Department of Computer Science

IT3212 - Data-Driven Software

# Assignment 3

*Authors:*
Birk Strand Bjørnaa
Carl Edward Storlien
Christian Stensøe
Åsmund Løvoll

05.12.2024

# Table of Contents

## List of Figures

## List of Tables

## Changelog

For the revised version of Assignment 3, we have addressed the feedback and made additional improvements. Below is a list of changes made to the respective sections.

**Preprocessing and Feature Engineering**

- Added a short discussion on alternative interpolation techniques and their potential benefits.
- Updated Table 6 with all values. The previous table was truncated.
- Expanded Figure 4 to include a figure for hourly prices for a single day (4b) in addition to a figure for hourly price over two weeks (4a), making variations in electricity prices more visible.
- Added specific examples regarding business hours, weekdays, and weighting with references to Figure 4 to improve arguments.

**Bagging and boosting**

- The third paragraph of 4.1 Bagging was edited to explain the steps further and justify the hyperparameters for this algorithm.
- Added a more in-depth explanation in the 4.2 Boosting subsection on the different hyperparameters used and why.

**Transfer Learning**

- Explained our decision to use the same dataset for transfer learning and addressed the limitations.

# 1 Problem Statement

## 1.1 Forecasting Electricity Prices

Forecasting electricity prices is useful for electricity producers, the industry, and people in general. For instance, it allows energy producers to adjust their supply, the industry to schedule energy-hungry production, and the everyday person to schedule their charging of EVs, laundry, etc. Needless to say, it is also important for the energy producers for their price model.

Electricity prices are in large part a free market, meaning the prices are set based on demand and supply with little or no governmental control. At least, this is the case for the electricity markets connected to Nord Pool[1]. *"The power price is determined by the balance between supply and demand. Factors such as the weather or power plants not producing to their full capacity can impact power prices."*[2] In addition, different energy sources have different costs of generation which affect the pricing[3]. Especially in recent years in Norway, electricity prices have been up for debate because of large fluctuations and high prices.[4]

## 1.2 Task

We want to experiment with machine learning models for forecasting electricity prices. We will use datasets with historical time-series data of electricity generation sources and meteorological factors to train the models and evaluate them against the actual prices. We will try different models and techniques, and compare their results. We will also perform transfer learning by training a model on a bigger, related dataset, and leveraging this gained knowledge when training another model on our original dataset.

## 1.3 Chosen Datasets

We chose the `spanish-cities-energy-consumption` dataset for our task. The dataset is divided into two distinct datasets, `energy_dataset.csv` and `weather_features.csv`, but we intend to join these two datasets for a more complete multivariate time-series forecasting. The former dataset includes values of electricity generated by different sources, and predicted and actual electricity prices. The latter contains weather conditions for the same period for five major cities in Spain. The cities are Madrid, Bilbao, Seville, Valencia and Barcelona. Their geographic distribution is uniformly spread out over Spain. 1/3rd of Spain's population resides in these five cities, so we can assume that the electricity demand is largely influenced by the weather conditions in these five cities.

[1] *The Power Market* 2024.
[2] *The Power Market* 2024.
[3] Oğuz 2023.
[4] *Strømprisene* 2024.

# 2 Preprocessing and Feature Engineering

## 2.1 Energy Dataset

### 2.1.1 Initial Feature Selection

The preprocessing of the datasets began with an initial feature selection by removing all features that were all null (NaN). Afterward, we removed the features that were unnecessary for our analysis, `forecast wind offshore eday ahead, total load forecast, forecast solar day ahead, forecast wind onshore day ahead`.

Table 1 shows summary statistics of the features we were left with. The feature headers have been shortened in order to fit nicely in this report. Full feature headers are shown in table 2.

Table 1: Summary statistics for energy generation and load data

| | **biomass** | **fossil brown** | **fossil gas** | **fossil hard** | **fossil oil** | **hydro pumped** | **hydro river** | **hydro reservoir** |
|---|---|---|---|---|---|---|---|---|
| **count** | 35045.00 | 35046.00 | 35046.00 | 35046.00 | 35045.00 | 35045.00 | 35045.00 | 35046.00 |
| **mean** | 383.51 | 448.06 | 5622.74 | 4256.07 | 298.32 | 475.58 | 972.12 | 2605.11 |
| **std** | 85.35 | 354.57 | 2201.83 | 1961.60 | 52.52 | 792.41 | 400.78 | 1835.20 |
| **min** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **25%** | 333.00 | 0.00 | 4126.00 | 2527.00 | 263.00 | 0.00 | 637.00 | 1077.25 |
| **50%** | 367.00 | 509.00 | 4969.00 | 4474.00 | 300.00 | 68.00 | 906.00 | 2164.00 |
| **75%** | 433.00 | 757.00 | 6429.00 | 5838.75 | 330.00 | 616.00 | 1250.00 | 3757.00 |
| **max** | 592.00 | 999.00 | 20034.00 | 8359.00 | 449.00 | 4523.00 | 2000.00 | 9728.00 |

| | **nuclear** | **other** | **other renew** | **solar** | **waste** | **wind onshore** | **load actual** |
|---|---|---|---|---|---|---|---|
| **count** | 35047.00 | 35046.00 | 35046.00 | 35046.00 | 35045.00 | 35046.00 | 35028.00 |
| **mean** | 6263.91 | 60.23 | 85.64 | 1432.67 | 269.45 | 5464.48 | 28696.94 |
| **std** | 839.67 | 20.24 | 14.08 | 1680.12 | 50.20 | 3213.69 | 4574.99 |
| **min** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 18041.00 |
| **25%** | 5760.00 | 53.00 | 73.00 | 71.00 | 240.00 | 2933.00 | 24807.75 |
| **50%** | 6566.00 | 57.00 | 88.00 | 616.00 | 279.00 | 4849.00 | 28901.00 |
| **75%** | 7025.00 | 80.00 | 97.00 | 2578.00 | 310.00 | 7398.00 | 32192.00 |
| **max** | 7117.00 | 106.00 | 119.00 | 5792.00 | 357.00 | 17436.00 | 41015.00 |

| | **price day ahead** | **price actual** |
|---|---|---|
| **count** | 35064.00 | 35064.00 |
| **mean** | 49.87 | 57.88 |
| **std** | 14.62 | 14.20 |
| **min** | 2.06 | 9.33 |
| **25%** | 41.49 | 49.35 |
| **50%** | 50.52 | 58.02 |
| **75%** | 60.53 | 68.01 |
| **max** | 101.99 | 116.80 |

### 2.1.2 Null Values and Interpolation

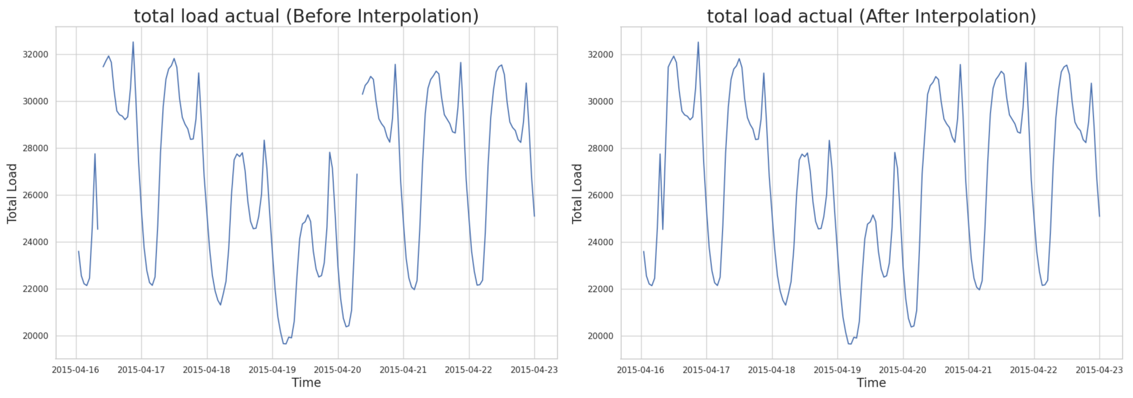
Table 2 shows the null value count for each feature. Most of the null values are in the `total load actual` feature. Luckily, `price actual` has zero null values. This feature will be used as the target for training. Examining the rows containing null values shows that the null values for energy generation mostly happen at the same time (same row). For the remaining rows, `total load actual` contains a null value.

To handle the null values, we used `pandas`' linear `interpolate` function with forward direction. This interpolation function works by estimating a straight line (linear) between the last known value and the next known value. Figure 1 shows an example of the time series of `total load actual` before and after interpolation. The gaps at the start and middle of the time series have been interpolated linearly. Looking closely, it is possible to see that the gaps have been filled with straight lines. Forward direction means that the interpolation function won't fill null values if they occur at the beginning of the time series, without preceding known values.

While this method maintains continuity and avoids data loss, it may not fully capture temporal trends inherent in time-series data. Alternative approaches, such as polynomial or spline interpolation, could be used for their ability to model continuous trends more smoothly. However, given the relatively low count of missing values shown in Table 2 compared to the dataset's `35064` rows, linear interpolation was deemed sufficient to preserve data integrity without introducing significant biases. After interpolation, the `count` of all non-zero rows per feature is `35064`.

Table 2: Number of rows with null values per feature

| **Feature** | **Null Count** |
|---|---:|
| generation biomass | 19 |
| generation fossil brown coal/lignite | 18 |
| generation fossil gas | 18 |
| generation fossil hard coal | 18 |
| generation fossil oil | 19 |
| generation hydro pumped storage consumption | 19 |
| generation hydro run-of-river and poundage | 19 |
| generation hydro water reservoir | 18 |
| generation nuclear | 17 |
| generation other | 18 |
| generation other renewable | 18 |
| generation solar | 18 |
| generation waste | 19 |
| generation wind onshore | 18 |
| total load actual | 36 |
| price day ahead | 0 |
| price actual | 0 |



(a) Before interpolation (b) After interpolation

Figure 1: A part of `total load actual`'s time series

### 2.1.3 Outliers and Duplicates

According to the summary statistics in table 1, there are no apparent outliers. `Pandas' info` function shows that all features are correctly identified as `float64`. `Pandas' duplicated` function shows that there are no duplicate rows with respect to time.

## 2.2 Weather Dataset

### 2.2.1 Null Values and Duplicates

As with the energy dataset, we need to check and handle null values and duplicates. `Pandas' isna().sum()` shows no null values in any feature. We also have to check that every city has the same number of rows. This is not the case. Valencia, Madrid, Bilbao, Barcelona, and Seville have `35145, 36267, 35951, 35476` and `35557` rows, respectively. This means there are duplicate rows since all cities have more rows than the energy dataset's `35064` rows, and cannot be merged with this dataset yet. `Pandas' drop_duplicates` function solved this and brought the number of rows per city down to `35064` after dropping duplicates with respect to time and city name.

### 2.2.2 Feature Selection and Encoding of Categorical Features

We removed an unnecessary feature, `weather_icon`. Three features describe the weather for a given hour, `weather_main, weather_description` and `weather_id`. They are all categorical, but all of them are not needed. We also need to encode the categories to numerical values. Before we remove any of the features and encode the remaining, we have to check for consistency. When counting the unique values of `weather_main, weather_description` and `weather_id`, we get `12, 41` and `37`, respectively. It's expected that the `weather_description` has a high count of unique values since it contains a greater description than the more generalized `weather_main` feature. However, it is not clear why `weather_description` and `weather_id` have different counts. Examining the unique values of `weather_description` per ID shows that the following IDs are mapped to 2 different descriptions: `501, 521, 211, 301`. Table 3a shows the IDs that were mapped to more than one weather description. This was addressed by assigning one of the descriptions of each ID with two mappings to a new, unused ID. Table 3b shows the updated IDs. It is now possible to drop the features `weather_main` and `weather_description` without substantial loss of information since all weather descriptions are mapped to a unique ID.

Table 3: Weather IDs and mapped descriptions

(a) Original IDs

| ID | Descriptions |
|---|---|
| 501 | moderate rain, proximity moderate rain |
| 521 | shower rain, proximity shower rain |
| 211 | proximity thunderstorm, thunderstorm |
| 301 | drizzle, proximity drizzle |

(b) New IDs for descriptions

| ID | Descriptions |
|---|---|
| 501 | moderate rain |
| 504 | proximity moderate rain |
| 521 | shower rain |
| 523 | proximity shower rain |
| 211 | thunderstorm |
| 212 | proximity thunderstorm |
| 301 | drizzle |
| 303 | proximity drizzle |

We are now left with a single categorical feature that has not been encoded to numerical value: `city_name`. Table 4 shows the mapping of the city names and numerical values after using `LabelEncoder` class from `sklearn.preprocessing`.

Table 4: Label Mappings for Cities

| Label | City |
|---|---|
| 0 | Barcelona |
| 1 | Bilbao |
| 2 | Madrid |
| 3 | Seville |
| 4 | Valencia |

### 2.2.3 Outliers

`Pandas' info` function shows that not all numerical features are correctly identified as `float64`, as some are identified as `int64`. We converted all the numerical features to `float64`. Table 5 shows summary statistics of the weather dataset after our preprocessing steps thus far. Note that `city_name` and `weather_id` are encoded categorical features, and the temperatures are measured in Kelvin.

The summary statistics reveal some outliers. Outliers are present in the features `pressure`, `humidity` and `wind_speed`. There are outliers in both ends of the scale for `pressure`; neither `0.00 hPa` or `1008371 hPa` is a possible air pressure. The same applies to humidity; a relative humidity of `0.00` is virtually impossible. A max `wind_speed` of `133 m/s` has not occurred in Spain. To deal with these outliers, we first plotted box plots for the aforementioned features to see the extent of outliers. Figures 2a, 2b and 2c show these.

A natural high and low cap for the `pressure` is `1051 hPa` and `950 hPa`, respectively. Those are the pressure records for Spain[5]. For `humidity`, the box plot in figure 2b shows that only values at `0.00` are outliers because of the whiskers close by. For `wind_speed`, we chose to cap at `50 m/s` since there haven't been any tornadoes with winds stronger than `50 m/s` in Spain during the period of the datasets[6].

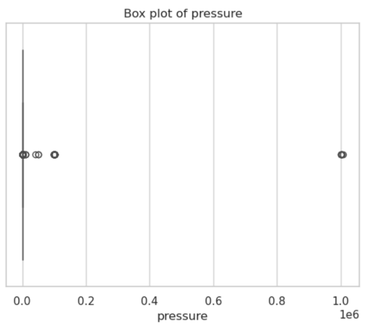
We chose to set the outliers to NaN and then interpolate the same way we did in 2.1.2. Figures 2d, 2e and 2f show box plots of the features after interpolating the outliers.

Table 5: Summary Statistics of Weather Dataset

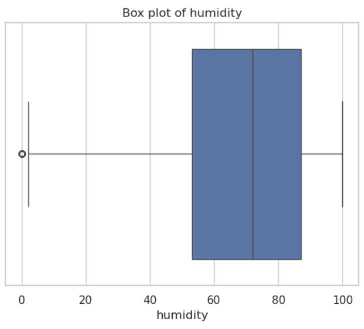
| | **city_name** | **temp** | **temp_min** | **temp_max** | **pressure** | **humidity** | **wind_speed** |
|---|---|---|---|---|---|---|---|
| **count** | 175320.00 | 175320.00 | 175320.00 | 175320.00 | 175320.00 | 175320.00 | 175320.00 |
| **mean** | 2.00 | 289.71 | 288.43 | 291.17 | 1070.20 | 68.03 | 2.47 |
| **std** | 1.41 | 8.02 | 7.95 | 8.61 | 6021.77 | 21.84 | 2.10 |
| **min** | 0.00 | 262.24 | 262.24 | 262.24 | 0.00 | 0.00 | 0.00 |
| **25%** | 1.00 | 283.83 | 282.78 | 284.91 | 1013.00 | 53.00 | 1.00 |
| **50%** | 2.00 | 289.15 | 288.15 | 290.15 | 1018.00 | 72.00 | 2.00 |
| **75%** | 3.00 | 295.24 | 294.15 | 297.15 | 1022.00 | 87.00 | 4.00 |
| **max** | 4.00 | 315.60 | 315.15 | 321.15 | 1008371.00 | 100.00 | 133.00 |

| | **wind_deg** | **rain_1h** | **rain_3h** | **snow_3h** | **clouds_all** | **weather_id** |
|---|---|---|---|---|---|---|
| **count** | 175320.00 | 175320.00 | 175320.00 | 175320.00 | 175320.00 | 175320.00 |
| **mean** | 166.72 | 0.07 | 0.00 | 0.00 | 24.34 | 763.46 |
| **std** | 116.55 | 0.39 | 0.01 | 0.22 | 30.34 | 103.10 |
| **min** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 200.00 |
| **25%** | 56.00 | 0.00 | 0.00 | 0.00 | 0.00 | 800.00 |
| **50%** | 178.00 | 0.00 | 0.00 | 0.00 | 16.00 | 800.00 |
| **75%** | 270.00 | 0.00 | 0.00 | 0.00 | 40.00 | 801.00 |
| **max** | 360.00 | 12.00 | 2.32 | 21.50 | 100.00 | 804.00 |

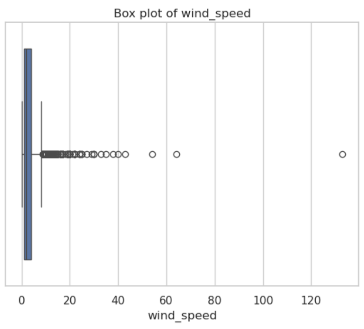
[5] *List of atmospheric pressure records in Europe* 2024.
[6] *List of European tornadoes and tornado outbreaks* 2024.



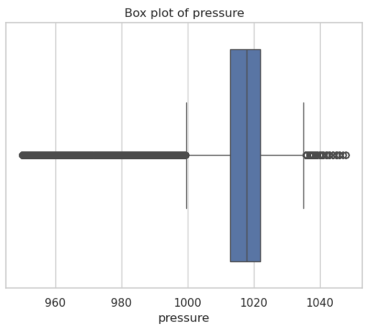
(a) Box plot of `pressure`



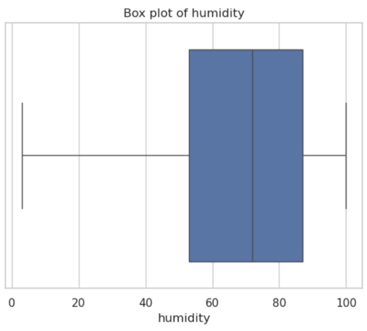
(b) Box plot of `humidity`



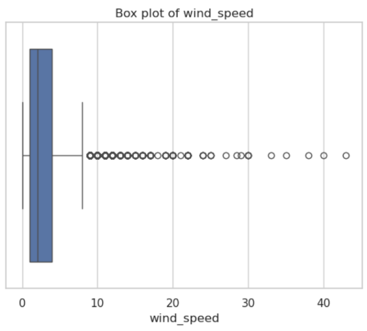
(c) Box plot of `wind_speed`



(d) After outlier handling for `pressure`



(e) After outlier handling for `humidity`

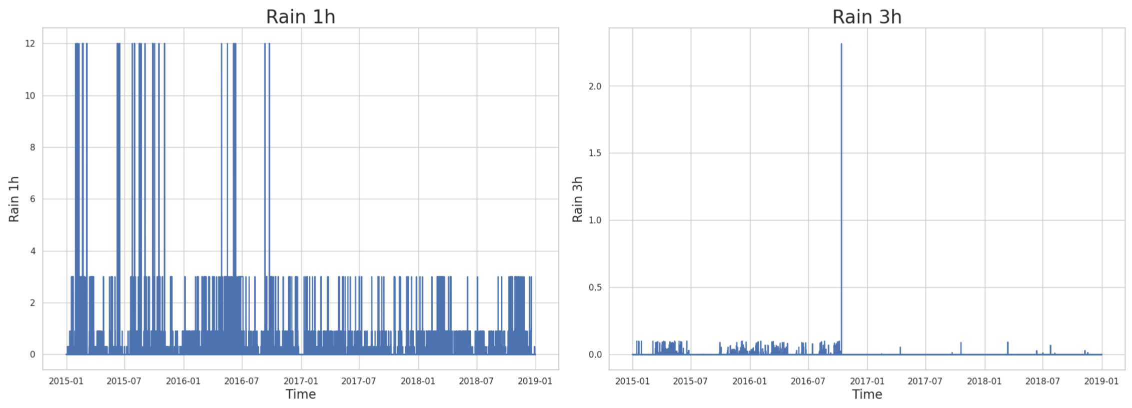


(f) After outlier handling for `wind_speed`

Figure 2: Box plots before and after outlier handling

### 2.2.4 Further Feature Selection

The summary statistics in table 5 show that there is something odd with `rain_1h` and `rain_3h`. `rain_1h` has a higher mean and max value than `rain_3h`, but that is not possible when the features are actual data, not forecasted. To examine this further, we plotted the time series of the two features, as seen in figure 3. The plots show that the `rain_3h` has strange values: a sharp peak and low values for a 3-hour amount of precipitation. We chose to drop this feature as `rain_1h` contains the necessary information.



(a) Time series of `rain_1h`          (b) Time series `rain_3h`

Figure 3: Time series of rain features

## 2.3 Combining Datasets

In order to combine the two datasets, all the features from the weather dataset for each city have to join the rows of the energy dataset. Each city gets its own features in the combined dataset for each of the features in the weather dataset. This was accomplished by splitting the weather dataset for each of the city using `pandas' groupby` function, then renaming the features to include the city's encoded numerical value, and then merging the datasets with the energy dataset. The resultant dataset has the shape `(35064, 72)`: 17 features from the energy dataset, and 5 * 11 features from the weather dataset. Double checking with `pandas' isnull` and `duplicated` functions confirm that the resultant dataset has zero null values and zero duplicate rows.

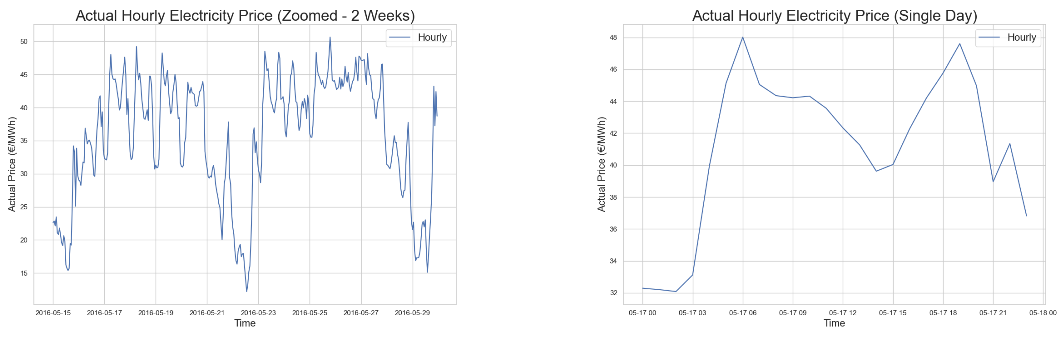
### 2.3.1 Checking Correlation

Using the `corr(method='pearson')` method, We observe some notable correlations between the target variable, energy price, and various features. For instance, total energy load and the energy generated from fossil fuel sources show a positive correlation with electricity price. On the other hand, wind speed across nearly all cities and the energy consumed through hydroelectric pumping storage are negatively correlated with electricity price. Given that the features `snow_3h_0` and `snow_3h_3` display NaNs in their correlations with the actual electricity price, we will proceed to drop these features from the analysis.

Table 6: Feature Correlations

| Feature | Correlation | Feature | Correlation |
|---|---:|---|---:|
| price actual | 1.000000 | pressure_3 | 0.090162 |
| price day ahead | 0.732155 | temp_2 | 0.087995 |
| generation fossil hard coal | 0.465637 | temp_0 | 0.085857 |
| generation fossil gas | 0.461452 | humidity_4 | 0.078819 |
| total load actual | 0.435253 | weather_id_1 | 0.077478 |
| generation fossil brown coal/lignite | 0.363993 | temp_min_3 | 0.077296 |
| generation fossil oil | 0.285050 | temp_max_1 | 0.076766 |
| generation other renewable | 0.255551 | temp_min_1 | 0.074776 |
| pressure_0 | 0.254772 | temp_1 | 0.073018 |
| pressure_1 | 0.194063 | generation hydro water reservoir | 0.071910 |
| generation waste | 0.168710 | temp_max_0 | 0.068936 |
| generation biomass | 0.142671 | temp_min_2 | 0.066777 |
| temp_min_4 | 0.133141 | weather_id_3 | 0.059085 |
| pressure_4 | 0.109812 | temp_3 | 0.050274 |
| temp_min_0 | 0.103726 | temp_max_4 | 0.047478 |
| generation other | 0.099914 | clouds_all_4 | 0.040055 |
| generation solar | 0.098529 | weather_id_0 | 0.039325 |
| temp_max_2 | 0.096279 | pressure_2 | 0.015320 |
| temp_4 | 0.090505 | weather_id_2 | 0.015275 |
| wind_speed_2 | -0.245861 | wind_deg_4 | -0.092710 |
| generation hydro pumped storage consumption | -0.426196 | wind_deg_0 | -0.096248 |
| generation nuclear | -0.053016 | wind_deg_1 | -0.103097 |
| clouds_all_1 | -0.132669 | generation hydro run-of-river and poundage | -0.136659 |
| wind_deg_3 | -0.137099 | wind_speed_0 | -0.138658 |
| wind_speed_4 | -0.142360 | wind_speed_1 | -0.143327 |
| generation wind onshore | -0.220497 | snow_3h_0 | NaN |
| snow_3h_1 | 0.014920 | snow_3h_3 | NaN |

## 2.4 Feature Extraction



(a) Electricity prices over 2 weeks (b) Electricity prices for a single day

Figure 4: Hourly electricity prices over two weeks (4a) and for a single day (4b)

Feature extraction entails constructing new features that retain key information from the original dataset in a more efficient manner, transforming raw data into numerical features that a computer program can easily understand and use [7].

In figure 4a, we display the actual hourly electricity prices from 15/05/2016 (Sunday) at 00:00 to 29/05/2016 (Sunday) at 23:00, covering two full weeks of data. In figure 4b we display the hourly electricity price for each hour from 00:00 to 23:00 at 17/05/2016 (Tuesday). Several patterns and periodicities emerge from this time series:

**Weekly Periodicity:** The electricity price generally trends higher on business days and dips over the weekends, especially on Sundays. The electricity price barely exceeds 35€/MWh on Sundays, and approaches 50€/MWh on weekdays.
**Intraday Periodicity:** Prices follow a daily cycle, with higher rates during the day and lower rates at night. This is clearly visible from figure 4b, where the electricity price reached a peak of €48/MWh at 6:00 am and 47€/MWh at 18:pm but dropped to €32/MWh at 2:00 am, highlighting the daily cycle of higher prices during the day and lower prices at night.
**Business hour variability:** From figure 4a brief drops can be spotted, possibly reflecting Spain's traditional *siesta*. From figure 4b we can see that electricity prices dipped from 44€/MWh at 9 am, to 40€/MWh at 2 pm, and then rose to 47€/MWh at 6 pm. In order to take siesta into account, which is an integral part in defining main business hours in Spain, `pandas' to_datetime()` function was used on the time-column of the preprocessed dataset. After dividing into hours, days of the week and month, values for business hours were set based on time and day of the week .

These patterns and periodicities were extracted as features `business hours` and `weekday category`. For example:

- At 10:00 am (business hour): `business_hour_category` = 2
- At 3:00 pm (siesta hour[8]): `business_hour_category` = 1
- At 11:00 pm (night hour): `business_hour_category` = 0
- For 20/05/2016 (Monday): `weekday_category` = 0
- For 25/05/2016 (Saturday): `weekday_category` = 1
- For 26/05/2016 (Sunday): `weekday_category` = 2

[7]Geeks 2024b
[8]Wikipedia contributors 2024

As the five cities in the dataset are spread across Spain, and the population vary greatly, we added weights for the different cities. This is because the weather conditions, and thus also the energy consumption in Madrid will have a greater impact on the total energy consumption than that of Valencia. This was performed by adding together the inhabitants of the five cities in the dataset, and weighting them according to percentage of total inhabitants in the five cities. This led to the following weighting: The total population is calculated as:

$$\text{population} = 5179243 + 987000 + 6155116 + 1305342 + 1645342 = 15205403$$

The weights for each city are computed as the ratio of the city's population to the total population:

$$\text{weights} = \begin{bmatrix} \frac{5179243}{15205403}, \\ \frac{987000}{15205403}, \\ \frac{6155116}{15205403}, \\ \frac{1305342}{15205403}, \\ \frac{1645342}{15205403} \end{bmatrix}$$

Calculating these values gives:

$$\text{weights} = \begin{bmatrix} 0.3407, \\ 0.0649, \\ 0.4047, \\ 0.0858, \\ 0.1083 \end{bmatrix}$$

For example, the weight for Barcelona is calculated as:

$$\text{weight}_{\text{Barcelona}} = \frac{\text{population of Barcelona}}{\text{total population}} = \frac{5179243}{15205403} \approx 0.3407$$

Each weight represents the proportion of the total population attributed to each city. The weights are applied to calculate a weighted average temperature for the cities. For instance, consider a hypothetical row from the dataset:

$$\text{temp\_Barcelona} = 30, \text{temp\_Bilbao} = 25, \text{temp\_Madrid} = 35, \text{temp\_Seville} = 40, \text{temp\_Valencia} = 28$$

Using the weights:

$$\text{temp\_weighted} = (30 \times 0.3407) + (25 \times 0.0649) + (35 \times 0.4047) + (40 \times 0.0858) + (28 \times 0.1083)$$

Calculating this gives:

$$\text{temp\_weighted} \approx 32.2$$

This weighted temperature reflects the relative influence of each city's temperature based on its population. By using weights, the model accounts for population differences, ensuring that cities with higher energy demands due to their size have a stronger influence on the analysis.

### 2.5 Data splitting

As the data frame used consisted of a time series, we split the data chronologically in order to apply different algorithms, and later on apply transfer learning. We used the first 70% of the dataset for training. The following 20% were split evenly for test and validation. The last 10% of the dataset was to be used as the *related dataset* for transfer learning (more on this later).

# 3 Algorithm Selection

## 3.1 Justifying Selection of Algorithms

Initially, having domain knowledge and an overview of the dataset we are working with is crucial when selecting an algorithm. Since the dataset consists of time-series data, two algorithms quickly become disadvantageous.

Firstly, linear and polynomial regression assume a fixed relationship between the features and target variable[9], which may be too simplistic for time-series data with non-linear, time-dependent relationships. For our dataset, electricity prices are often influenced by seasonality, trends, and non-linear factors, which are challenging to capture with a simple regression model.

Furthermore, Naïve Bayes is generally unsuitable for time-series forecasting because it assumes independence among features, contradicting the dependencies in sequential data. Power prices at one time step are likely correlated with previous time steps, so Naïve Bayes would not capture the temporal relationships effectively.

On the other hand, additive models are well-suited for time-series data with clear seasonal patterns, trends, and cycles. Prophet, for example, can capture daily, weekly, and yearly seasonality, which is often relevant in power price data. It is designed specifically for forecasting time-series and can handle irregularities and missing data gracefully.

Additionally, neural networks, particularly recurrent architectures (like LSTMs or GRUs), are designed to handle sequence data and can effectively capture temporal dependencies[10]. They are highly suitable for time-series forecasting tasks where the order and previous values are critical. Neural networks can learn complex, non-linear relationships from sequential data, making them ideal for prediction tasks.

Decision trees and random forests can be adapted for time-series forecasting by using lagged features to incorporate past values as predictors. Therefore, we elected these algorithms over regression and Naïve Bayes. However, they lack the ability to capture sequential dependencies and long-term temporal patterns, as they treat each observation independently. This limitation makes them less effective for time-series data compared to models like recurrent neural networks, which handle seasonality and trend directly.

## 3.2 Algorithm Performance Evaluation

The results from our time-series forecasting models highlight varying levels of effectiveness in predicting electricity prices. We evaluated performance based on Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and the $R^2$ Score across training, validation, and test datasets.

MAE and RMSE measure the average error magnitude, where RMSE penalizes larger errors more heavily, making it effective in assessing models with high variance in prediction error. The $R^2$ score indicates how well the model explains variance in the data, with higher values showing better model fit.

The **decision tree** model performed well with a validation MAE of 0.0322 and an RMSE of 0.0434, achieving an $R^2$ score of 0.8881 on the validation set and 0.8925 on the test set. This indicates that while the decision tree captures some of the data's underlying structure, it struggles with certain complexities in the time series, leading to lower performance compared to more advanced models.

[9]Tavasoli 2024
[10]Tavasoli 2024

The **random forest** model demonstrated strong predictive capability, outperforming the decision tree with a validation MAE of 0.0211 and an RMSE of 0.0305, alongside a high $R^2$ score of 0.9445 on the validation set. On the test set, the random forest achieved an even lower MAE of 0.0161 and an RMSE of 0.0225, with an $R^2$ score of 0.9607. This suggests that the ensemble approach effectively reduces overfitting and captures non-linear relationships within the time-series data.

**Support Vector Machine (SVM)** regression was also tested, but it did not perform as well as the tree-based models. The validation MAE of 0.0460 and RMSE of 0.0602 indicate that SVM was less effective at modeling the temporal dependencies, achieving a validation $R^2$ score of 0.7846 and a test $R^2$ score of 0.8067.

The **additive method**, such as Prophet, offers an effective baseline for capturing seasonality but has limitations in assuming linear or slightly non-linear trends, which may not fully capture the complex dynamics present in electricity prices. Although Prophet allows for both additive and multiplicative seasonalities, the model may still struggle with highly non-linear dependencies or rapid shifts in trends. In our evaluation, its performance, with an MAE of 5.6503 and an $R^2$ score of 0.7121, was not as competitive as the neural network, which is better suited to non-linear and temporal dependencies.

Finally, a **neural network** was trained with a focus on minimizing loss over several epochs, achieving a test MAE of 1.6207 and a test MSE of 5.0208. This model yielded an impressive $R^2$ score of 0.9635, showcasing its ability to capture complex, non-linear patterns over time. The neural network, specifically through its sequential layers, learns patterns across time steps, making it highly suited for capturing the sequential and seasonal patterns inherent in electricity pricing.

The random forest model performed well, with a test MAE of 0.0161 and $R^2$ of 0.9607, benefiting from ensemble averaging. However, flattening the data for random forest, decision tree, and SVM discards valuable temporal information and limits the model's ability to capture seasonality and trends. This may explain why these models yield slightly inflated metrics relative to the neural network. Neural networks, by contrast, retain sequential dependencies, achieving a test MAE of 1.6207 and $R^2$ of 0.9635, proving their suitability for complex temporal patterns

In summary, while random forests provide effective baseline predictions, the neural network's sequential handling capacity makes it preferable for time-series forecasting in electricity pricing.

Table 7: Performance Metrics of Different Algorithms

| **Algorithm** | **Dataset** | **MAE** | **MSE** | **RMSE** | $R^2$ **Score** |
|---|---|---|---|---|---|
| Decision Tree | Validation | 0.0322 | 0.0019 | 0.0434 | 0.8881 |
| Decision Tree | Test | 0.0269 | 0.0013 | 0.0372 | 0.8925 |
| Random Forest | Validation | 0.0211 | 0.0009 | 0.0305 | 0.9445 |
| Random Forest | Test | 0.0161 | 0.0005 | 0.0225 | 0.9607 |
| Support Vector Machine (SVM) | Validation | 0.0460 | 0.0036 | 0.0602 | 0.7846 |
| Support Vector Machine (SVM) | Test | 0.0390 | 0.0025 | 0.0499 | 0.8067 |
| Neural Network | Validation | X | X | X | X |
| Neural Network | Test | 1.6207 | 5.0208 | 2.2407 | 0.9635 |
| Additive Method | Validation | X | X | X | X |
| Additive Method | Test | 5.6503 | 58.0727 | 7.6205 | 0.7121 |

# 4 Ensemble Learning

## 4.1 Bagging

For the bagging implementation, we utilized `BaggingRegressor()` from `sklearn.ensemble` with `DecisionTreeRegressor()` from `sklearn.tree` as the base model. To optimize performance, we tested a range of values for critical parameters: `n_estimators`, `max_features`, and `max_samples`. Here, `n_estimators` controls the number of decision trees in the ensemble, while `max_features` and `max_samples` determine the proportion of features and samples each base estimator uses.

Bootstrapping, or sampling with replacement, allowed each base estimator to train on unique subsets of the data, potentially containing repeated samples of some instances. This approach helps reduce model variance by averaging the predictions of individual trees, each trained on a different subset.[11]

The bagging process involved looping through each combination of parameter values. Specifically, we tested the following ranges: `n_estimators` with values $[50, 100, 150]$, `max_features` with values $[0.5, 0.7, 1.0]$, and `max_samples` with values $[0.5, 0.7, 1.0]$. These ranges allowed us to explore different configurations of the ensemble.

While experimenting with even more values of the different hyperparameters might yield an even greater performance of the algorithm, this also comes at the cost of computational time. This is especially true for `n_estimators`, where larger values can result in significant computational overhead, as we experienced. Adding three different options for both the `max_samples` and `max_features` values was an easy and relatively computationally cost-effective way of improving generalization and preventing overfitting.[12]

For each combination, a new `BaggingRegressor` model was initialized using a `DecisionTreeRegressor` as the base estimator and trained on the training dataset with the `.fit()` method. Predictions were then generated on the validation dataset using the `.predict()` method, and the Mean Squared Error (MSE) was calculated to evaluate performance.

After iterating over each combination, we selected the parameters that produced the model with the lowest MSE as optimal.

To evaluate performance, we calculated the Mean Squared Error (MSE), and the $R^2$ score using, `mean_squared_error()`, and `r2_score()` functions from Scikit-Learn. The results showed that bagging effectively reduced variance and provided a more stable prediction performance, as indicated by the optimized MSE and a high $R^2$ score.

[11]Geeks 2024a.
[12]Restack 2024.

## 4.2 Boosting

For the boosting implementation, we used `xgboost` (extreme Gradient Boosting). XGBoost is an advanced implementation of the gradient boosting algorithm with several improvements over its predecessor, including efficiency and regularization, which reduce overfitting and improve overall performance.

In contrast to bagging, where models are trained independently and in parallel, boosting is a technique where models are trained sequentially, with each model learning from the errors of the previous one[13].

We implemented the XGBoost algorithm via the `xgboost` library. To ensure a robust model, we divided the dataset into training, validation, and test sets, with 25% of the training data set aside for validation. This approach allowed us to monitor whether the model was overfitting by comparing the Mean Squared Error (MSE) of the training and validation data during training.

XGBoost has several hyperparameters that can be tuned to enhance model performance. To find optimal values for the most important parameters `max_depth`, `n_estimators`, and `learning_rate` (eta), we used `GridSearchCV` from `sklearn.model_selection`. GridSearchCV is a tool in scikit-learn that automates hyperparameter tuning by exhaustively searching over a specified parameter grid, evaluating each combination to find the best configuration for model performance.[14]

For our implementation, we set `cv=3` to perform 3-fold cross-validation. This means that for each combination of hyperparameters, the data is split into three parts: the model is trained on two folds, tested on the third, and the results are averaged to evaluate the performance of that combination.

For `max_depth`, values of $[3, 5, 6]$ were tested to identify the appropriate tree depth that balances underfitting and overfitting. We were cautious not to set the `max_depth` parameter too high, as deeper trees may cause the algorithm to overfit to the training data by learning patterns specific to particular samples[15]. Hence, we included smaller values (3 and 5) alongside the default value (6) to explore a balanced configuration. Given the relatively simple nature of our model, this range was a logical choice.

For `n_estimators`, values of $[50, 70, 100, 200]$ were tested to determine the optimal number of trees without overfitting. Since `n_estimators` interacts with other parameters, such as `learning_rate`, we included both in the hyperparameter tuning process to improve performance.[16] For the `learning_rate` (`eta`), values of $[0.01, 0.1, 0.2]$ were tested to find a balance between convergence speed and model performance.

As with the bagging algorithm, we wanted to balance computational efficiency and model performance. While `GridSearchCV` automates the hyperparameter tuning process, the computational cost increases significantly as the parameter grid expands. To address this, we limited the range and number of parameters tested to maintain reasonable training times while ensuring the chosen parameter values were diverse enough to meaningfully impact the model's performance.

Other parameters, such as `min_child_weight`, `gamma`, `subsample`, and `colsample_bytree`, were left at their default values. This was done to manage the complexity and computational time of the algorithm. We also set `early_stopping_rounds` to 10, which helps prevent overfitting by stopping the training process if validation performance does not improve after 10 rounds.

Finally, the model was trained using the `fit()` method on the training data (`X_train`, `y_train`), while the validation set (`X_val`, `y_val`) was monitored for early stopping, using the hyperparameters identified by GridSearchCV as optimal.

[13]Geeks 2024a.
[14]Spagnola 2024.
[15]Banerjee 2020.
[16]XGBoosting 2024.

## 5 Transfer Learning

The transfer learning process was constrained by the lack of an external, larger dataset. We were unable to find such a dataset after doing a sufficient amount of research. We therefore went for a suboptimal solution to experiment with and demonstrate transfer learning nonetheless. Ideally, we would have utilized a dataset, such as global energy consumption data, to fully leverage the transfer learning process.

We used 90% of our composed dataset to train a model and show the transfer process on the remaining 10%. We are certain of the precautions needed to be taken for this process to make sense. We ensured temporal separation to minimize data leakage, which allowed the model to generalize patterns from the pre-training phase to unseen data during fine-tuning.

### 5.1 Model description

The model we developed for this transfer learning task is a recurrent neural network (RNN) built using TensorFlow and Keras. We used the `Sequential` API from `tensorflow.keras.models` to stack layers in a linear order, starting with an LSTM (Long Short-Term Memory) layer, implemented via `LSTM` from `tensorflow.keras.layers`. The LSTM layer has 64 units, with `relu` activation. After the LSTM layer, we included a Dense layer with 32 units and `relu` activation, which helps further transform the output from the LSTM. The final layer is a Dense layer with a single unit, providing the output for the regression.[17]

To compile the model, we used the Adam optimizer from `tensorflow.keras.optimizers` with a learning rate of 0.001. The loss function selected is mean squared error (MSE). Training was performed with `model.fit()` on the main dataset ($X_{\text{transfer}}$ and $y_{\text{transfer}}$), which represents 90% of our data. This split allowed us to train, test and tune the model on data from January 1st 2015 to August 7th 2018, and use this to predict the energy consumption for the remaining four months of 2018. An important distinction when applying the trained model to predict the price of energy for the last four month is that the weather conditions are viewed as forecasts. To prevent overfitting and avoid excessive epochs, we incorporated an early stopping mechanism using `EarlyStopping` from `tensorflow.keras.callbacks`. We set `monitor='val_loss'`, which tracks validation loss, and `patience=5` to stop training if the validation loss doesn't improve over five consecutive epochs.

After training, we applied transfer learning by freezing the LSTM layer weights to retain the patterns learned on the main dataset.[18] We set the `.trainable` attribute to `False` for all layers except the last Dense layer, allowing only the last layer to be retrained on new data. We recompiled the model with a learning rate of 0.0001 for fine-tuning. The remaining 10% of the data was then split into two sets: one for fine-tuning and one for final testing. Fine-tuning was performed on the first half of this subset ($X_{\text{finetune}}, y_{\text{finetune}}$) using `model.fit()`. Once trained, we used `model.predict()` to generate predictions ($y_{\text{pred}}$) on the test set ($X_{\text{test\_final}}$).

[17] Warya 2024.
[18] Han 2024.

## 5.2 Advanced Algorithm Performance Evaluation

The Bagging model, implemented with a `DecisionTreeRegressor` as the base algorithm, was configured with 100 estimators, each using 90% of the features and samples. On the validation set, this model achieved an $R^2$ score of 0.9332, an RMSE of 0.0288, an MAE of 0.0203, and an MSE of 0.0008. The test set results were slightly better, with an $R^2$ score of 0.9070, an RMSE of 0.0230, an MAE of 0.0174, and an MSE of 0.0005. These metrics indicate that the model captures a substantial portion of the variance in the data, with the $R^2$ score above 0.90 in both validation and test sets. However, the RMSE values indicate that the model may not be as precise as the other algorithms in predicting specific electricity prices, as the error is still relatively high. Bagging performed well overall, yet its limitation lies in its inability to capture sequential dependencies in time-series data[19], which might explain why its $R^2$ and RMSE values are lower than the other models.

Table 8 shows the validation and test performance metrics for all three algorithms.

The XGBoost model, configured with a grid search for parameters, resulted in an optimal setup of 500 estimators, a learning rate of 0.1, and a maximum depth of 3. Early stopping was used to avoid overfitting[20], and the model reached the best performance at iteration 345. Figure 5 shows the training process and early stopping. On the validation set, XGBoost achieved an $R^2$ score of 0.9485, an RMSE of 0.0253, an MAE of 0.0180, and an MSE of 0.0006. On the test set, the model's performance was strong, with an $R^2$ score of 0.9261, an RMSE of 0.0205, an MAE of 0.0153, and an MSE of 0.0004. XGBoost performed very well, achieving the highest $R^2$ scores and lowest RMSE values across both validation and test sets, showing a strong fit and high precision. The boost in accuracy is due to XGBoost's iterative improvement, as each estimator corrects errors from the previous one[21]. The use of early stopping was beneficial, as it avoided overfitting and maintained generalizability. It is also worth mentioning that it performed exceptionally well compared to other models in efficiency.

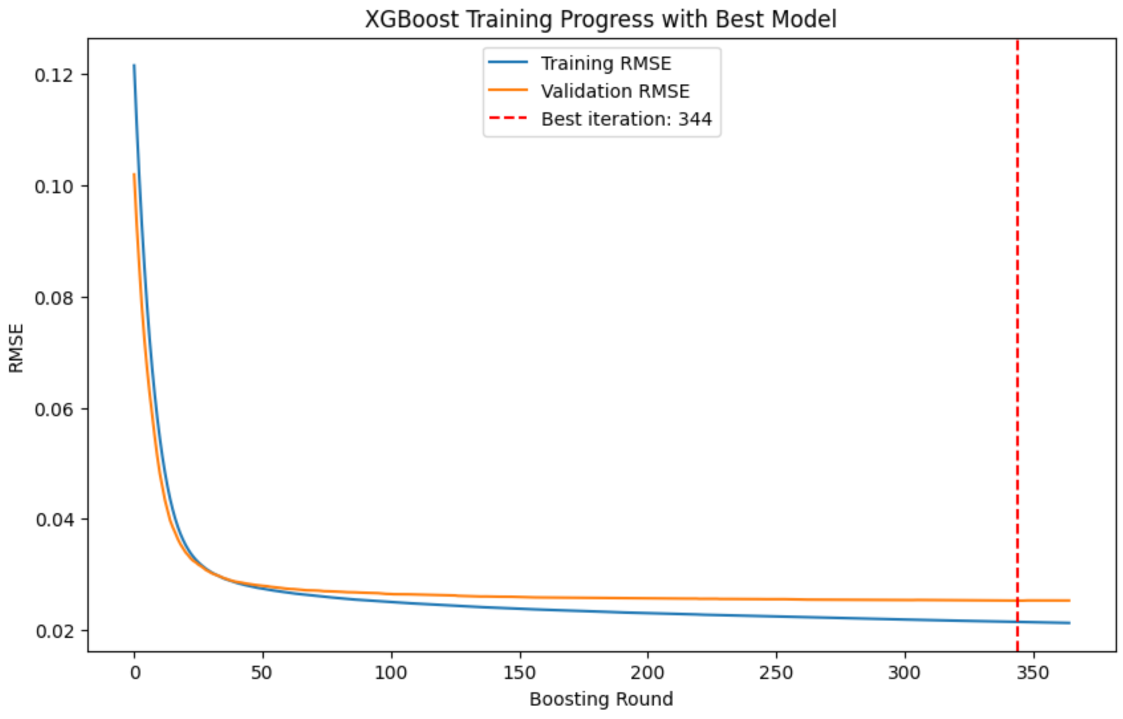


Figure 5: XGBoost Training Process

[19]Acharya 2024.
[20]Spagnola 2024.
[21]Acharya 2024.

The LSTM model with transfer learning was structured in two parts: an initial training on 90% of the dataset followed by fine-tuning on the remaining 10% of the data. For the test set, the transfer learning model achieved an $R^2$ score of 0.9229, an RMSE of 0.0223, an MAE of 0.0161, and an MSE of 0.0005. The $R^2$ score above 0.92 on the test sets indicates that the LSTM model was able to effectively capture the temporal dependencies in the time-series data[22]. However, the same dataset was used for both pretraining and transfer learning, which may have slightly boosted the performance metrics. Despite this, the model's RMSE and MAE values are comparable to XGBoost, highlighting its capability to model sequential patterns well. However, the risk of overlapping information between training and fine-tuning stages might suggests that the results should be interpreted with caution.

Compared to the basic algorithms, the advanced models—Bagging with `DecisionTreeRegressor`, Boosting with XGBoost, and LSTM with transfer learning, demonstrated an advantage in capturing the complex temporal dependencies needed for effective electricity price forecasting[23]. The Random Forest model showed strong performance with a test $R^2$ of 0.9296, benefiting from ensemble averaging to handle non-linear relationships. However, flattening the data limited its ability to model sequential dependencies, a critical aspect in time-series data. XGBoost improved on this with a test $R^2$ of 0.9261, using iterative boosting to refine predictions and minimize errors, capturing complex patterns more effectively. The LSTM model with transfer learning also improved, achieving a test $R^2$ of 0.9229, thanks to its architecture, which is specifically designed for sequential data, allowing it to model intricate time dependencies[24]. In contrast, simpler models like the Decision Tree ($R^2$ of 0.7916) and SVM ($R^2$ of 0.6615) seemed to struggle to capture temporal patterns, while the additive model, with an $R^2$ of 0.7121, failed to account for the non-linear trends present in electricity prices. Overall, while the Random Forest model provided a strong baseline, advanced algorithms like XGBoost and LSTM, with their handling of sequential dependencies, proved to be better choices for accurate and robust time-series forecasting[25].

| **Model** | **Dataset** | **MSE** | **RMSE** | $\mathbf{R^2}$ | **MAE** |
|---|---|---|---|---|---|
| Bagging (DecisionTreeRegressor) | Validation | 0.0008 | 0.0288 | 0.9332 | 0.0203 |
| | Test | 0.0005 | 0.0230 | 0.9070 | 0.0174 |
| XGBoost | Validation | 0.0006 | 0.0253 | 0.9485 | 0.0180 |
| | Test | 0.0004 | 0.0205 | 0.9261 | 0.0153 |
| LSTM (Transfer Learning) | Validation | x | x | x | x |
| | Test | 0.0005 | 0.0223 | 0.9229 | 0.0161 |

Table 8: Performance Metrics for Advanced Algorithms

[22]Warya 2024.
[23]Encord 2024.
[24]Encord 2024.
[25]Acharya 2024.

## References

Acharya, Nirajan (2024). *Comparative Analysis of Bagging and Boosting in Ensemble Learning.* Accessed: November 12, 2024. URL: https://medium.com/@nirajan.acharya777/comparative-analysis-of-bagging-and-boosting-in-ensemble-learning-4fc9f2aac91c.

Banerjee, Prashant (2020). *A Guide on XGBoost hyperparameters tuning.* Accessed: December 3, 2024. URL: https://www.kaggle.com/code/prashant111/a-guide-on-xgboost-hyperparameters-tuning.

Encord (2024). *Time Series Predictions with RNNs.* Accessed: November 12, 2024. URL: https://encord.com/blog/time-series-predictions-with-recurrent-neural-networks/.

Geeks, Geeks for (2024a). *Bagging vs Boosting in Machine Learning.* Accessed: November 11, 2024. URL: https://www.geeksforgeeks.org/bagging-vs-boosting-in-machine-learning/.

— (2024b). *What is Feature Extraction?* Accessed: November 7, 2024. URL: https://www.geeksforgeeks.org/what-is-feature-extraction/.

Han, Vikash Chou (2024). *What is Transfer Learning?* Accessed: November 12, 2024. URL: https://www.geeksforgeeks.org/ml-introduction-to-transfer-learning/.

*List of atmospheric pressure records in Europe* (2024). URL: https://en.wikipedia.org/wiki/List_of_atmospheric_pressure_records_in_Europe (visited on 4th Nov. 2024).

*List of European tornadoes and tornado outbreaks* (2024). URL: https://en.wikipedia.org/wiki/List_of_European_tornadoes_and_tornado_outbreaks (visited on 4th Nov. 2024).

Oğuz, Selin (2023). *Ranked: The Cheapest Sources of Electricity in the U.S.* URL: https://decarbonization.visualcapitalist.com/the-cheapest-sources-of-electricity-in-the-us/ (visited on 1st Nov. 2024).

Restack (2024). *Hyperparameter Tuning for Bagging.* Accessed: December 3, 2024. URL: https://www.restack.io/p/hyperparameter-tuning-answer-bagging-cat-ai.

Spagnola, Jeff (2024). *The Basics of Gridsearch.* Accessed: November 11, 2024. URL: https://jeffspagnola.medium.com/the-basics-of-gridsearch-e9cc9da7578f.

*Strømprisene* (2024). URL: https://www.aftenposten.no/emne/stroemprisene (visited on 1st Nov. 2024).

Tavasoli, Simon (2024). *10 Types of Machine Learning Algorithms and Models.* Accessed: November 11, 2024. URL: https://www.simplilearn.com/10-algorithms-machine-learning-engineers-need-to-know-article.

*The Power Market* (2024). URL: https://www.nordpoolgroup.com/en/the-power-market/ (visited on 1st Nov. 2024).

Warya, Aish (2024). *Introduction to Recurrent Neural Network.* Accessed: November 11, 2024. URL: https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/.

Wikipedia contributors (2024). *Siesta.* Accessed: November 6, 2024. URL: https://en.wikipedia.org/wiki/Siesta.

XGBoosting (2024). *Configure XGBoost "$n_e$stimators" Parameter.* Accessed: December 3, 2024. URL: https://xgboosting.com/configure-xgboost-n_estimators-parameter/.