Machine Learning Project Documentation

Model Refinement

1. Overview

Our project consists in creating an **AI** model for forecasting future energy Consumption for Household and Businesses. In that process model refining plays an important role. It is a crucial step for improving the performance of the machine learning model. This phase involves fine-tuning the model, exploring different algorithms, and ensuring that the model generalizes well to unseen data. his goal is to enhance the model's accuracy and robustness, making it ready for deployment.

For our purpose three models have been refined notably: **ARIMA, XGBoost and LSTM models**. The best model obtained is the **LSTM** with a score of **95.3**.

2. Model Evaluation

The problem we are trying to solve is **a time series problem**. The user energy consumption is collected over the time. Models adapted to such problem are models like: **ARIMA**, **LSTM** or Auto Regression models. Then for evaluation, we can use common regression evaluation metrics including **Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), R-squared (Coefficient of Determination), and Mean Absolute Percentage Error (MAPE)** [1].

In our case, we used **Mean Absolute Error (MAE**) because of its simplicity. Indeed, it calculates the absolute difference between actual and predicted values. More its closes to **0** more the evaluated model is doing a good job. Between two models the best model is the one which has a smallest **MAE**.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$$

MAE equation

In addition to the **MAE**, we had also used graphics like **Normal Q-Q**, **residual distribution plot, model loss graphic and prediction visualization** to check if the models are generalizing well.

3. Refinement Techniques

To refine the model, we explored several techniques:

- **Hyperparameter Tuning:**
    - We performed a **grid search cv** to find the optimal hyperparameters for the **XGBoost model**. He key hyperparameters tuned included **n_estimators, max_depth, learning_rate, and subsample**. The optimal parameters were found to be:

Parameters	Values
n_estimators	***50, 100, 200***
max_depth	***3, 5, 7***
learning_rate	***0.01, 0.1, 0.3***
subsample	***0.6, 0.8, 1.0***
cv	***2,3,4,5***

These adjustments led to a significant reduction in **MAE.**

- For **ARIMA** model we performed hyperparameter tuning by training and evaluating the model with the parameters **(p,0, q) in a for loop**.

  The couple **(p,q)** checked can be find the table below:

	0	**8**	**16**	**24**
0	**(0,0,0)**	**(8,0,0)**	**(16,0,0)**	**(24,0,0)**
1	**(0,0,1)**	**(8,0,1)**	**(16,0,1)**	**(24,0,1)**
2	**(0,0,2)**	**(8,0,2)**	**(16,0,2)**	**(24,0,2)**

In order to figure out the optimum value for **q and p** we had checked **the autocorrelation (ACF) plot and the partial autocorrelation (PACF) plot**.

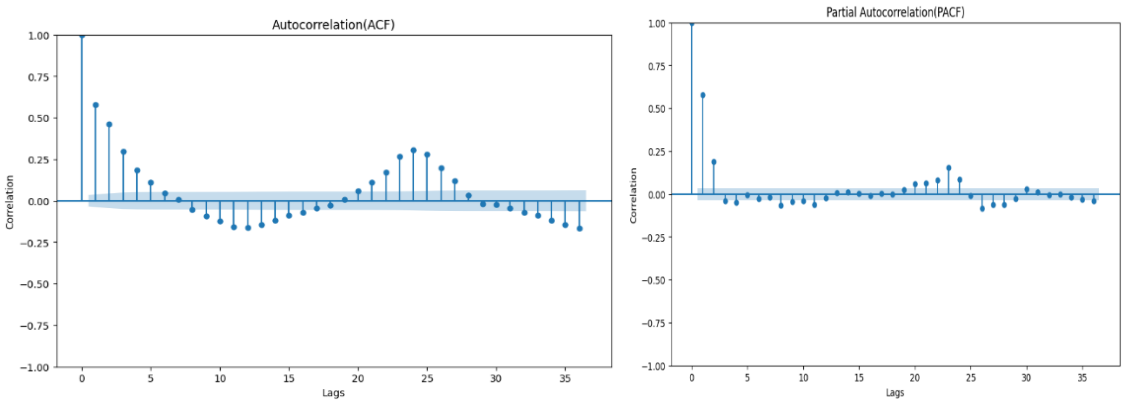


Figure 2: Autocorrelation and partial autocorrelation plot

Through the partial autocorrelation plot we can see that there a last short correlation at lag 28.

- For the **LSTM** model, we tried to fine tune parameters like: **layers, activation functions, the learning rate and the optimizer function** used to compile the model.

  The summary of the architecture shows below:

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 24, 50)	10,400
lstm_1 (LSTM)	(None, 25)	7,600
dense (Dense)	(None, 1)	26

Total params: 18,026 (70.41 KB)
Trainable params: 18,026 (70.41 KB)
Non-trainable params: 0 (0.00 B)

We can notice that the model is not too deep and the required input data shape is **24**. This number has been chosen based on the best **ARIMA** model obtained, **24** corresponds to the **partial autocorrelation lag** of the best **ARIMA** model obtained.

We have also used **model loss visualization** in order to view if the model is **generalizing well**.

The result is below:

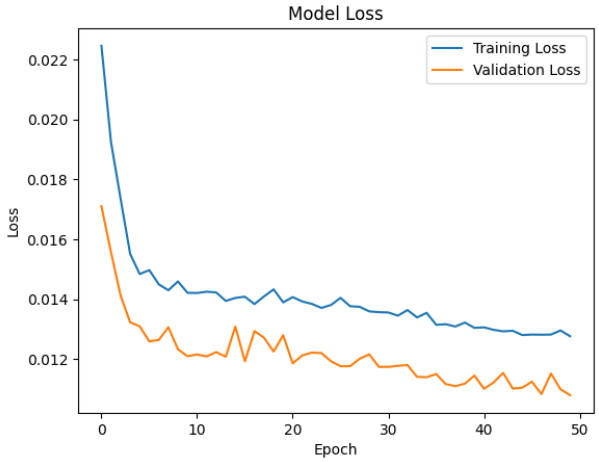


Figure 3: model loss graphic

- **Algorithm Selection:** we compared the performance of **ARIMA, LSTM, and XGBoost models** and selected **LSTM** model for its balance between performance and interpretability. We used **a bar chat** to visualize easily the comparison.

4. Hyperparameter Tuning

In addition to what have been done for parameters tuning in the previous section, we also played **with normalization technics**. We have tried **MinMaxScaler and StandardScaler**. What, we observed is that the model performing was changing with the method chosen. **MinMaxScaler** had been the one with better performance for all the model. So, we choose it.

5. Cross-Validation

We maintained the same cross-validation strategy of **5-fold** cross-validation but ensured that the time series data was split appropriately to avoid **data leakage**. This was crucial for models like **ARIMA and LSTM**, which are sensitive to the order of data. What we observed give cross validation is that the model was taking **too much time to train** when we increased the value of **k**, and the performance of models was increasing also when we increased that the value.

6. Feature Selection

Feature selection **was not explicitly performed in this phase**, as the dataset was already reduced to **a single feature** of the energy consumed by household over the time ("Appliances"). However, we considered incorporating additional features from the original dataset to improve model performance.

Test Submission

1. Overview

The test submission phase involved preparing the final model for evaluation on the test dataset. This phase ensured that the model was ready for deployment and could handle new, unseen data effectively. During our work, all the models went through this step. The model which had obtained the highest score during this phase had been selected as the one to be deploy, notably our **LSTM model**.

2. Data Preparation for Testing

The test dataset was prepared by resampling the data to match the training dataset's frequency (**hourly**). For the reason that we faced a time series problem, we used **horizontal splitting** to split our initial dataset into training and testing dataset. We took **80 %** of the data for the training set and **20 %** testing set. The code is below:

```
# dataset splitting
cut_off = int(len(df) * 0.8) # cutting point
train = df.iloc[:cut_off] # horizontal splitting
test = df.iloc[cut_off:]

print('train shape: ', train.shape)
print('test shape: ', test.shape)
```

Later, as soon as the previous operation had been successfully realized, during the models building step, we also **normalized** the testing dataset after **reshaping** it into **2 dim arrays**. Here is the code of that operation for both training and testing dataset:

```
# Reshape data for scaling
train_reshaped = train.values.reshape(-1, 1)
test_reshaped  = test.values.reshape(-1, 1)

# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))# StandardScaler() used also for tuning
train_scaled = scaler.fit_transform(train_reshaped)
test_scaled = scaler.transform(test_reshaped)
```

Throughout the work, the same preprocessing steps, **including scaling and any feature engineering**, were applied to the test data.

3. Model Application

For ARIMA model we used this snippet of code:

```
# Evalute the Test score
y_pred_test = model.predict(start = test.index[0], end = test.index[-1])
test_mae.append(round(mean_absolute_error(test.values, y_pred_test), 2))
print("Test MAE  ARMA(24,0,1):", test_mae[-1])
```

For LSTM model we used this snippet of code:

```
    test_pred = lstm_model.predict(X_test) # model predict dataset
    # Print scores
    print("LSTM Train MAE:", round(mean_absolute_error(y_train, train_pred), 2))
    print("LSTM Test MAE:", round(mean_absolute_error(y_test, test_pred), 2))
    # Append into test_mae
    test_mae.append(round(mean_absolute_error(y_test, test_pred), 2))
```

For XGBoost model we used this snippet of code:

```
    test_pred = best_model.predict(X_test) # Test data prediction
    # Print scores
    print("Train MAE:", round(mean_absolute_error(y_train, train_pred), 2))
    print("Test MAE:", round(mean_absolute_error(y_test, test_pred), 2))
```

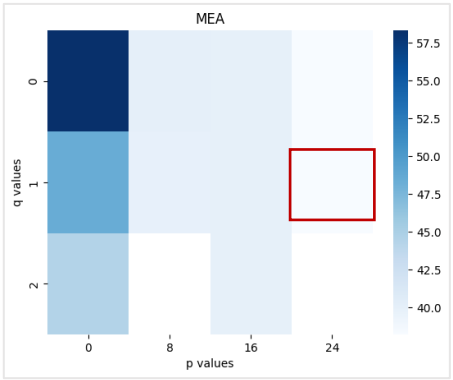
As soon as the models have been trained and evaluated, after features engineering and fine-tuning operations, we just used the **predict function** to obtain the prediction of our unknow test data.

4. Test Metrics

The metrics used for testing are the same with those which had been used during the training step: the **MAE** and **graphics**.

For ARIMA model

The best model was the model **ARIMA (24,0,1).** We can observe with the **confusion matrix and the table score** below that it is the best one:



	0	8	16	24
0	58.3101	40.0153	39.9563	38.2095
1	48.4501	39.8911	39.9556	38.2093
2	44.4207	NaN	39.9590	NaN

Figure 4: Confusion matrix and fune-tuning score table of ARIMA model

The result during the test is below:

```
Trained ARIMA in 44.04 seconds.
Train MAE  ARMA(24,0,1): 38.21
Test MAE  ARMA(24,0,1): 48.36
```

Here are graphics inspected to complete the evaluation:

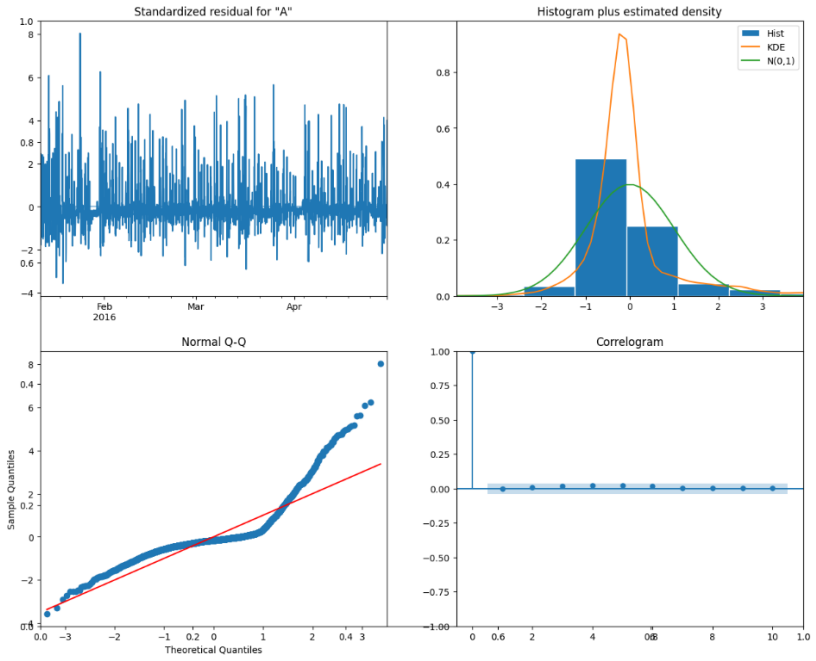


Figure 6: Residual inspection

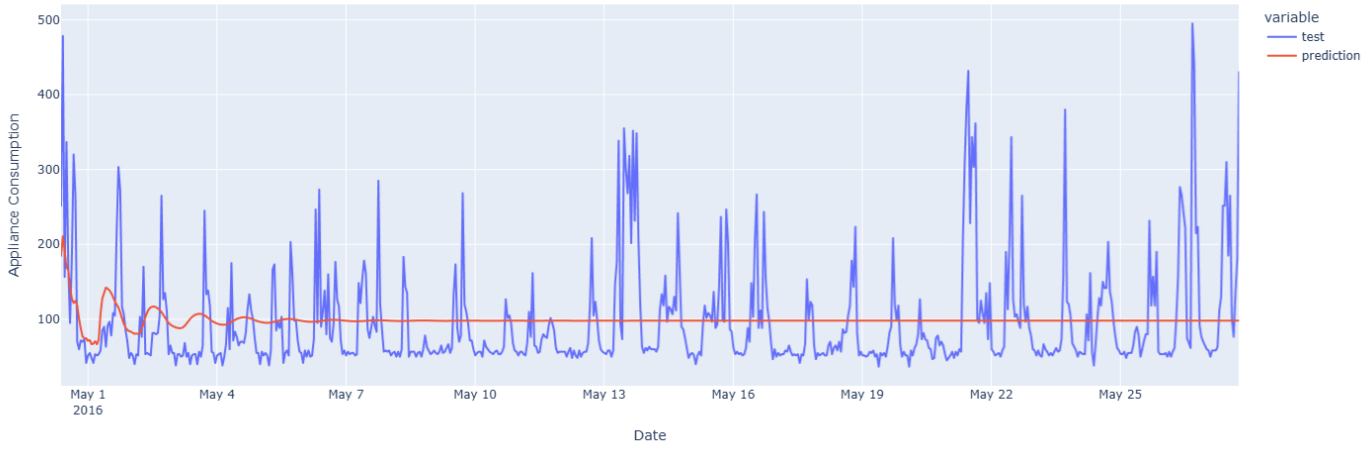


Figure 7: Comparison between true test data and model prediction

For LSTM model:

The results are below:

```
LSTM Train MAE: 95.65
LSTM Test MAE: 93.35
```

We also check the model ability to generalize we **the model Loss graphic**:

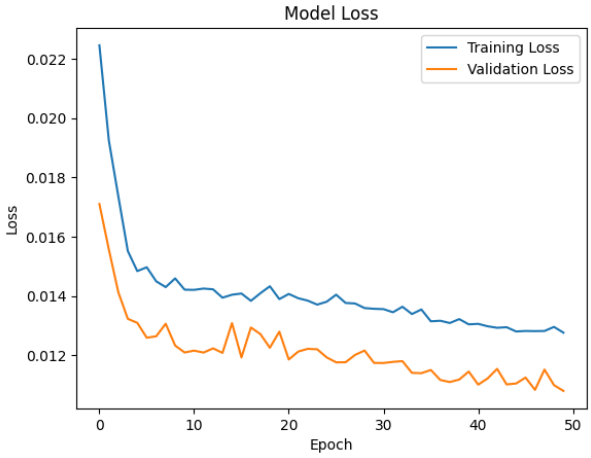


Figure 7: LSTM generalization inspection graphic

We concluded that there is **no overfitting and underfitting**.

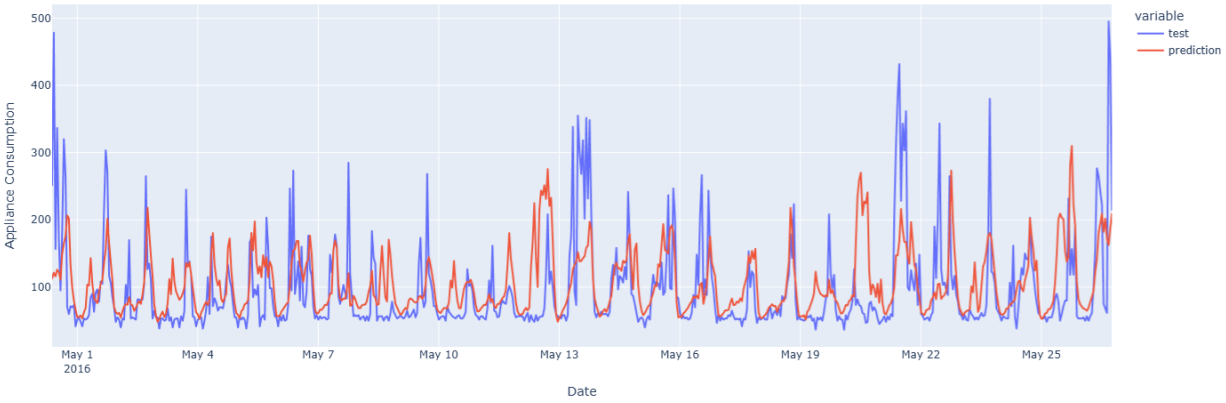


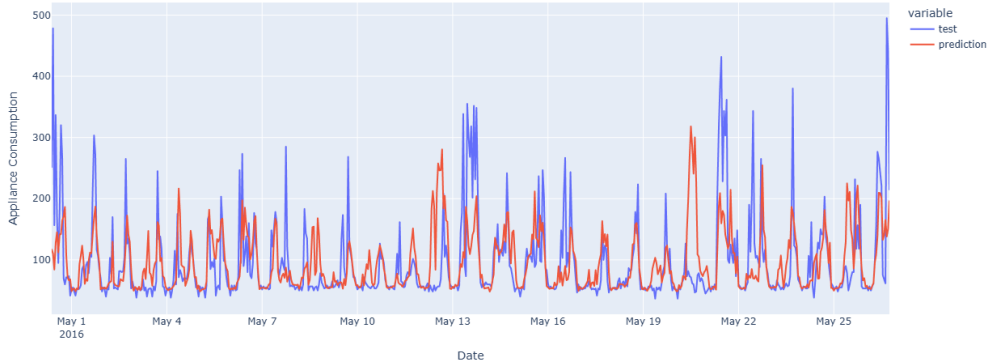
Figure 8: comparison between prediction and true test data

For XGBoost model

The result is:

```
LSTM Train MAE: 98.65
LSTM Test MAE: 96.35
```

Comparaison



5. Model Deployment

For the deployment of the model, we followed these steps:

1. **Best model saving**

We save the model (**LSTM**) in a **binary format using pickle**. We also save **the scaler** used and the **input data size**. The code is below:

```
# add LSTM model to models
models["lstm"] = lstm_model
model_deploy_data["scaler"] = scaler
model_deploy_data["look_back"] = 24
```

2. **API building and testing**

In order to use our we have successful created **an API with Django.** This **API** contains:

a. **A Make prediction function**

This function is the function used to make the prediction. It takes **the model, the last 24 data in the original dataset, the scaler and the number of features** as parameters. Here is the code:

```
# predictions Function
def make_prediction(model, X, scaler=None, features=1):
    if X.ndim == 2:
        X = X.reshape((X.shape[0], X.shape[1], features))
    elif X.ndim != 3:
        raise ValueError("Input data must be 2D or 3D.")
    predictions = model.predict(X)
    if scaler is not None:
        predictions = scaler.inverse_transform(predictions) # for real value predict
    return predictions.flatten()
```

b. **A Model loading function**

This function is used to **load the model and the scaler saved** during the building phase. The code is below:

```
def load_model(model_path):
    with open(model_path, 'rb') as f:
        final_model = pickle.load(f)

        # Read model object
        model = final_model['model']
        scaler = final_model['scaler']
        look_back_window = final_model['look_back']

    return model, scaler,look_back_window
```

c. **API view**

The **API view** that processes the user request code is below. The user should provide **the number of hours** for which he wants to make the prediction in a **json object**:

```
# Prediction View
@api_view(['POST'])
def EnergypredictionAPIView(request):
    if request.method == 'POST':
        results = []
        # Hours to predict data
        hours = request.data.get("hours")

        if hours is None:
            raise exceptions.ParseError("No hours provided.")
        hours = int(hours) # convert into in

        # Load model and scaler
        final_model_path = os.path.join(
                os.path.dirname(__file__), 'aiModel/final_model.pkl'
        )
        model, scaler,look_back_window = load_model(final_model_path)
        #  Last look_back_window hours data
        past_hours_data  = previewData()

         # Ensure past_hours_data has at least look_back_window data points
        if len(past_hours_data) < look_back_window:
            raise exceptions.ParseError("Insufficient past data for prediction.")

        # Reshape data to fit the model requirements
        new_data_reshaped= past_hours_data.reshape(-1, 1)
        new_data_scaled  = scaler.transform(new_data_reshaped)

        # Data send to the model
        input_sequence   = new_data_scaled[-look_back_window:, :]

        # Make predictions for the specified number of hours
        for _ in range(hours):
            predictions      = make_prediction(model, input_sequence, scaler)
            # Append the predicted value to results
            results.append(predictions[0])
            # Scale the prediction and append it to the input data
            prediction_data_scaled = scaler.transform(np.array([[predictions[0]]]))
            new_data_scaled = np.concatenate(
                   (new_data_scaled, prediction_data_scaled), axis=0
            )
            # Update the input sequence
            input_sequence = new_data_scaled[-look_back_window:, :]

        return Response({"predictions": results}, status=status.HTTP_200_OK)
    return Response(
              {"error": "Invalid request method"}, status=status.HTTP_400_BAD_REQUEST)
```

Request json object:

{
“hours”:” Number of hours”
}

The **API** endpoint is:

```
api/predict/
```

We haven’t included an **authentication** endpoint because we don’t want the users to be authenticate before using our model.

d. Frontend application

In order to used the model, we need a frontend application. For that purpose, we used **React** to create **a web application.** The user will have to specify **the number of hours** that he want to make the prediction and submit it to the model through that **website**.

6. Code Implementation

All the implementation code can be found inside **our notebook**. We have also submitted the code for the **API.**

Conclusion

The model refinement and test submission phases resulted in a robust LSTM model with improved performance metrics. Challenges included hyperparameter tuning and ensuring that the model generalized well to unseen data. The final model achieved a **MAE of 95.3** making it suitable for deployment in a real-world setting. The model was deployed using **a Django Rest API**, allowing it to receive real-time data and make predictions accordingly. The deployment involved integrating the model with a web service and ensuring it could handle multiple requests simultaneously.

References

List of libraries

Pandas, NumPy, matplotlib, seaborn, time, warnings, pickle, plotly express, sklearn, statsmodels, TensorFlow.

Documentation

[1] https://medium.com/latinxinai/evaluation-metrics-for-regression-models-03f2143ecec2