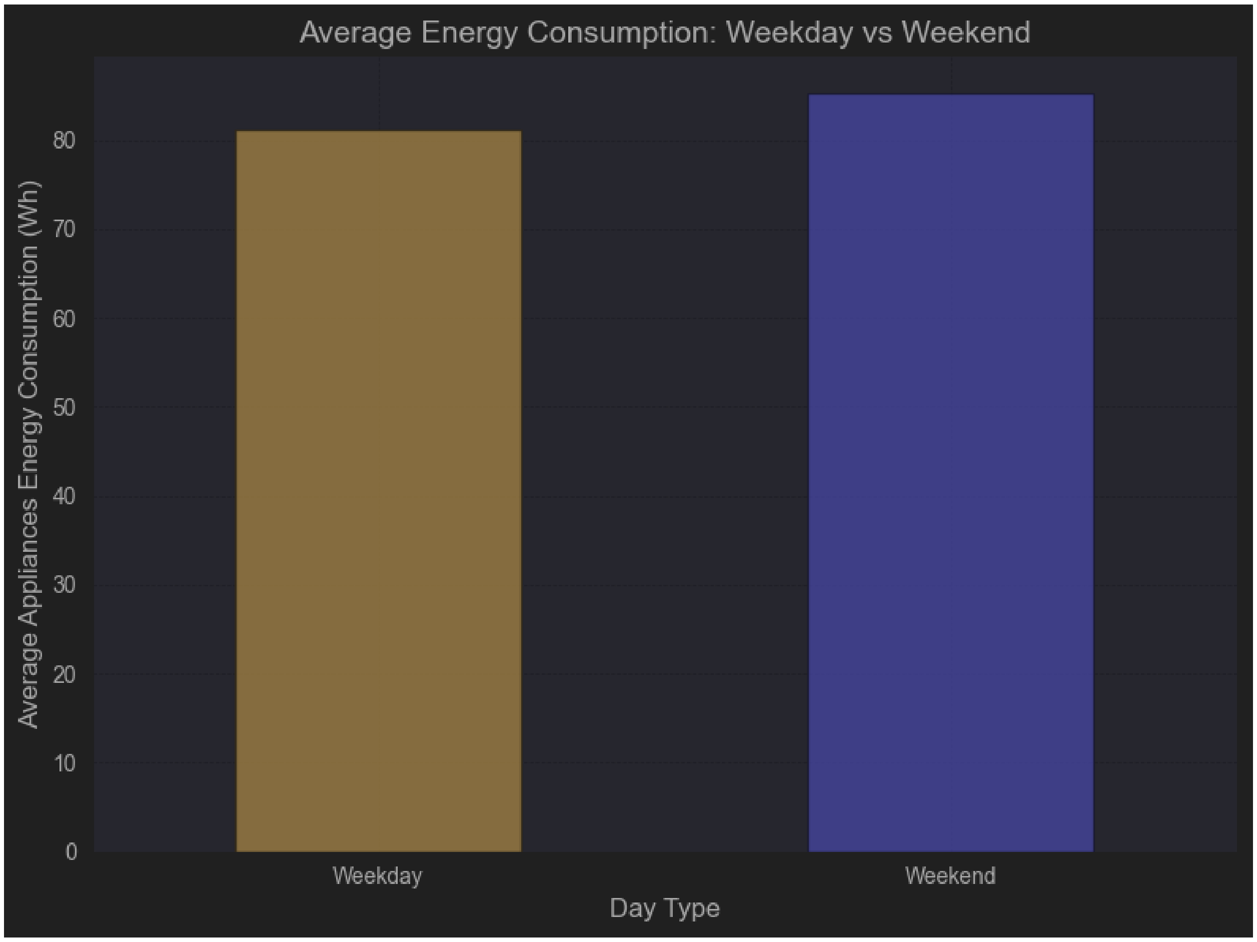


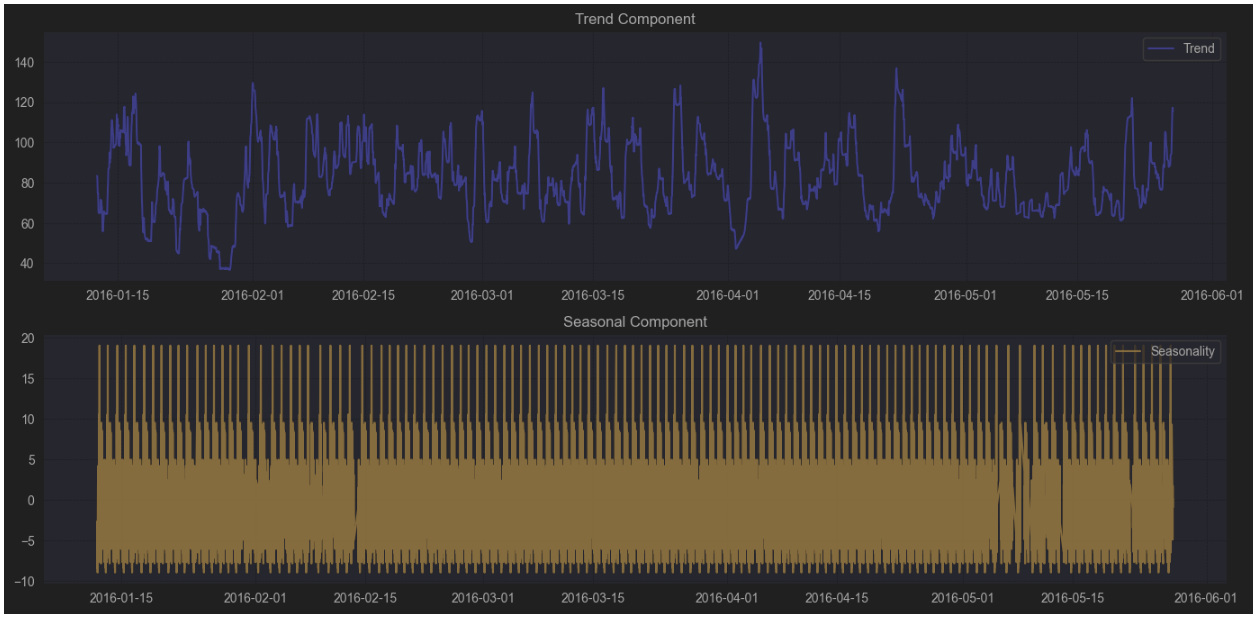

Energy Consumption of Appliances in 2016-02
Appliances Energy Consumption (Wh)
Date

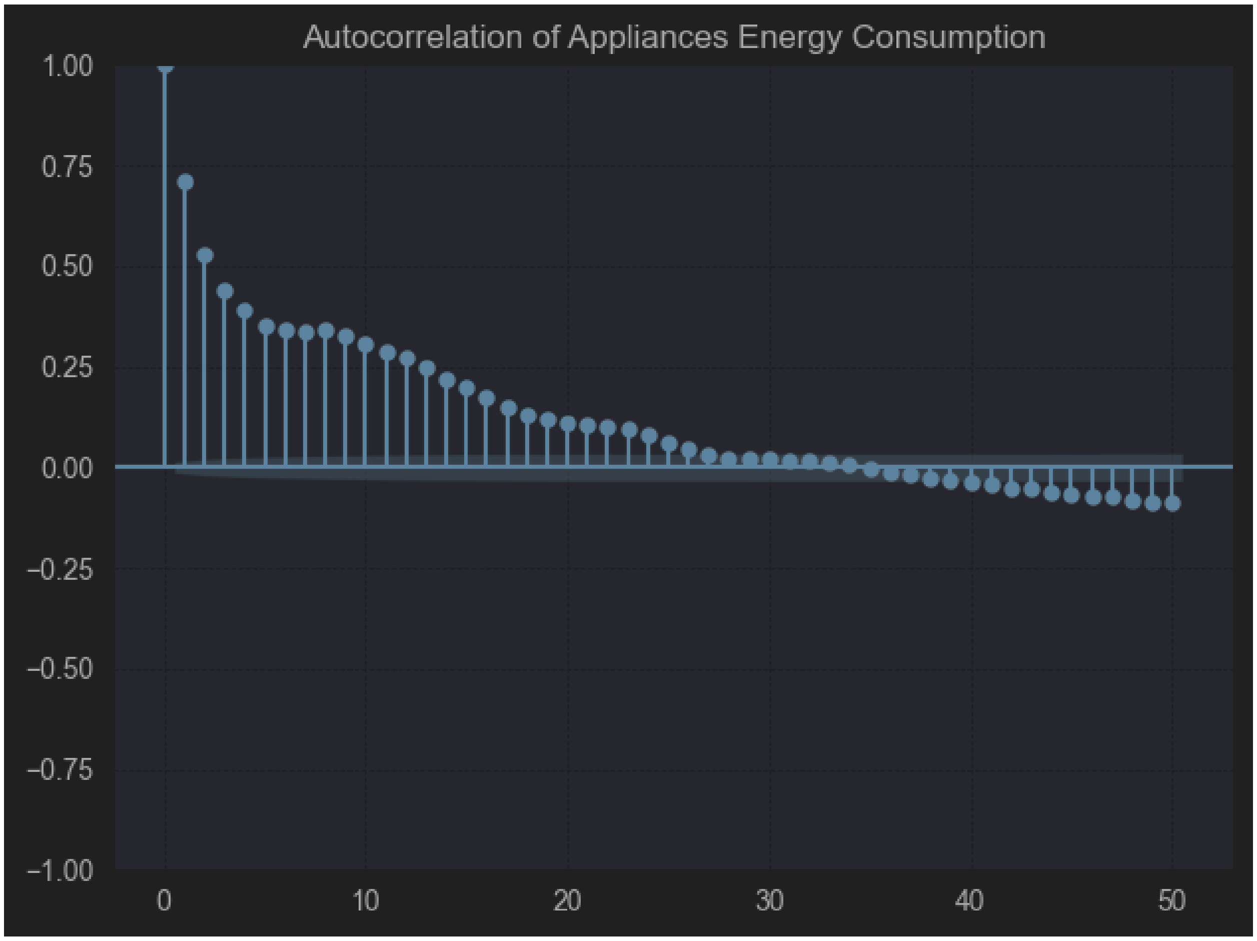

Energy Consumption of Appliances in 2016-03
Appliances Energy Consumption (Wh)
Date


Energy Consumption of Appliances in 2016-04
Appliances Energy Consumption (Wh)
Date


Energy Consumption of Appliances in 2016-05
Appliances Energy Consumption (Wh)
400
350
300
250
200
150
100
50
2016-05-01
2016-05-05
2016-05-09
2016-05-13
2016-05-17
2016-05-21
2016-05-25
2016-05-29
Date


Average Energy Consumption: Weekday vs Weekend
Average Appliances Energy Consumption (Wh)
0
10
20
30
40
50
60
70
80
Weekday
Weekend
Day Type


Trend Component
Trend
140
120
100
80
60
40
2016-01-15
2016-02-01
2016-02-15
2016-03-01
2016-03-15
2016-04-01
2016-04-15
2016-05-01
2016-05-15
2016-06-01
Seasonal Component
Seasonality
20
15
10
5
0
−5
−10


Autocorrelation of Appliances Energy Consumption
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50

Stationarity Check

ADF Test Results:

- ADF Statistic: −21.609593410592964
- p-value: 0.0
- Critical Values: {'1%': np.float64(−3.4307274611324305), '5%': np.float64(−2.861706823136832), '10%': np.float64(−2.5668587956436157)}

  Across all months, the appliance energy consumption shows a repeating daily cycle with noticeable spikes and dips. These patterns are likely linked to human activities such as morning, evening, and night routines. We could confidently assume that the spikes in energy usage we can observe are due to daytime activities since they are also prominent in the middle of the days observed.

Month-to-Month Variations:

- **January 2016**: Shows consistent daily patterns with moderate peaks.
- **February 2016**: More prominent spikes, possibly due to colder weather leading to higher appliance usage for heating.
- **March to May 2016**: Peaks remain consistent, but daily usage appears slightly higher in March and April compared to January. This may reflect seasonal or behavioral changes.

The dataset exhibits strong short-term dependencies, making it well-suited for time-series models like ARIMA or LSTM that rely on lagged values.
Next is the Feature Engineering:

4.2 Feature Engineering

```
# Feature Engineering
# 1. Lag Features
for lag in range(1, 4):
    energy_data[f'Appliances_lag_{lag}'] = energy_data['Appliances'
       ].shift(lag)

# 2. Rolling Window Statistics
energy_data['Appliances_rolling_mean'] = energy_data['Appliances'].
   rolling(window=3).mean()
energy_data['Appliances_rolling_std'] = energy_data['Appliances'].
   rolling(window=3).std()

# 3. Time-Based Features
energy_data['hour'] = energy_data.index.hour
```

```
energy_data['day'] = energy_data.index.day
energy_data['weekday'] = energy_data.index.weekday
energy_data['month'] = energy_data.index.month

# 4. Cyclical Features for Time
energy_data['hour_sin'] = np.sin(2 * np.pi * energy_data['hour'] /
   24)
energy_data['hour_cos'] = np.cos(2 * np.pi * energy_data['hour'] /
   24)
energy_data['weekday_sin'] = np.sin(2 * np.pi * energy_data['weekday
   '] / 7)
energy_data['weekday_cos'] = np.cos(2 * np.pi * energy_data['weekday
   '] / 7)

# 5. Interaction Terms
energy_data['lag1_lag2_interaction'] = energy_data['Appliances_lag_1
   '] * energy_data['Appliances_lag_2']

# 6. Aggregated Statistics for Each Day
daily_stats = energy_data.resample('D').agg({
    'Appliances': ['mean', 'sum', 'min', 'max']
})
daily_stats.columns = ['daily_mean', 'daily_sum', 'daily_min', '
   daily_max']
energy_data = energy_data.merge(daily_stats, left_index=True,
   right_index=True, how='left')

# Drop rows with NaN values introduced by lag and rolling
   calculations
energy_data.dropna(inplace=True)

# Save the cleaned and processed data
cleaned_data_path = 'cleaned_energy_data.csv'
energy_data.to_csv(cleaned_data_path)

print(f"Cleaned data saved to {cleaned_data_path}")

# Feature Engineering
# 1. Lag Features
for lag in range(1, 4):
    energy_data[f'Appliances_lag_{lag}'] = energy_data['Appliances'
       ].shift(lag)

# 2. Rolling Window Statistics
energy_data['Appliances_rolling_mean'] = energy_data['Appliances'].
   rolling(window=3).mean()
```

```
energy_data['Appliances_rolling_std'] = energy_data['Appliances'].
    rolling(window=3).std()

# 3. Time-Based Features
energy_data['hour'] = energy_data.index.hour
energy_data['day'] = energy_data.index.day
energy_data['weekday'] = energy_data.index.weekday
energy_data['month'] = energy_data.index.month

# 4. Cyclical Features for Time
energy_data['hour_sin'] = np.sin(2 * np.pi * energy_data['hour'] /
    24)
energy_data['hour_cos'] = np.cos(2 * np.pi * energy_data['hour'] /
    24)
energy_data['weekday_sin'] = np.sin(2 * np.pi * energy_data['weekday
    '] / 7)
energy_data['weekday_cos'] = np.cos(2 * np.pi * energy_data['weekday
    '] / 7)

# 5. Interaction Terms
energy_data['lag1_lag2_interaction'] = energy_data['Appliances_lag_1
    '] * energy_data['Appliances_lag_2']

# 6. Aggregated Statistics for Each Day
daily_stats = energy_data.resample('D').agg({
    'Appliances': ['mean', 'sum', 'min', 'max']
})
daily_stats.columns = ['daily_mean', 'daily_sum', 'daily_min', '
    daily_max']
energy_data = energy_data.merge(daily_stats, left_index=True,
    right_index=True, how='left')

# Drop rows with NaN values introduced by lag and rolling
    calculations
energy_data.dropna(inplace=True)

# Save the cleaned and processed data
cleaned_data_path = 'cleaned_energy_data.csv'
energy_data.to_csv(cleaned_data_path)

print(f"Cleaned data saved to {cleaned_data_path}")
```

After the Feature engineering demonstrated above , we export the cleaned file.

4.3 Conclusion of Dataset Cleaning

After observing the dataset , looking at each column separately and trying to understand its purpose we performed some Cleaning and Feature Engineering code that we felt was necessary, We are now ready to start implementing the models in order to predict the `Energy Consumption`.

5 Implementation of GBM

We began our analysis by implementing a Gradient Boosting Machine (GBM) as the initial model to predict appliance energy consumption. GBM was chosen for its ability to handle tabular data efficiently and its proven performance in regression tasks. This provided a baseline for comparison with more advanced models like GRU, Transformers, and CNNs that were to be implemented afterwards.So we started with very basic code that implemented GBM to the dataset:

```
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split

file_path = 'C:\our_file_path_to_the_dataset\cleaned_energy_data.csv
   '
df = pd.read_csv(file_path)

df['date'] = pd.to_datetime(df['date'])

df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute

df = df.drop(columns=['date'])

X = df.drop(columns=["Appliances"])
y = df["Appliances"].astype(float)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.3, random_state=42)

GBM_regressor = GradientBoostingRegressor()
GBM_regressor.fit(X_train, y_train)
predictions = GBM_regressor.predict(X_test)

print(predictions)
```

Although after calculating the Cross-validated RMSE with:

```
scores = cross_val_score(GBM_regressor, X, y, cv=5, scoring='
   neg_mean_squared_error')
print("Cross-validated RMSE:", (-scores.mean())**0.5)
```

We got Cross-validated RMSE: 62.25273931500234 , which seemed high so we decised to compare the RMSE to a simple baseline model, such as predicting the mean of Appliances for all instances.We calculated the Baseline RMSE with the following code:

```
from sklearn.metrics import mean_squared_error
import numpy as np

baseline_rmse = mean_squared_error(y, [np.mean(y)] * len(y), squared
   =False)
print("Baseline RMSE:", baseline_rmse)
```

And we got: Baseline RMSE: 20.748890597642127
So we knew that we had to apply some changes to the code to get improved results. We ended up with this code:

```
import pandas as pd
import sklearn.ensemble
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split,
   cross_val_score
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error,
    r2_score
import numpy as np
from sklearn.preprocessing import LabelEncoder

file_path = 'C:\our_file_path_to_the_dataset\cleaned_energy_data.csv
   '
df = pd.read_csv(file_path)

df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute
df = df.drop(columns=['date'])

label_encoder = LabelEncoder()
df['day_type'] = label_encoder.fit_transform(df['day_type'])
```

```
correlation_matrix = df.corr()

plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="
   coolwarm", cbar=True)
plt.title("Correlation Matrix")
plt.show()

target_correlation = correlation_matrix["Appliances"].sort_values(
   ascending=False)
print("Correlation with the target (Appliances):\n",
   target_correlation)

selected_features = target_correlation[target_correlation.abs() >
   0.2].index.tolist()
selected_features.remove("Appliances")

X = df[selected_features]
y = df["Appliances"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.3, random_state=42)

GBM_regressor = sklearn.ensemble.GradientBoostingRegressor()
GBM_regressor.fit(X_train, y_train)
y_pred = GBM_regressor.predict(X_test)
```

Essentially we performed a correlation analysis which helps identify relationships between features and the target variable, as well as detect highly correlated features that might lead to redundancy.

This time we got , Cross-validated RMSE: 16.19287462128741 and we knew we were on the right track since the Cross-validated RMSE was lower than the Baseline.Also with a basic computation of multiple evaluation metrics using this code:

```
rmse = mean_squared_error(y_test, y_pred, squared=False)
mae = mean_absolute_error(y_test, y_pred)
mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100
r2 = r2_score(y_test, y_pred)

print("RMSE:", rmse)
print("MAE:", mae)
print("MAPE (%):", mape)
print("\[ R^2 \]:", r2)
```

we got these results:

- RMSE: 8.100254626478874

- MAE: 4.321641894633511
- MAPE (%): 8.215886885293386
- R^2 : 0.4474576409104656

The results were looking very good so we went straight to the next model.

6 Implementation of XGBoost

Implementing XGBoost wasn't this tricky this time , having seen how to implement GBM. The only part where we had to be careful was to understand the situation and what problem we are working on. In this part , we had to solve a regression problem which was important to recognize because we then knew that we had to use the XGBRegressor library and not the XGBClassifier. In the problem of predicting energy consumption (Appliances), the target variable consists of continuous numerical values. Therefore, the task is a regression problem, not a classification problem. We used XGBRegressor, which is specifically designed for regression tasks, to train the model. In contrast, XGBClassifier would have attempted to convert the continuous values into discrete categories, which is incompatible with the nature of the problem. Using XGBRegressor allowed us to accurately predict the target values and evaluate the model using RMSE, a standard metric for regression problems.The code we used was:

```
import pandas as pd
import numpy as np
import sklearn.ensemble
from sklearn.model_selection import train_test_split
import xgboost as xgb
from xgboost import XGBRegressor
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error, r2_score

file_path = 'C:\our_file_path_to_the_dataset\cleaned_energy_data.csv
    '
df = pd.read_csv(file_path)

df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute
df = df.drop(columns=['date'])

X = df.drop(columns=["Appliances"])
```

```
y = df["Appliances"].astype(float)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.3, random_state=42)

X_train['day_type'] = X_train['day_type'].astype('category')
X_test['day_type'] = X_test['day_type'].astype('category')

xgb = XGBRegressor()
xgb.fit(X_train, y_train)
y_pred = xgb.predict(X_test)

print(y_pred)
```

After that , what was left was to evaluate the model using the previous metrics:

```
rmse = mean_squared_error(y_test, y_pred, squared=False)
print("RMSE:", rmse)
mae = mean_absolute_error(y_test, y_pred)
print("MAE:", mae)
mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100
print("MAPE:", mape)
r2 = r2_score(y_test, y_pred)
print("R^2:", r2)
```

The results we got these results:

- RMSE: 10.6391314815527
- MAE: 7.169465236174754
- MAPE (%): 14.858014428135244
- R^2 :0.059214800464668094

which are very good but we see that XGBoost was outperformed by GBM.

7 Implementation of LSTM model

7.1 Architecture and Sequential Dependencies

LSTMs (Long Short-Term Memory networks) are a type of RNN (Recurrent Neural Network) designed to capture long-term dependencies in sequential data. They address the vanishing gradient problem common in standard RNNs, enabling the effective modeling of both long- and short-term dependencies.

Key Components:

- **Cell State:** Acts as the memory of the network, retaining information over time.
- **Input Gate:** Determines what new information to store in the cell state.
- **Forget Gate:** Controls which information to discard from the cell state.
- **Output Gate:** Decides the output at each time step based on the cell state and hidden state.

These gates work together to allow LSTMs to selectively remember or forget information, making them suitable for tasks like time series forecasting, language modeling, and more.

7.2 Hyperparameter Tuning in LSTMs

Key hyperparameters that influence LSTM performance:

1. **Batch Size:** Determines the number of samples processed before the model updates its weights. Larger batch sizes reduce noise but require more memory.
2. **Epochs:** The number of times the entire dataset is passed through the network during training. Optimal values depend on convergence rates.
3. **Dropout:** Adds regularization by randomly ignoring a fraction of neurons during training, preventing overfitting.
4. **Learning Rate:** Controls how much to adjust model weights during backpropagation.
5. **Hidden Units:** The number of neurons in each LSTM layer, which influences the model's capacity to learn patterns.

Proper tuning of these parameters is essential to balance model performance and computational efficiency.
So the next thing was to implement the model.

```
import pandas as pd
import numpy as np
import keras
from keras.src.layers import Dropout
from tensorflow.keras.preprocessing.sequence import
   TimeseriesGenerator
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_absolute_error
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Input
from sklearn.metrics import r2_score
```

```
file_path = 'C:\our_file_path_to_the_dataset\cleaned_energy_data.csv
    '
df = pd.read_csv(file_path)

target_column = 'Appliances'
features = df.columns.difference([target_column])

df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute

df = df.select_dtypes(include=['float64', 'int64', 'int32'])
features = df.select_dtypes(include=['float64', 'int64']).columns

scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(df[features])

sequence_length = 50
X = []
y = []

for i in range(sequence_length, len(scaled_data)):
    X.append(scaled_data[i - sequence_length:i])
    y.append(df[target_column].iloc[i])

X = np.array(X)
y = np.array(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.2, random_state=42)

def build_model(units=50, optimizer='adam', dropout=0.2):
    model = Sequential()
    model.add(Input(shape=(sequence_length, 44)))
    model.add(LSTM(units, return_sequences=False))
    model.add(Dropout(dropout))
    model.add(Dense(1))
    model.compile(optimizer, loss='mse', metrics=['mae'])
    return model

 model = build_model()

history = model.fit(
    X_train, y_train,
    epochs=150,
    batch_size=8,
    validation_split=0.2,
    verbose=0
)

test_loss, test_mae = model.evaluate(X_test, y_test, verbose=0)

print(f"Test Loss: {test_loss:.4f}")
print(f"Test MAE: {test_mae:.4f}")
print(f"Test RMSE: {np.sqrt(test_loss):.4f}")
print(f"Test MAPE: {test_mae/np.mean(y_test):.4f}")
print(f"Test R2: {r2_score(y_test, model.predict(X_test)):.4f}")

pred = model.predict(X_test)
```

So after implementing and evaluating the model we got the following results:

- RMSE: 13.0474
- MAE: 11.1328
- MAPE (%): 0.2096
- R^2 :-1.3814

which seemed good! Now what was left , the hyper-parameter tuning of the model:

```
units_list = [50, 100, 150]
dropout_list = [0.2, 0.3, 0.5]
batch_size_list = [16, 32]
epochs_list = [10, 20]
optimizer_list = ['adam', 'sgd']

    def train_and_evaluate_model(units, dropout, optimizer,
       batch_size, epochs):
    model = build_model(units, optimizer, dropout)

    model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size
       , validation_split=0.1, verbose=0)

    y_pred = model.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100
    r2 = r2_score(y_test, y_pred)

    return mae, rmse, mape, r2
```

```
best_mae = float('inf')
best_rmse = float('inf')
best_mape = float('inf')
best_r2 = float('-inf')

best_params = {
    'mae': {},
    'rmse': {},
    'mape': {},
    'r2': {}
}

for units in units_list:
    for dropout in dropout_list:
        for batch_size in batch_size_list:
            for epochs in epochs_list:
                for optimizer in optimizer_list:
                    print(f"Training model with units={units},
                       dropout={dropout}, batch_size={batch_size},
                       epochs={epochs}, optimizer={optimizer}")

                    mae, rmse, mape, r2 = train_and_evaluate_model(
                       units, dropout, optimizer, batch_size, epochs
                       )


                    if mae < best_mae:
                        best_mae = mae
                        best_params['mae'] = {
                            'units': units,
                            'dropout': dropout,
                            'batch_size': batch_size,
                            'epochs': epochs,
                            'optimizer': optimizer
                        }


                    if rmse < best_rmse:
                        best_rmse = rmse
                        best_params['rmse'] = {
                            'units': units,
                            'dropout': dropout,
                            'batch_size': batch_size,
                            'epochs': epochs,
                            'optimizer': optimizer
                        }
                    if mape < best_mape:
                        best_mape = mape
                        best_params['mape'] = {
                            'units': units,
                            'dropout': dropout,
                            'batch_size': batch_size,
                            'epochs': epochs,
                            'optimizer': optimizer
                        }


                    if r2 > best_r2:
                        best_r2 = r2
                        best_params['r2'] = {
                            'units': units,
                            'dropout': dropout,
                            'batch_size': batch_size,
                            'epochs': epochs,
                            'optimizer': optimizer
                        }


print("\nBest hyperparameters found:")
print(f"Best MAE: {best_mae} with parameters {best_params['mae']}")
print(f"Best RMSE: {best_rmse} with parameters {best_params['rmse']}
    ")
print(f"Best MAPE: {best_mape}% with parameters {best_params['mape
    ']}")
print(f"Best $R^2$ : {best_r2} with parameters {best_params['r2']}")
```

The results of the hyper-parameter tuning were:

- **Best hyper-parameters found:**
  - {units: 50, dropout: 0.2, batch_size: 16, epochs: 10, optimizer: sgd}
- **Best MAE:** 6.9401490688323975 with parameters {units: 100, dropout: 0.2, batch_size: 32, epochs: 20, optimizer: sgd}
- **Best RMSE:** 8.415918064181458 with parameters {units: 100, dropout: 0.2, batch_size: 16, epochs: 20, optimizer: sgd}
- **Best MAPE:** 13.039987799312389% with parameters {units: 100, dropout: 0.2, batch_size: 32, epochs: 20, optimizer: sgd}
- **Best R^2:** 0.009186625480651855 with parameters {units: 100, dropout: 0.2, batch_size: 16, epochs: 20, optimizer: sgd}

8 Implementation of GRU model

8.1 Comparing GRUs to LSTMs

GRUs (Gated Recurrent Units) are a simplified version of LSTMs designed to model sequential data efficiently. They have **two gates**—the *reset gate* and the *update gate*—compared to LSTMs' three gates. GRUs combine the *cell state* and *hidden state* into one, reducing complexity and making them faster to train.

- **Reset Gate:** Decides how much of the past information to forget.
- **Update Gate:** Balances between keeping old information and incorporating new input.

GRUs are computationally lighter than LSTMs, perform well on shorter sequences, and are widely used in tasks like time series forecasting and natural language processing.

For the GRU , the second deep learning model implemented , we knew things were going to be different getting into this.The first part of the preprocessing was the same as before with simple code needed:

```
file_path='C:\our_file_path_to_the_dataset\cleaned_energy_data.csv'
df = pd.read_csv(file_path)

X = df.drop(columns=["Appliances"])
y = df["Appliances"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.2, random_state=42)
```

However , there are some key details this time. Firstly , our dataset. Since our dataset is a Timeseries , which we know from the nature of the data:

- Data over some period of time
- The exact date of the recorder data is critical information
- The data is chronologically ordered

We do know that GRUs are perfect for Handling Timeseries so we know that on this side of things we are going to be giving it special treatment.
After splitting the dataset into train and test subsets , we have to scale the data and then prepare the time series data for the GRU model.

```
mms = MinMaxScaler()
train = y_train.values.reshape(-1, 1)
test = y_test.values.reshape(-1, 1)
```

```
scaled_train = mms.fit_transform(train)
scaled_test = mms.transform(test)

sequence_length = 12
generator = TimeseriesGenerator(scaled_train, scaled_train, length=
   sequence_length, batch_size=1)
```

As you can see we used the TimeseriesGenerator function from the preprocessing.sequence library of keras , tensorflow.
What's left is to define the model and then compile it and train it and lastly we define a function to get the predictions of the model and evaluate them with the same known metrics.

The final code is this:

```
import pandas as pd
import numpy as np
from tensorflow.keras.preprocessing.sequence import
   TimeseriesGenerator
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GRU
from sklearn.metrics import mean_squared_error, mean_absolute_error,
    r2_score

file_path='C:\our_file_path_to_the_dataset\cleaned_energy_data.csv'
df = pd.read_csv(file_path)

X = df.drop(columns=["Appliances"])
y = df["Appliances"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.2, random_state=42)

mms = MinMaxScaler()
train = y_train.values.reshape(-1, 1)
test = y_test.values.reshape(-1, 1)
scaled_train = mms.fit_transform(train)
scaled_test = mms.transform(test)

sequence_length = 12
generator = TimeseriesGenerator(scaled_train, scaled_train, length=
   sequence_length, batch_size=1)

def gru_model():
    model = Sequential()
    model.add(GRU(units=120, activation="relu", input_shape=(
       sequence_length, 1)))
    model.add(Dense(1))
    model.compile(optimizer="rmsprop", loss="mse")
    return model

gru = gru_model()
gru.summary()
early_stopping = EarlyStopping(monitor="loss", min_delta=1e-5,
   patience=5, mode="min")
gru.fit(generator, epochs=50, callbacks=[early_stopping])

def get_model_predictions(train, test, model):
    y_hat = []
    batch = train[-sequence_length:].reshape((1, sequence_length, 1)
       )
    for i in range(len(test)):
        new_pred = model.predict(batch)[0]
        y_hat.append(new_pred)
        batch = np.append(batch[:, 1:, :], [[new_pred]], axis=1)
    return np.array(y_hat).reshape(-1, 1)

gru_pred = get_model_predictions(train=scaled_train, test=
   scaled_test, model=gru)
gru_pred_unscaled = mms.inverse_transform(gru_pred)

rmse = mean_squared_error(y_test, gru_pred_unscaled, squared=False)
mae = mean_absolute_error(y_test, gru_pred_unscaled)

y_test_1d = y_test.flatten() if y_test.ndim > 1 else y_test
gru_pred_unscaled_1d = gru_pred_unscaled.flatten() if
   gru_pred_unscaled.ndim > 1 else gru_pred_unscaled

mape = (abs((y_test_1d - gru_pred_unscaled_1d) / y_test_1d).mean())
   * 100
r2 = r2_score(y_test, gru_pred_unscaled)

print(f"RMSE: {rmse:.4f}")
print(f"MAE: {mae:.4f}")
print(f"MAPE: {mape:.2f}%")
print(f"R^2: {r2:.4f}")
```

It must be noted that despite defining an EarlyStopping mechanism the code did take its time to run , with both the compile-train and the evaluating cells taking more time to run comparing to the other models.

Despite all that , the metrics results that we got were:

- RMSE: 9.6987
- MAE: 7.6838
- MAPE (%): 16.48%
- R^2 :-0.0256

9 Implementation of CNN model

9.1 Steps in CNN for Time Series

1. **Input Preparation:**
   - Represent the time series as a sequence of values. For example, for univariate time series:
   $$Y = [y_1, y_2, \ldots, y_n].$$
   - For multivariate time series, include multiple features for each time step to create a richer representation of the data.
2. **Convolution:**
   - Apply filters (kernels) that slide over small overlapping windows of the time series.
   - Each filter computes the weighted sum of values in its window, producing a *feature map* that highlights significant patterns in the data (e.g., trends or cycles).
3. **Activation Function:**
   - Pass the resulting feature map through an activation function (commonly ReLU) to introduce non-linearity.
   - This step emphasizes critical patterns while discarding less significant information.
4. **Pooling:**
   - Use pooling operations (e.g., max pooling or average pooling) to down-sample the feature map.
   - Pooling reduces the size of the data, keeps the most significant features, and helps avoid overfitting.
5. **Flattening:**
   - Transform the reduced feature maps into a 1D vector, which serves as input for the next step (fully connected layers).
6. **Fully Connected Layers:**
   - Pass the flattened vector through dense (fully connected) layers to learn high-level representations.

- These layers map the learned features to the desired output (e.g., regression or classification).

7. **Output:**

   - Use a final activation function appropriate for the task:
     - For regression tasks (e.g., forecasting values), use a linear activation function.
     - For classification tasks, use softmax to output class probabilities.

8. **Training:**

   - Train the model by optimizing filter weights using a loss function (e.g., Mean Squared Error for regression) and backpropagation with a suitable optimizer (e.g., Adam).

Summary

A **Convolutional Neural Network (CNN)** for time series forecasting identifies patterns in sequential data through:

- **Convolutional layers** to capture local temporal dependencies and patterns.
- **Pooling layers** to reduce data size while retaining key information.
- **Dense layers** to map learned patterns to predictions or classifications.

By extracting features efficiently, CNNs are particularly useful for tasks like time series classification or regression, offering computational efficiency and strong performance on large datasets with complex patterns.

So the next thing was to implement the model.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Dense, Flatten
from sklearn.metrics import mean_squared_error, mean_absolute_error,
    r2_score

file_path='C:\our_file_path_to_the_dataset\cleaned_energy_data.csv'
df = pd.read_csv(file_path)
df = pd.read_csv(file_path)

df = df.drop(columns=["date"])
df = pd.get_dummies(df, columns=["day_type"], drop_first=True)
```

```
X = df.drop(columns=["Appliances"])
y = df["Appliances"]

scaler = MinMaxScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.2, random_state=42)

X_train = np.expand_dims(X_train, axis=2)
X_test = np.expand_dims(X_test, axis=2)

model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu',
   input_shape=(X_train.shape[1], 1)))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))

model.compile(optimizer='adam', loss='mean_squared_error')

history = model.fit(X_train, y_train, epochs=20, batch_size=32,
   validation_data=(X_test, y_test), verbose=1)

y_pred = model.predict(X_test)

plt.figure(figsize=(12, 6))
plt.plot(y_test.values, label='Actual')
plt.plot(y_pred.flatten(), label='Predicted')
plt.title('CNN Multivariate Time Series Forecasting')
plt.xlabel('Time Step')
plt.ylabel('Value')
plt.legend()
plt.show()
```

And now for the model evaluation:

```
rmse = mean_squared_error(y_test, y_pred, squared=False)
mae = mean_absolute_error(y_test, y_pred)
mape = np.mean(np.abs((y_test - y_pred.flatten()) / y_test)) * 100
r2 = r2_score(y_test, y_pred)

print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
print(f"MAPE: {mape}%")
print(f"$R^2$  Score: {r2}")
```

The final results were:

- RMSE: 10.152999879413224
- MAE: 7.760974443875826
- MAPE (%): 15.624856117444159%
- R^2 :-0.09577891101111091

10 Implementation of Transformsers model

Since the start of this project , having read the its requirements and understood what we will have to complete we knew that the transformers model would be the most exciting part.

10.1 Overview of Transformers

Transformers are a powerful deep learning model primarily designed for sequential data, leveraging *self-attention* to identify dependencies between input elements regardless of their distance. They employ an *encoder-decoder architecture*, where the encoder processes the input sequence, and the decoder generates the output sequence. Scalability is a key advantage; parallelized computation and attention mechanisms enable handling large datasets and complex tasks efficiently.

10.2 Time Series Transformers

- **Transformer-based Models:** Adapted for time series tasks, these models replace recurrent operations with self-attention, capturing temporal patterns directly.
- **Advantages for Long-term Dependencies:** Transformers excel at learning relationships over long time horizons, as self-attention allows access to the entire sequence simultaneously without suffering from vanishing gradients.
- **Clarity with Architecture:** Key components include positional encoding (for time-step awareness), multi-head self-attention layers (for capturing dependencies), and feed-forward layers (for feature extraction). Diagrams can showcase how attention mechanisms are applied across time steps.

The steps that we had to follow were:

- Data Preprocessing
- Positional Encoding
- Transformer Encoder Layer
- Building the Model
- Training and Validation

- Evaluation

Let's dive into each section and walk-through it briefly.

Preprocessing:

- Scaling the Features and Target with MinMaxScaler
- Time-Series Reshaping:
  - The data is structured into sequences using a `look_back` window (e.g., 60 time steps). Each sequence contains data from the previous 60 time steps to predict the next value.
  - This reshaping creates 3D input data of the shape [samples, `look_back` , features].

Positional Encoding:

- `Why positional encoding , somebody may ask`
  - Transformers, unlike RNNs (e.g., GRU, LSTM), lack a built-in mechanism to understand the sequential order of data. Positional encoding introduces relative positional information to the model by adding sinusoidal values to the input embeddings.
- `How It Works?`
  - A sinusoidal function computes unique positional encodings for each time step using the `get_angles` method. These encodings are pre-computed by the `positional_encoding` method and stored in `self.pos_encoding`. The `call` method then adds the positional encodings to the input sequence.

`In this part , this piece of code should be highlighted`

```
    class PositionalEncoding(Layer):
    def __init__(self, max_len, embed_dim, **kwargs):
        super(PositionalEncoding, self).__init__(**kwargs)
        self.max_len = max_len
        self.embed_dim = embed_dim
        self.pos_encoding = self.positional_encoding(max_len,
           embed_dim)

    def positional_encoding(self, position, d_model):
        angle_rads = self.get_angles(
            np.arange(position)[:, np.newaxis],
            np.arange(d_model)[np.newaxis, :],
            d_model
        )
        angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
        angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
        pos_encoding = angle_rads[np.newaxis, ...]
        return tf.cast(pos_encoding, dtype=tf.float32)

    def call(self, inputs):
        seq_len = tf.shape(inputs)[1]
        return inputs + self.pos_encoding[:, :seq_len, :]
```

Transformer Encoder Layer:
The Transformer encoder processes the input sequences using:

- Multi-Head Attention
- Feed-Forward Network
- Residual Connections and Layer Normalization

Building the Model:
The model consists of:

- An Input Layer
- Embedding and Positional Encoding
- Transformer Encoder Blocks
- Global Average Pooling
- Output Layer

Training and Validation:
The training and validation part was fairly easy , since the code we used was almost identical to the one from other models too:

```
    history = model.fit(
    train_gen,
    validation_data=test_gen,
    epochs=20,
    batch_size=32,
    verbose=1
)
```

Lastly , what was left was the evaluation of the model:

```
rmse = mean_squared_error(y_test, model.predict(X_test), squared=
   False)
print(f"Transformer RMSE: {rmse}")
y_pred = model.predict(X_test).flatten()

mae = mean_absolute_error(y_test, y_pred)
print(f"MAE: {mae}")

mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100
print(f"MAPE: {mape}%")

r2 = r2_score(y_test, y_pred)
print(f"$R^2$ Score: {r2}")
```

The final code is this:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Model
from tensorflow.keras.layers import (
    Input, Dense, Dropout, LayerNormalization,
       GlobalAveragePooling1D
)
from tensorflow.keras.layers import MultiHeadAttention, Add
from tensorflow.keras.optimizers import Adam
import tensorflow as tf
from tensorflow.keras.layers import Layer
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error, r2_score

file_path='C:\our_file_path_to_the_dataset\cleaned_energy_data.csv'
df = pd.read_csv(file_path)

class PositionalEncoding(Layer):
    def __init__(self, max_len, embed_dim, **kwargs):
        super(PositionalEncoding, self).__init__(**kwargs)
        self.max_len = max_len
        self.embed_dim = embed_dim
        self.pos_encoding = self.positional_encoding(max_len,
           embed_dim)
    def get_config(self):
        config = super(PositionalEncoding, self).get_config()
        config.update({
            'max_len': self.max_len,
            'embed_dim': self.embed_dim,
        })
        return config

    def positional_encoding(self, position, d_model):
        angle_rads = self.get_angles(
            np.arange(position)[:, np.newaxis],
            np.arange(d_model)[np.newaxis, :],
            d_model
        )
        angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
        angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
        pos_encoding = angle_rads[np.newaxis, ...]
        return tf.cast(pos_encoding, dtype=tf.float32)

    def get_angles(self, pos, i, d_model):
        angle_rates = 1 / np.power(
            10000,
            (2 * (i // 2)) / np.float32(d_model)
        )
        return pos * angle_rates

    def call(self, inputs):
        seq_len = tf.shape(inputs)[1]
        return inputs + self.pos_encoding[:, :seq_len, :]

def generate_time_series(n_timesteps):
    x = np.arange(n_timesteps)
    y = np.sin(0.02 * x) + np.random.normal(scale=0.5, size=
       n_timesteps)
    return y

def transformer_encoder(inputs, head_size, num_heads, ff_dim,
   dropout=0.1):
    x = MultiHeadAttention(key_dim=head_size, num_heads=num_heads,
       dropout=dropout)(inputs, inputs)
    x = Add()([x, inputs])
    x = LayerNormalization(epsilon=1e-6)(x)

    x_ff = Dense(embed_dim, activation='relu')(x)
    x_ff = Dense(embed_dim)(x_ff)
    x = Add()([x_ff, x])
    x = LayerNormalization(epsilon=1e-6)(x)
    return x

n_timesteps = 1000
look_back = 46
batch_size = 32
epochs = 20

series = generate_time_series(n_timesteps)

X = df.drop(columns=["Appliances", "date"])
X = X.select_dtypes(include=[np.number])

y = df["Appliances"].astype(float)

scaler = MinMaxScaler()
X = scaler.fit_transform(X)

if len(X.shape) == 2:
    X = X.reshape((X.shape[0], X.shape[1], 1))

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:]

input_shape = X_train.shape[1:]
inputs = Input(shape=input_shape)

embed_dim = 32
x = Dense(embed_dim)(inputs)

x = PositionalEncoding(max_len=look_back, embed_dim=embed_dim)(x)

x = transformer_encoder(x, head_size=embed_dim, num_heads=4, ff_dim
   =128, dropout=0.1)
x = transformer_encoder(x, head_size=embed_dim, num_heads=4, ff_dim
   =128, dropout=0.1)

x = GlobalAveragePooling1D()(x)

outputs = Dense(1)(x)

model = Model(inputs, outputs)
model.compile(optimizer=SGD(learning_rate=1e-4, momentum=0.9), loss=
   'mean_squared_error')

history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=epochs,
    batch_size=batch_size,
    verbose=1
)

rmse = mean_squared_error(y_test, model.predict(X_test), squared=
   False)
print(f"Transformer RMSE: {rmse}")
y_pred = model.predict(X_test).flatten()

mae = mean_absolute_error(y_test, y_pred)
print(f"MAE: {mae}")

mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100
print(f"MAPE: {mape}%")

r2 = r2_score(y_test, y_pred)
print(f"$R^2$ Score: {r2}")
```

The evaluation of the model gave us:

- RMSE: 10.050828386913619
- MAE: 8.46144045316256
- MAPE (%): 16.341890677427635%
- R^2 :-0.0042492096164044035

11 Results and Discussion

11.1 Model Performance

The performance of the model will be assessed using four key metrics: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and the Coefficient of Determination (R^2). These metrics will be computed at the conclusion of the modeling process to provide quantitative insights into the model's accuracy and predictive capabilities.

In addition to these numerical evaluations, the performance will also be analyzed through a graphical representation of actual versus predicted values. This visualization serves as a complementary diagnostic tool, helping to identify potential issues like overfitting or underfitting that may not be apparent through metrics alone. While RMSE, MAE, MAPE, and R^2 are widely used indicators of model performance, they can sometimes provide a misleading perspective when considered in isolation. For example, a high R^2 value might suggest a well-performing model, but the actual versus predicted graph could reveal systematic errors or inconsistencies, indicating areas for further refinement.

By combining both metric-based evaluations and visual analysis, a more holistic understanding of the model's performance can be achieved, ensuring that any limitations or biases are appropriately addressed and mitigated.

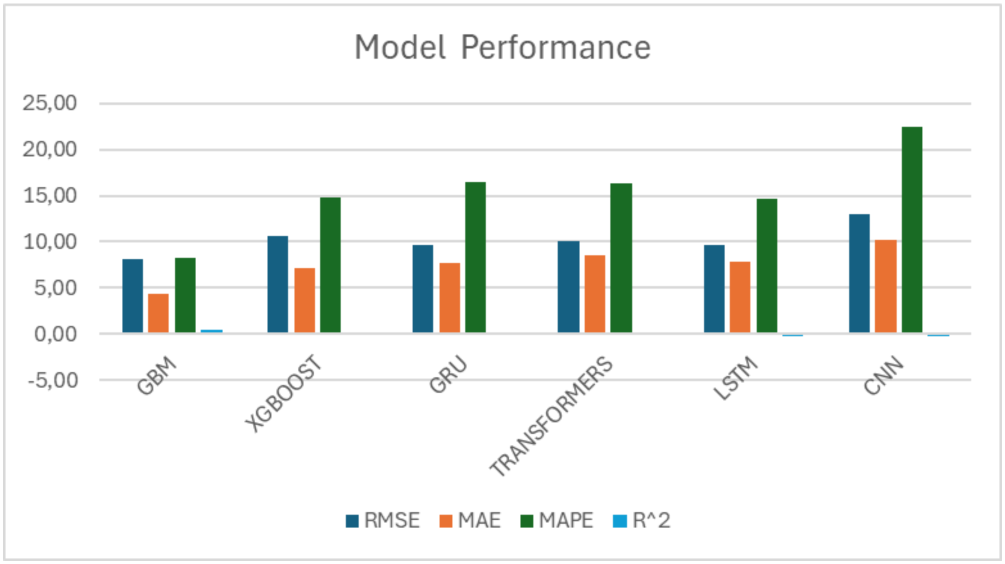
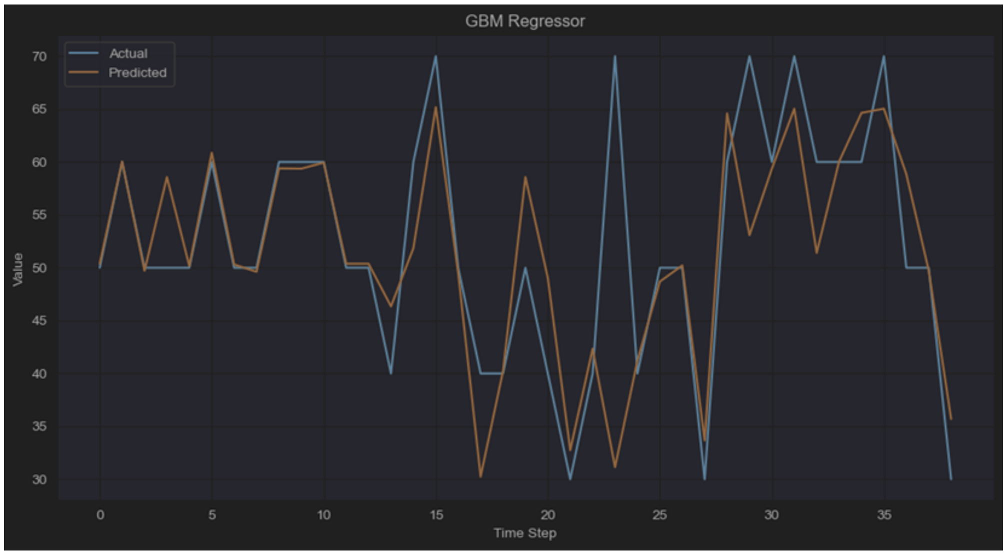


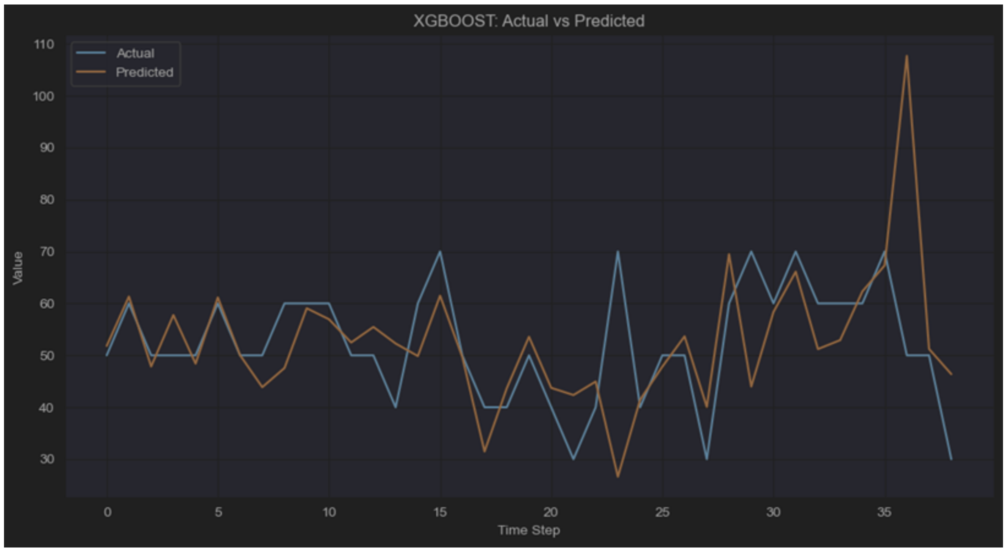
Figure 1: Performance Comparison Across Models

Based on the metrics, the two best-performing models are GBM and LSTM, with XGBoost, GRU, and Transformers closely following. However, CNN lags significantly behind, showing the weakest performance. While these metrics provide valuable insights and highlight the top contenders, they cannot be the sole determinants of model effectiveness. Metrics alone fail to capture the flow and dynamics of the data, which are equally critical in assessing a model's overall performance. A deeper analysis of how each model aligns with the trends and patterns in the data is essential to gain a comprehensive understanding of their true capabilities.

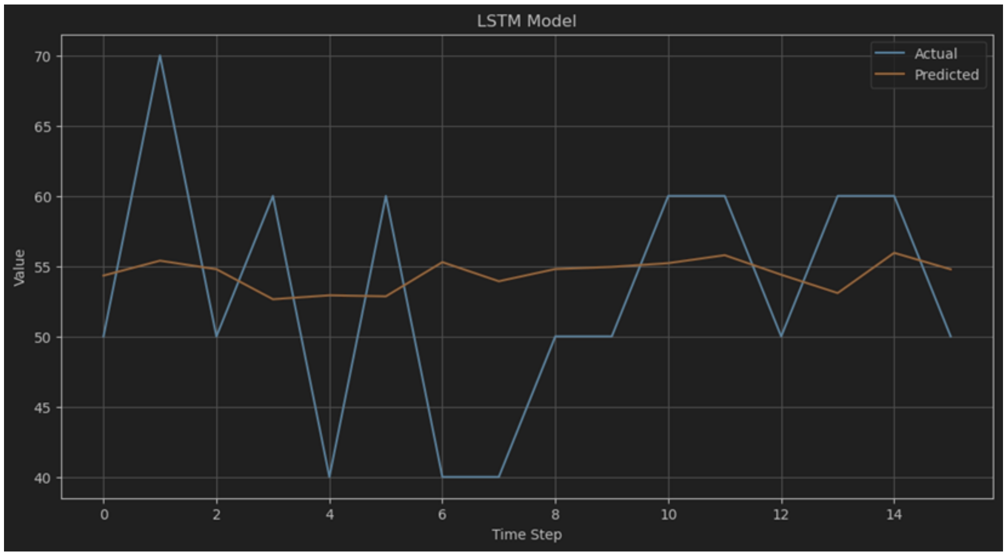
11.2 Plots



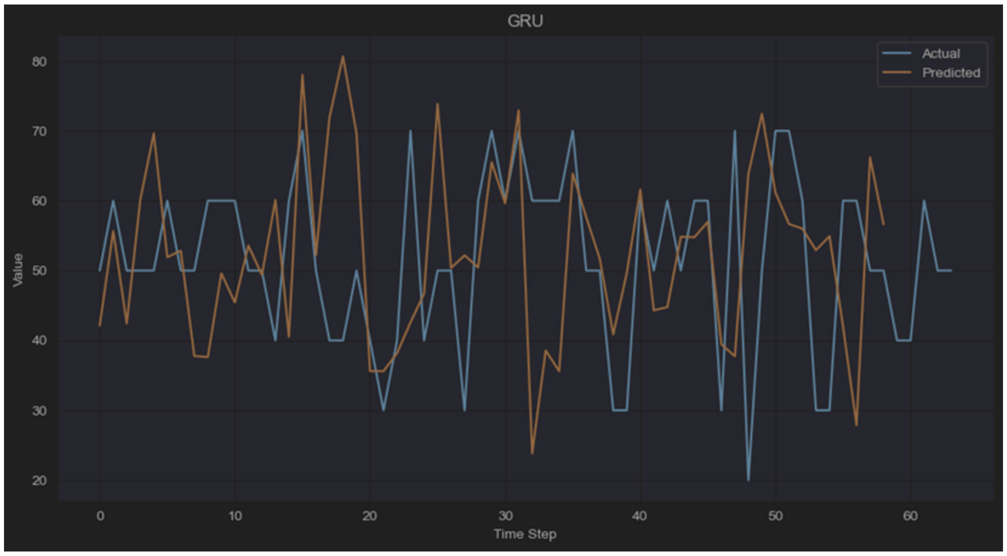
The GBM model shows a reasonable ability to follow overall data trends but struggles with extreme peaks and troughs, resulting in higher error metrics. Residuals cluster around zero but include significant outliers, indicating areas where the model fails to capture data complexity. Refinements like hyperparameter tuning could improve performance.



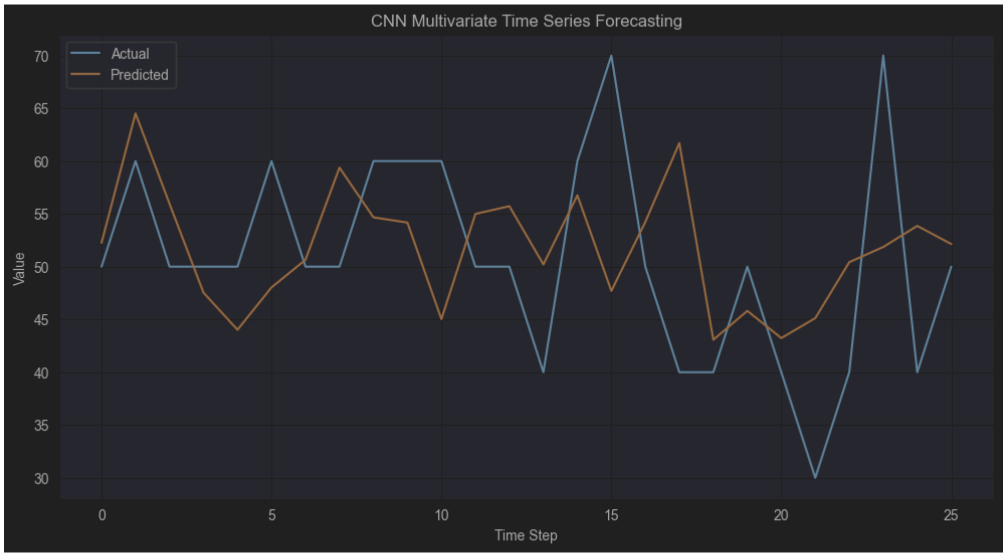
The XGBoost model captures general trends effectively but shows discrepancies during extreme variations, such as underestimating sharp peaks. Residuals are mostly centered around zero but include notable outliers and slight patterns of systematic bias. Adjustments in features or hyperparameters could enhance accuracy.



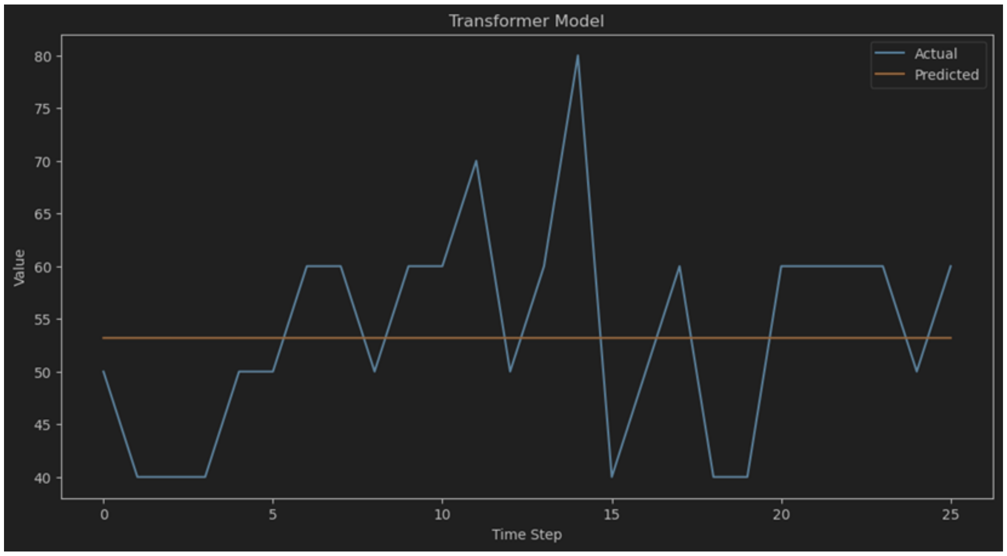
The LSTM model produces smooth predictions that miss sharp changes in the data, suggesting underfitting. Residuals show a clear pattern of consistent errors, especially for highly variable points. Increasing model capacity or enhancing features could improve performance.



The GRU model follows general fluctuations but fails to replicate sharp peaks and troughs accurately, leading to significant errors. Residuals show a wide spread, with outliers indicating specific data points where the model struggles. Improvements in architecture or data preprocessing could help.



The CNN model captures basic trends but struggles with variability and sharp fluctuations, leading to large deviations. Residuals show a wide spread and patterns, suggesting systematic errors. Architectural tuning or feature enhancements could improve predictions.



The Transformer model performs poorly, with predictions showing minimal response to data variability. Residuals reveal systematic underperformance, indicating a lack of adaptation to time series dynamics. Architectural changes or additional training data might help.

11.3 Why does the LSTM model have good metrics but bad overall performance?

The LSTM model produces favorable metrics (like low RMSE, MAE, and MAPE) but shows poor performance in graphs due to several factors:

Smoothing of Predictions

LSTM models often smooth predictions, focusing on long-term trends while neglecting sharp changes in the data. This leads to a flatter trend in predictions, which may seem accurate on average but fails to capture variability.

Metrics Masking Performance Issues

Metrics such as RMSE and MAE calculate average errors. If the LSTM predicts values close to the overall mean, the errors for most points will be small, leading to deceptively good metrics, even if the model misses peaks and troughs.

Inability to Handle High-Frequency Variations

LSTMs struggle with abrupt or high-frequency variations in data, especially when the training set lacks sufficient examples of such patterns. This results in predictions that prioritize general trends over short-term dynamics.

Effects of Regularization

Regularization techniques, such as dropout or early stopping, can cause underfitting, preventing the LSTM from learning intricate patterns. This leads to oversimplified and generalized predictions.

Limited Dataset for Evaluation

A limited or unrepresentative test dataset can skew metric calculations, making them appear better than they are. Graphs, however, reveal the model's inability to replicate the nuances of the actual data.

Summary

LSTMs achieve good metrics because they average out errors across the dataset, minimizing penalties for small deviations but failing to capture the data's full complexity. This results in poor graphical performance despite seemingly accurate metrics.

11.4 Best performance

Based on the analysis, the Gradient Boosting Model (GBM) and XGBoost demonstrate the best performance among the evaluated models. The GBM shows strong alignment with the actual values, effectively capturing data trends with fewer deviations. Its residuals are scattered around the zero-error line with minimal extreme outliers, indicating consistent performance. Similarly, XGBoost closely follows actual data trends, adapting well to fluctuations. The residuals are relatively small and lack significant patterns, showing that the model generalizes effectively without systematic errors. While both models could benefit from further tuning to better handle extreme variations, their performance is clearly superior to models like GRU, LSTM, Transformers, and CNN.

11.5 Deep Learning and Transformer Models Benefits and limitations

11.5.1 Deep learning

Deep learning models, including CNNs, RNNs, LSTMs, and GRUs, offer significant benefits due to their flexibility and ability to learn complex, non-linear relationships in data. They automatically extract features from raw data, reducing the need for manual feature engineering, which is particularly advantageous for unstructured data like text, images, and audio. These models scale effectively with large data sets, often improving as more data becomes available, and they achieve state-of-the-art performance in fields like computer vision, natural language processing, and reinforcement learning. Additionally, models like LSTMs and GRUs are well-suited for handling sequential data, making them ideal for tasks such as time series forecasting, speech recognition, and text analysis. However, deep learning models come with limitations. They require substantial computational resources, often needing GPUs or TPUs for efficient training. Overfitting can be a concern, especially with small or non-diverse datasets, and their reliance on large amounts of labeled data can be a challenge in domains where such data is scarce. These models are often criticized for their lack of interpretability, as their predictions and learned features can be difficult to understand. Furthermore, training deep learning models is complex, requiring extensive hyperparameter tuning and experimentation, which can be both time-consuming and resource intensive

11.5.2 Transformer Models

Transformer models offer several advantages. They process input data in parallel, enabling faster training compared to traditional sequential models like LSTMs and GRUs. Their self-attention mechanisms allow them to effectively capture long-term dependencies, making them highly suitable for tasks such as machine translation, text summarization, and sequence modeling. Transformers scale efficiently with larger datasets and model sizes, as seen in architectures like GPT and BERT, and are versatile across domains, from natural language processing to computer vision and time series forecasting. Additionally, many pretrained transformer models, such as BERT and GPT, are available for fine-tuning on specific tasks, even with relatively small datasets. Despite these benefits, transformers have notable limitations. They require significant computational resources and memory, making them expensive to train and dependent on high-end hardware like GPUs or TPUs. Their performance is highly reliant on large datasets, and they may struggle in low-data scenarios unless pretrained. Transformers can become exceedingly large, posing challenges for deployment on devices with limited resources. Like other deep learning models, they are often criticized for their lack of interpretability, making it difficult to explain predictions. Training transformers from scratch is also highly complex, requiring careful management of hyperparameters, optimization methods, and computational resources.

11.6 Discussion

Overfitting happens when a model performs well on training data but poorly on unseen data, often due to excessive model complexity or insufficient data. Metrics suggest that

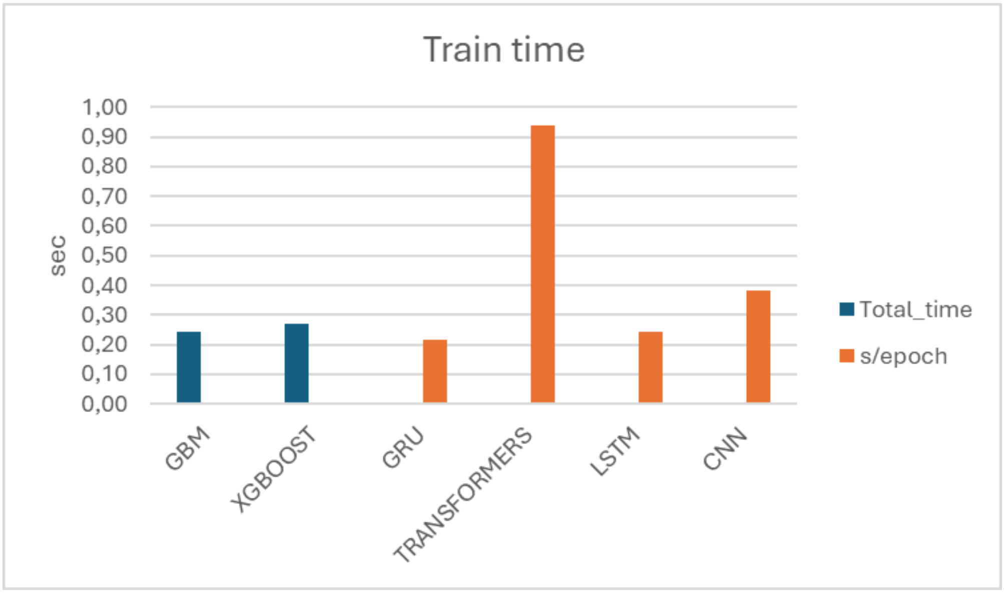
GRU, Transformers, LSTM, and CNN struggle with generalization, as indicated by low or negative R^2 scores, which may point to overfitting. On the other hand, GBM and XGBoost perform better, with GBM achieving an R^2 score of 0.45, reflecting the effectiveness of tree-based methods with built-in regularization techniques. For deep learning models, techniques like dropout, L2 regularization, early stopping, increasing dataset size, and reducing model complexity can help mitigate overfitting. For GBM and XGBoost, tuning hyperparameters such as learning rate, tree depth, and tree count can further optimize performance.

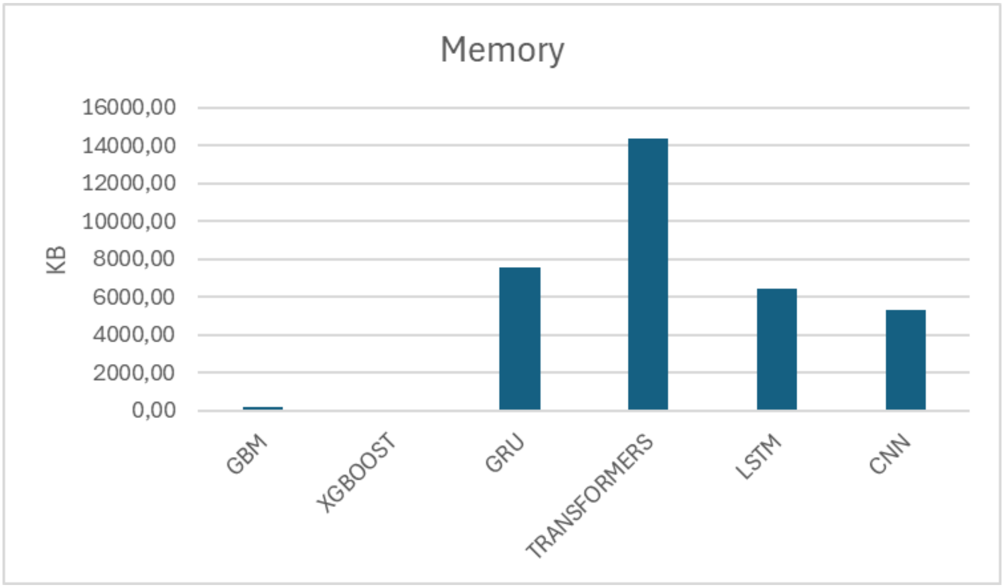
Model training time is crucial for practical deployment. GBM and XGBoost exhibit minimal training times (0.24 and 0.27 seconds, respectively), making them computationally efficient. In contrast, deep learning models like Transformers, LSTM, and CNN require significantly longer training times, with Transformers being the most time-intensive at 0.94 seconds per epoch. To reduce deep learning training times, techniques such as using pre-trained models, leveraging hardware accelerations like GPUs or TPUs, and employing dynamic learning rate schedules can be effective. In time-constrained scenarios, tree-based models like GBM and XGBoost provide faster results while maintaining strong performance.

The volume and quality of data play a key role in model performance. Deep learning models like GRU, Transformers, LSTM, and CNN require large and diverse datasets to perform optimally, and their lower R^2 scores suggest potential struggles with data limitations. Conversely, GBM and XGBoost are better suited for smaller datasets, as shown by their lower RMSE and higher R^2 scores. Enhancing deep learning models could involve augmenting datasets with synthetic data, applying techniques like SMOTE, or using transfer learning with pre-trained models. For tree-based models, emphasizing feature engineering and selection can maximize data utility.

GBM offers a balance between computational efficiency and predictive performance, making it ideal for scenarios with limited resources and smaller datasets. Deep learning models like GRU, LSTM, and Transformers excel in capturing complex patterns when provided with ample data and computational power, but their success depends on addressing overfitting and resource demands. XGBoost is a strong alternative for structured data, offering good performance and fast training times, though additional tuning may be needed for optimal accuracy. Ultimately, model selection should align with application requirements, including data volume, computational constraints, and the trade-off between interpretability and performance.

Below are the training times for all the models as well as their memory usage:





12 Conclusion and Recommendations

12.1 Models Evaluated

12.1.1 Gradient Boosting Models (GBM and XGBoost)

Strengths:

- Both GBM and XGBoost demonstrated strong predictive performance, aligning closely with the actual energy consumption values.

- These models excel at capturing non-linear relationships between features and the target variable, making them ideal for tabular data.
- They are robust to small datasets, as their tree-based architecture can efficiently learn patterns without requiring excessive computational resources.

Weaknesses:

- Limited in capturing time-dependent patterns unless engineered features (e.g., lagged variables) are provided.
- May suffer from overfitting if hyperparameters (e.g., learning rate, number of trees) are not carefully tuned.

Performance Observations:

- The graphs show that GBM and XGBoost performed exceptionally well in steady-state conditions, but small deviations might occur in rapidly changing patterns.
- Both models produced reliable predictions and minimal deviations, making them suitable for energy prediction in real-world applications.

12.1.2 GRU (Gated Recurrent Unit)

Strengths:

- GRU captured sequential dependencies in the dataset, which is valuable for time-series data.
- It is computationally less expensive compared to LSTM while providing competitive performance.

Weaknesses:

- GRU struggled with extreme fluctuations in the data, as seen in cases of rapid spikes or dips in energy consumption.
- Requires a larger dataset to achieve optimal performance and minimize overfitting.

Performance Observations:

- GRU managed to follow the general trend of the data but missed some sharp transitions.
- Predictions smoothed over extreme changes, indicating that the model did not fully capture all the variations in the data.

12.1.3 LSTM (Long Short-Term Memory)

Strengths:

- LSTM is designed to handle long-term dependencies, making it suitable for time-series forecasting.
- It is effective at capturing patterns where temporal relationships play a critical role.

Weaknesses:

- Similar to GRU, LSTM struggled with rapid fluctuations and seemed to oversmooth predictions.
- Requires careful tuning of hyperparameters (e.g., hidden layers, sequence length) and a larger dataset to outperform traditional models.

Performance Observations:

- LSTM provided stable and consistent predictions but missed some fine-grained details in the data.
- Predicted values appeared smoother compared to the actual data, indicating an averaging effect.

12.1.4 Transformer Model

Strengths:

- Transformer models are powerful for capturing complex temporal relationships and patterns in very large datasets.
- They handle long-term dependencies better than traditional RNNs like GRU and LSTM.

Weaknesses:

- Transformers underperformed in this case, likely due to the dataset's size and lack of extensive temporal complexity.
- Predictions appeared overly smoothed, suggesting that the model failed to capture short-term variations.

Performance Observations:

- The Transformer model struggled to differentiate between variations in the data, leading to a flattened prediction line.
- Its potential was not fully realized, indicating the need for larger datasets or advanced architectures tailored to the task.

12.1.5 CNN (Convolutional Neural Network)

Strengths:

- CNN models can effectively learn spatial patterns, and when adapted to time-series data, they can capture local dependencies.
- They are computationally efficient compared to RNN-based models.

Weaknesses:

- CNN alone may not fully capture sequential dependencies, leading to oversmoothing of predictions.
- Performance is sensitive to the choice of kernel size and architecture.

Performance Observations:

- CNN captured general trends in the data but struggled with high variability and sharp changes.
- While the model produced reasonable predictions, it did not outperform GRU or traditional models.

12.2 Conclusion

12.2.1 Key Findings

- **Best Model**: Gradient Boosting Models (GBM and XGBoost) were the most effective for this dataset. They produced predictions that closely aligned with the actual values while requiring relatively little data preprocessing.
- **Deep Learning vs. Traditional Models**:
  - Traditional models (GBM/XGBoost) outperformed deep learning models due to the dataset's small size and tabular nature.
  - Deep learning models like GRU and LSTM showed potential but struggled with capturing rapid fluctuations in the data.
  - Transformer and CNN models underperformed, suggesting that they require larger datasets and additional tuning to match the performance of simpler models.

12.3 Recommendations

12.3.1 Practical Implications for Stakeholders

- For immediate implementation, **GBM or XGBoost** should be used due to their reliability and interpretability.
- Deep learning models should be considered if the dataset is expanded or if sequential patterns become more prominent in future tasks.

12.3.2 Potential Improvements

- **Hybrid Models**: Combine GBM/XGBoost with deep learning models (e.g., GRU or LSTM) to leverage the strengths of both approaches.

13 Future Work

13.1 Advanced Transformer Architectures

- Experiment with **Temporal Fusion Transformers (TFT)** or **Informer models**, which are specifically designed for time-series forecasting.

13.2 Hybrid Approaches

- Develop hybrid models that integrate **CNNs for feature extraction** with **GRU/LSTM/Transformers** for sequential learning.
- Ensemble models combining traditional methods (e.g., GBM/XGBoost) with deep learning could improve overall accuracy.

13.3 Scalability and Accuracy

- Expand the dataset with more samples and diverse features to improve model generalization.
- Conduct rigorous cross-validation to evaluate model robustness across different subsets of the data.

13.4 Deployment and Monitoring

- Deploy models in a real-time system to test performance on live data.
- Implement monitoring systems to track model performance and adapt to evolving patterns.

This comprehensive analysis underscores the importance of choosing the right model based on the dataset's characteristics and the task's requirements. While GBM and XGBoost emerged as the most practical solutions, deep learning models hold promise for future applications with larger datasets and more complex temporal patterns.

Sources

Of course, the most important source of information was the class' Piazza page, where we found very helpful resources for each model.

- https://piazza.com/class/ly07lzv5irr7h2

For the creation of this report, we got great help from this custom GPT:

- https://chatgpt.com/g/g-4S7zjQ7PH-latex-transformer

For Section 3 ”Literature Review”:

- https://machinelearningmastery.com/time-series-prediction-with-deep-learning-in-pyt
- https://huggingface.co/docs/transformers/model_doc/time_series_transformer
- https://towardsai.net/p/l/time-series-regression-using-transformer-models-a-plain-e
- https://ieeexplore.ieee.org/document/10583885
- https://arxiv.org/pdf/2205.01138