

## 1. Introduction

This paper presents a detailed analysis of advanced machine learning techniques and neural network models in the context of forecasting hourly electricity prices. The study utilizes data obtained from the EXIST Market Transparency Platform, and encompasses various steps such as data pre-processing, manipulation, normalization, and pipelining for model hyperparameter tuning.

To establish a solid foundation for comparison, an initial XGBM (Extreme Gradient-Boosting Machine) model is developed as a baseline model. The primary evaluation metric used for benchmarking purposes is the root mean squared error (RMSE), where lower values indicate better performance.

Variable	Description
PTF	Piyasa takas fiyatı
Volume	İşlem Hacmi
bid_amount	Teklif edilen alış miktarları
ask_amount	Teklif edilen satış miktarları
imbalance_delta	Pozitif ve negative dengesizlik miktarlarının deltası

## 2. Data Pre-processing

The input data undergoes a series of cleaning methodologies to ensure its integrity and usability. Firstly, the original dataset is subjected to separation of date and hour data, followed by the application of datetime formatting in the form of %d%m/%Y+%H:+%M (day, month, year: hour, minute). Subsequently, the concatenated datetime object is indexed for efficient handling.

To streamline the dataset, unnecessary columns such as "Tarih" and "Saat" are removed, as they are no longer required due to the desired datetime format being in place. Furthermore, certain variables undergo renaming to enhance clarity and consistency, with examples including the transformation of "Pozitif Dengesizlik Miktarı (MWh)" to "positive\_imbalance" and "PTF (TL/MWh)" to "PTF."

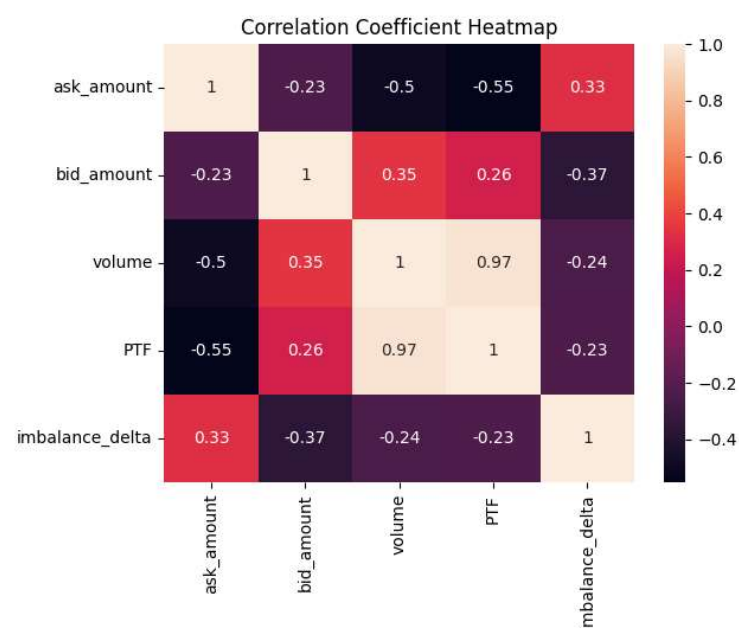
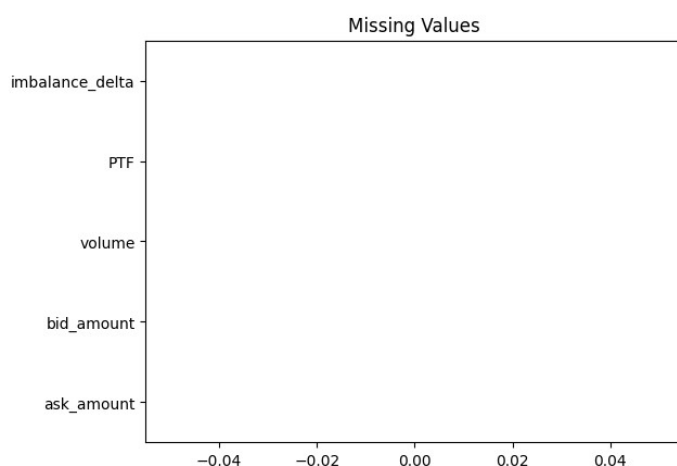
The data is then subjected to regular expression usage, effectively addressing and rectifying any instances of incorrect numeric formatting. As a subsequent step, all numeric variables are converted to float type to ensure uniformity and compatibility.

The introduction of the "imbalance\_delta" variable becomes pivotal, derived from the addition of "positive\_imbalance" and "negative\_imbalance." Finally, all the cleaned and processed data is consolidated into a single entity, facilitating comprehensive analysis and subsequent modeling tasks.<sup>1</sup>

Finally, exogenous variables (features) are shifted by one to create lag-1 values to be used when making predictions at time t-1 to forecast “PFT” at time t.

	ask_amount	bid_amount	volume	PTF	imbalance_delta
46699	20206.5	23439.0	45294546.43	2600.00	-762.58
46700	19436.6	24226.4	43636645.56	2600.00	-2236.11
46701	18790.2	25349.4	42515958.26	2600.00	-4256.34
46702	18729.1	24971.0	42581597.45	2600.00	-4574.26
46703	19674.5	23071.1	45500781.80	2348.33	-2537.56

## 2.1 Data Integrity



<sup>1</sup> [https://raw.githubusercontent.com/dfavenfre/electricity-price-forecasting/main/useful\\_functions.py](https://raw.githubusercontent.com/dfavenfre/electricity-price-forecasting/main/useful_functions.py)

### 3. Machine Learning Approach

A baseline XGBM regression model is built for later root mean squared error (RMSE) performance benchmark. The main strategy followed when building XGBM model is as follows.

$$y_t = a_1 + w_{11} * y_{1(t-1)} + w_{12} * y_{2(t-1)} + w_{13} * y_{3(t-1)} + w_{14} * y_{4(t-1)} + e_{2(t-1)}$$

Where;

\*  $a_1$ : *Constant term*

\*  $w_{11}, w_{12}, w_{13}, w_{14}$ : *Coefficients*

\*  $e_1$ : *Error term*

```
# XGBM Baseline Model Performance
xgbm = XGBRegressor(random_state=42)
xgbm.fit(X_train, y_train)
xgbm_pred = xgbm.predict(X_test)
xgbm_rmse = (MSE(y_test, xgbm_pred))**(1/2)
print("XGBM RMSE: {:.2f}".format(xgbm_rmse), "\nr^2:
{:.3f}".format(xgbm.score(X_test, y_test)))
```

Output:

```
XGBM RMSE: 61.42
r^2: 0.965
```

The baseline XGBM model achieved an RMSE score of 61.42 and an r-square value of 96.5. In order to improve upon these results, a hyperparameter tuning process was employed within a pipeline, incorporating the use of RobustScaler with the `with_scaling` parameter set to True. This scaler was chosen to address outliers present within the inter-quartile range (IQR) and ensure the robustness of the model.

The hyperparameter tuning of the XGBM model yielded superior results, specifically a reduction in RMSE. The inclusion of RobustScaler becomes particularly important when working with tree-based algorithms like XGBM, given their sensitivity to unsymmetrical scales and outlier variables. Nonetheless, it should be noted that while applying a scaler may effectively address outlier or outscaling issues, it is not a definitive solution for achieving optimal results.

```

# pipeline for DecisionTreeRegressor()
xgbm_pipeline = Pipeline([("RS", RobustScaler(with_scaling=True)),
                           ('XGBM', XGBRegressor(random_state=42))])

# XGBM Hyperparameters
xgbm_parameters = [{"XGBM__max_depth": np.arange(3, 30, 3),
                    "XGBM__n_estimators": np.arange(100, 351, 50),
                    "XGBM__learning_rate": np.arange(0.1, 0.45, 0.05),
                    "XGBM__subsample": np.arange(0.7, 0.95, 0.05)}]

# RandomizedSearchCV w/ pipeline
xgbm_rsg = RandomizedSearchCV(estimator=xgbm_pipeline,
                              param_distributions=xgbm_parameters,
                              scoring="neg_mean_squared_error",
                              verbose=1)

# fit and predict
xgbm_rsg.fit(X_train, y_train)
xgbm_rsg_preds = xgbm_rsg.predict(X_test)

# RMSE Score
xgbm_rsg_rmse = (MSE(y_test, xgbm_rsg_preds))**(1/2)

# Model Performance and Best Parameter Selection
print("XGBRegressor RMSE: {:.3f}".format(xgbm_rsg_rmse),
      "\nBest Parameters: {}".format(xgbm_rsg.best_params_))

```

Output:

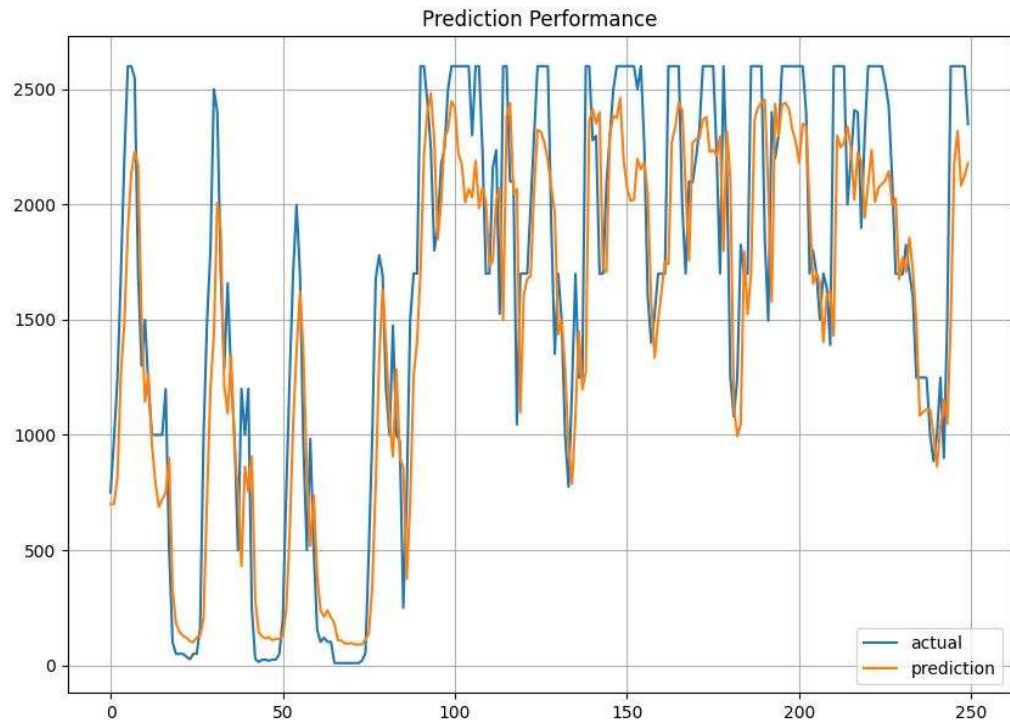
```

XGBRegressor RMSE: 59.972
Best Parameters: {'XGBM__subsample': 0.8, 'XGBM__n_estimators': 150,
'XGBM__max_depth': 3, 'XGBM__learning_rate': 0.1}

```

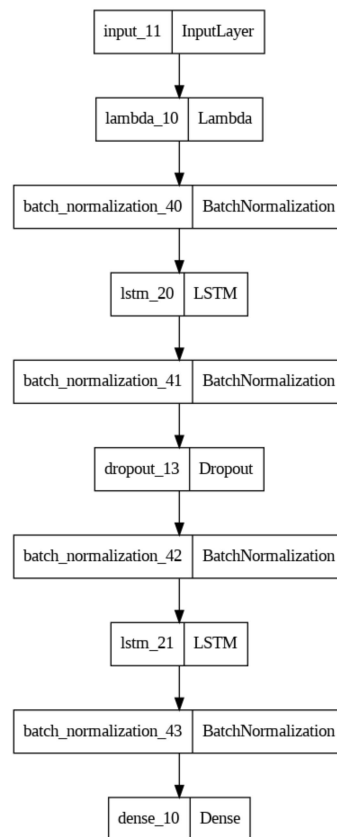
Hyperparameter tuning is conducted a couple of times, with wider range yet narrow intervals and the final result, or the best model, yielded the following hyperparameters as the best estimators for achieving less RMSE.

- subsample: 0.8
- n\_estimators: 150
- max\_depth: 3
- learning\_rate: 0.1



#### 4. Deep Neural Network Model

##### Model Architecture



The model architecture is built upon various layers including LSTM, Dense, BatchNormalization, and Dropout.

LSTMs are a type of recurrent neural network (RNN) that are designed to effectively capture long-term dependencies in sequential data. They are particularly useful when dealing with sequences that have long-range dependencies, where information from distant past or future time steps can significantly impact the current prediction. LSTMs achieve this by using a memory cell and a set of gates that control the flow of information within the network.

#### **4.1 Additional Layers**

##### BatchNormalization

Batch Norm is a normalization technique done between the layers of a Neural Network instead of in the raw data. It is done along mini-batches instead of the full data set. It serves to speed up training and use higher learning rates, making learning easier.

$$z_N = \frac{z - mz}{sz}$$

##### Dropout

A regularization method approximating concurrent training of many neural networks with various designs. During training, some layer outputs are ignored or dropped at random. This makes the layer appear and is regarded as having a different number of nodes and connectedness to the preceding layer. Adding a Dropout layer also helps overfitting by eliminating some nodes with vanishing gradient descent (dead-neuron) issue and not transferring empty nodes for later backward propagation weight optimizations.

##### Dense layer


Also known as a fully connected layer, is commonly used after the LSTM layer. The Dense layer is responsible for transforming the output of the LSTM layer into a suitable format for the specific task at hand. It performs a linear transformation followed by a non-linear activation function on the input data, allowing the model to learn complex patterns and make predictions based on the transformed representation.

## 4.2 Data Preparation

Predecessor 48-hour period will be used as look-back horizon to forecast the “PTF” (target variable) at time  $t+1$  horizon. To achieve that, the target data is shifted by 48 periods to create look-back horizon data.

	ask_amount	bid_amount	volume	imbalance_delta	PTF+1	PTF+2	PTF+3	PTF+4	PTF+5	PTF+6	...
49	30969.099609	20929.000000	2623939.00	960.739990	195.009995	169.910004	174.119995	182.990005	141.000000	182.050003	...
50	32707.599609	20487.300781	2410817.00	633.530029	179.130005	195.009995	169.910004	174.119995	182.990005	141.000000	...
51	33463.101562	20161.900391	2042677.00	687.820007	141.660004	179.130005	195.009995	169.910004	174.119995	182.990005	...
52	33910.601562	19745.900391	1856058.00	800.080017	130.000000	141.660004	179.130005	195.009995	169.910004	174.119995	...
53	34095.398438	19687.500000	1851811.75	889.780029	129.990005	130.000000	141.660004	179.130005	195.009995	169.910004	...

5 rows × 52 columns



Then, the model architecture is initiated with 3,260,099 trainable and 3,176 non-trainable parameters. The non-trainable parameters appear due to BatchNormalization layer. Vanished (dead-neurons) are not used and thus, not trained, during back-propagation weight optimization process.

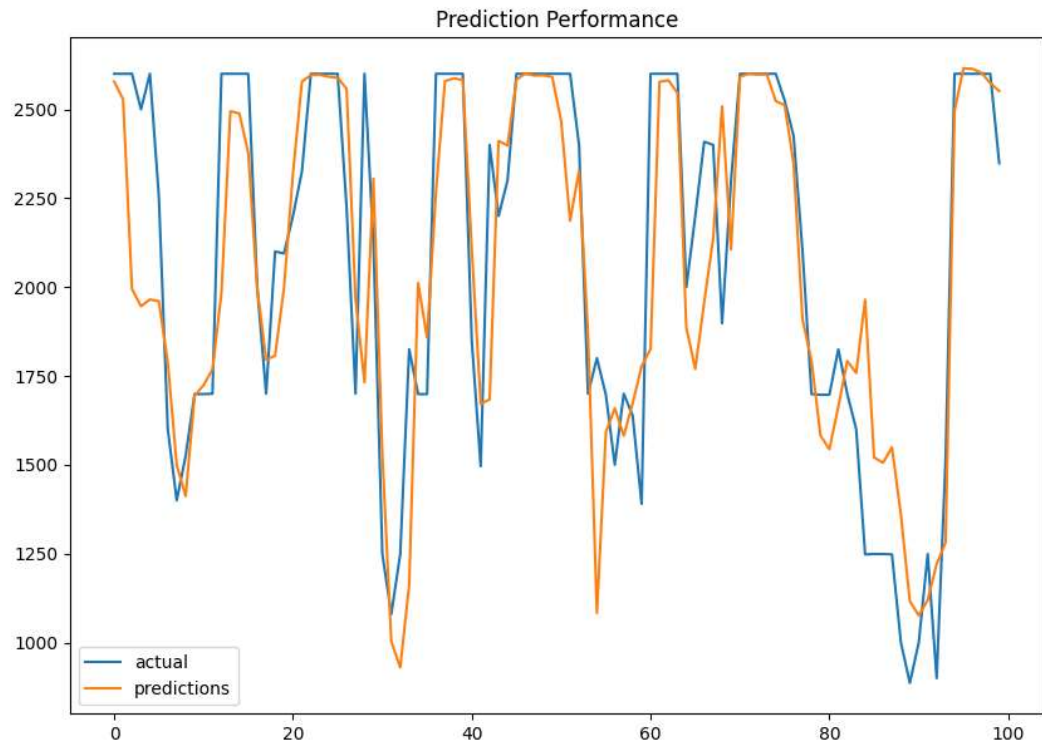
## 4.3 Model Summary

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	[(None, 52)]	0
lambda_10 (Lambda)	(None, 1, 52)	0
batch_normalization_40 (Batch Normalization)	(None, 1, 52)	208
lstm_20 (LSTM)	(None, 1, 512)	1157120
batch_normalization_41 (Batch Normalization)	(None, 1, 512)	2048
dropout_13 (Dropout)	(None, 1, 512)	0
batch_normalization_42 (Batch Normalization)	(None, 1, 512)	2048
lstm_21 (LSTM)	(None, 512)	2099200
batch_normalization_43 (Batch Normalization)	(None, 512)	2048
dense_10 (Dense)	(None, 1)	513
Total params: 3,263,185		
Trainable params: 3,260,009		
Non-trainable params: 3,176		

The fine-tuning process involved various hyperparameters, such as `batch_size`, number of epochs, number of layers, number of dropout layers, rate of dropouts, rate of batch normalization and so on.

To conclude, the final model achieved an RMSE of 57.348, which is lower than what fine-tuned XGBM model obtained.

#### 4.4 Model Performance



#### 5. Conclusion

Fine-tuned XGBM model appear to reveal less accurate results at capturing time-variant dependencies with lag-1 exogenous variables than nn.LSTM. Further improvements may include but not be limited to.

- Continuing hyperparameter tuning with wider ranges yet miniscule intervals.
- Adding additional layers, such as Dropout, LSTM, or Dense.
- More data may put forth additional robustness for later models.
- Better feature variables would add further model performance.



## 6. References

- <https://otexts.com/fpp2/tspatterns.html>
- <https://seffaflik.epias.com.tr/transparency/piyasalar/dengesizlik/dengesizlik-miktari.xhtml>
- <https://seffaflik.epias.com.tr/transparency/piyasalar/gop/islem-hacmi.xhtml>
- <https://seffaflik.epias.com.tr/transparency/piyasalar/gop/ptf.xhtml>
- <https://seffaflik.epias.com.tr/transparency/piyasalar/gop/teklif-edilen-satis-miktarlari.xhtml>
- <https://seffaflik.epias.com.tr/transparency/piyasalar/gop/teklif-edilen-alis-miktarlari.xhtml>
- <https://github.com/dfavenfre/electricity-price-forecasting/tree/main>
- <https://www.analyticsvidhya.com/blog/2018/09/multivariate-time-series-guide-forecasting-modeling-python-codes/>
- [https://www.tensorflow.org/api\\_docs/python/tf/keras/layers/BatchNormalization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization)
- <https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1>