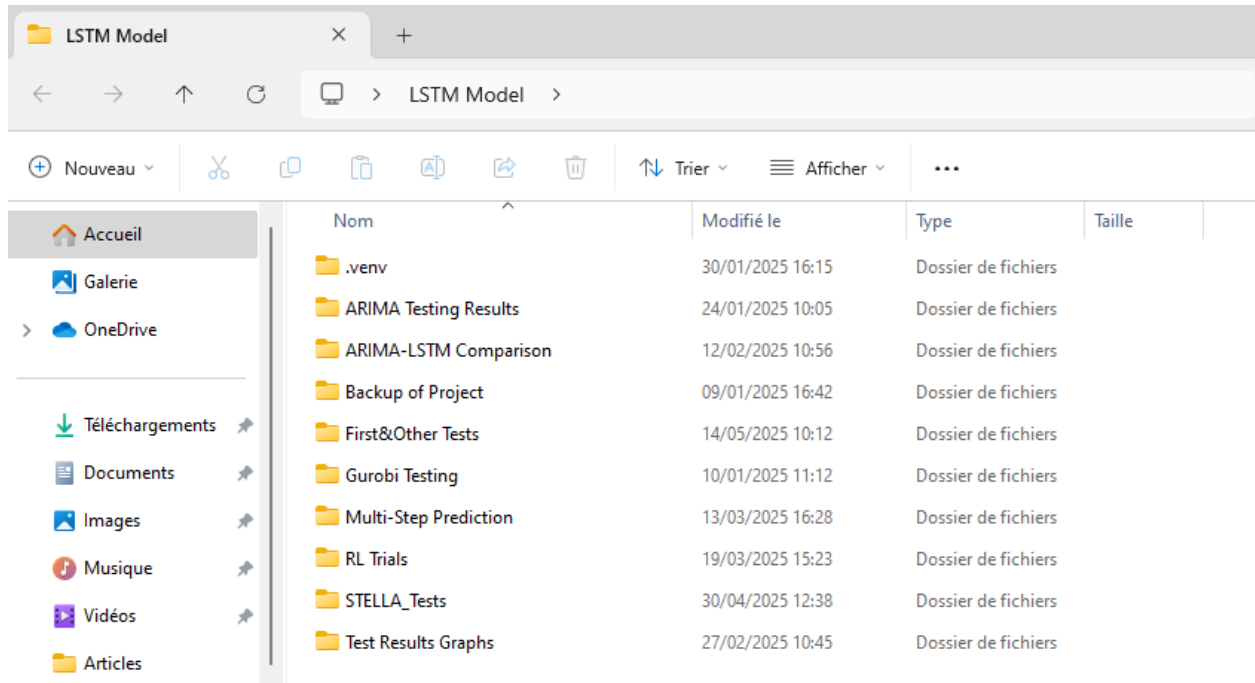


Instruction Manual for the LSTM – ARIMA Models

This instruction manual is written to explain which files to run, how certain parameters within the code works, and what information to change to do different tests.

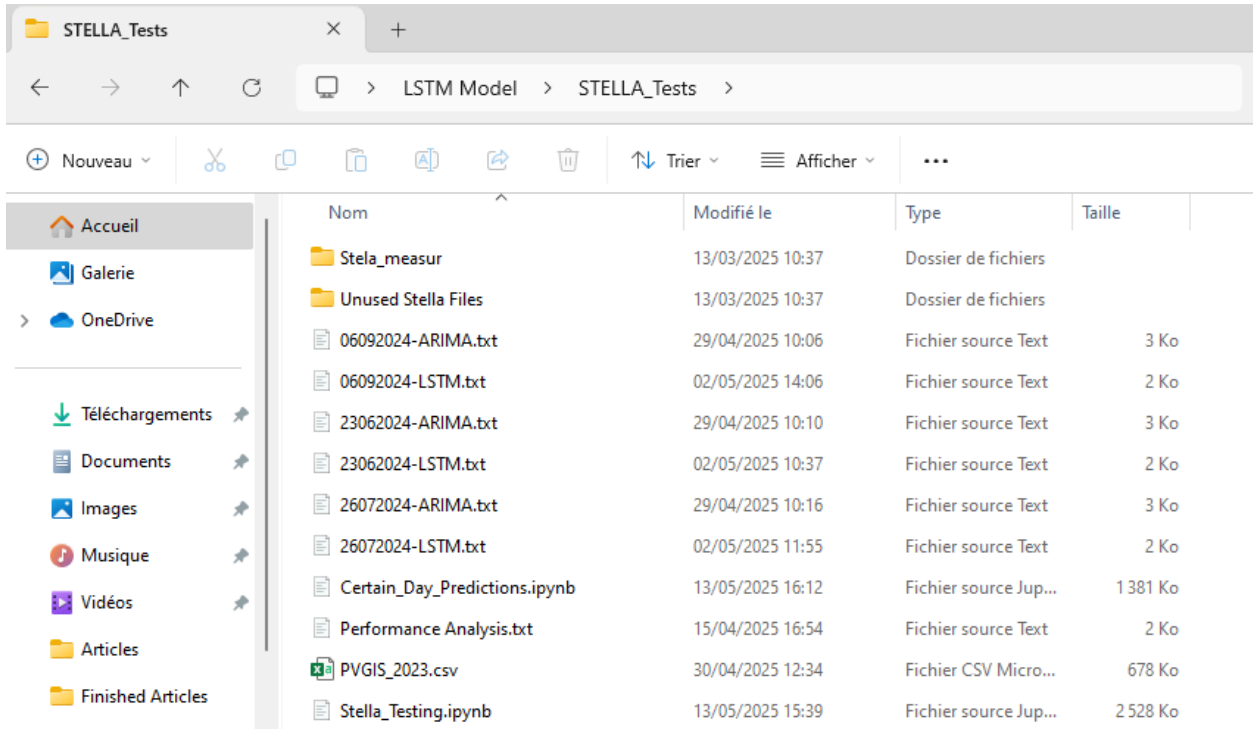
In the “LSTM Model” main folder, all of the most recent scripts are located inside the “STELLA_Tests” folder.



There are two .ipynb files located inside this folder titled “Certain_Day_Predictions.ipynb” and “Stella_Testing.ipynb”. “Certain_Day_Predictions” contains the most recent versions of every function and model, and should be enough to make predictions within specified dates and time intervals with one of the machine learning models: Long Short-Term Memory Model and Autoregressive Integrated Moving Average Model.

The “Stella_Testing” file contains the dynamically adjusted time interval forecasting method along with a script for comparing the results of both LSTM and ARIMA models. This script needs to be modified to work with static time interval forecasting comparisons as it is currently created in a way that will work on dynamic intervals.

The dated text files are the solar irradiance and ambient temperature predictions for that specific day, “PVGIS_2023” is the PVGIS data for the city center of Compiègne between the years of 2022 and 2023, and “Performance Analysis” text file contains small information on some of the tests that were run on the models, and how the models performed.



Nom	Modifié le	Type	Taille
Stela_meur	13/03/2025 10:37	Dossier de fichiers	
Unused Stella Files	13/03/2025 10:37	Dossier de fichiers	
06092024-ARIMA.txt	29/04/2025 10:06	Fichier source Text	3 Ko
06092024-LSTM.txt	02/05/2025 14:06	Fichier source Text	2 Ko
23062024-ARIMA.txt	29/04/2025 10:10	Fichier source Text	3 Ko
23062024-LSTM.txt	02/05/2025 10:37	Fichier source Text	2 Ko
26072024-ARIMA.txt	29/04/2025 10:16	Fichier source Text	3 Ko
26072024-LSTM.txt	02/05/2025 11:55	Fichier source Text	2 Ko
Certain_Day_Predictions.ipynb	13/05/2025 16:12	Fichier source Jup...	1 381 Ko
Performance Analysis.txt	15/04/2025 16:54	Fichier source Text	2 Ko
PVGIS_2023.csv	30/04/2025 12:34	Fichier CSV Micro...	678 Ko
Stella_Testing.ipynb	13/05/2025 15:39	Fichier source Jup...	2 528 Ko

Certain_Day_Predictions

While the codes and models themselves have a lot of comments on how a particular line of code works, this manual will go over them in general to give more detailed information on some aspects.

Before starting anything, it is important to fix a random seed for reproducibility of the results, and to eliminate the randomness of the models for the tests. This line of code makes it so that if the parameters of the models and the data given to the models are identical, they will give the same predictions in later tests as well. During practical deployment, this line of code should be commented out / deleted to keep the models' random but supervised learning qualities.

```
[2] # fix random seed for reproducibility
    tf.random.set_seed(7)
```

Data Pre-Processing

This file contains 2 separate data pre-processing steps for two different datasets: Adjusting the data for the STELLA platform, and adjusting the data for the PVGIS dataset. The STELLA pre-processing is done first.

I. STELLA Data

In the part given below, the script takes a folder and merges every text file within that folder in a single dataframe. The names of these text files are then used as the date information of the dataframe. Three things are variable depending on the dataset that is being used: “folder_path” variable needs to be equal to the actual folder path that contains the text files to be read / merged, “column_names” variable needs to be an array with the names of the desired columns and with the same amount of elements as the text files, and the “delimiter” parameter inside “read_csv()” method needs to be the command that specifies the separator of the elements of the text files (commas, tabs, etc.).

```
# Define the folder path where .txt files are stored
folder_path = "Stela_measur" # Change this to your actual folder path

# Use glob to get all .txt files in the folder
txt_files = glob.glob(os.path.join(folder_path, "*.txt"))

# Create an empty list to store DataFrames
df_list = []

# The column names of the DataFrame to be created
column_names = ["Time", "Tpv", "Tamb", "g", "Vwd"]

# Loop through each .txt file and read it
for file in txt_files:
    df = pd.read_csv(file, delimiter="\t", header=None, names=column_names) # Read with tab separator, change delimiter if separator is different
    df["Date"] = os.path.basename(file) # Add a new column with the file name
    df = df.drop([0])
    df_list.append(df) # Append to the list

# Concatenate all DataFrames into one big DataFrame
df_Stella_All = pd.concat(df_list, ignore_index=True)

# Display the first few rows
print(df_Stella_All.head())
```

The next part of the script moves the date column to the first position on the dataframe for easier readability and structure, converts the date strings and information into actual DateTime objects, and drops the NaN values so that the ML models will not stop training when they encounter a NaN data. Here, it is important to change the variables of “year”, “month”, and “day” depending on the indices in which they end in the string. For example, in the original dataset that was used, the date information that was collected before data-processing is given on the right. Here, the dates are given with the structure of “year-month-day”, and the indices in which those specific numeric values end are 4, 6, and 8 respectively. Therefore, those values were used in the variables for the original dataset.

	Time	Tpv	Tamb	g	Vwd	Date
0	00:02:13	NaN	NaN	NaN	NaN	20240101.txt
1	00:05:59	7	9	2	3	20240101.txt
2	00:10:13	7	9	0	2	20240101.txt
3	01:17:40	6	9	0	1	20240101.txt
4	01:22:13	6	9	0	3	20240101.txt

```
# Change the following values to the indices in which the date's property ends
year = 4 # Index in which year ends
month = 6 # Index in which month ends
day = 8 # Index in which day ends
df_Stella_copy['Date'] = df_Stella_copy['Date'].str[:year] + ':' + df_Stella_copy['Date'].str[year:month] + ':' + df_Stella_copy['Date'].str[month:day] + ':'
df_Stella_copy['d&t'] = df_Stella_copy['Date'] + df_Stella_copy['Time']
```

After this step, the created dataframe should look something like the following:

	Tpv	Tamb	g	Vwd
d&t				
2024-01-01 00:05:59	7	9	2	3
2024-01-01 00:10:13	7	9	0	2
2024-01-01 01:17:40	6	9	0	1
2024-01-01 01:22:13	6	9	0	3
2024-01-01 02:33:08	6	9	1	4
...

Now that the dataframe is clean of NULL values and with an organized Date column, the last things need to be done are interpolation and resampling of the data. For this step, every column of the dataframe besides the “d&t” (date&time) column is converted into a numerical value to alleviate the risk of object type mismatching. It is important to write the names of every column besides the “d&t” column individually.

```
# Using to_numeric() method with downcasting
df_Stella_copy['Tpv'] = pd.to_numeric(df_Stella_copy['Tpv'], downcast='integer', errors='coerce')
df_Stella_copy['Tamb'] = pd.to_numeric(df_Stella_copy['Tamb'], downcast='integer', errors='coerce')
df_Stella_copy['g'] = pd.to_numeric(df_Stella_copy['g'], downcast='integer', errors='coerce')
df_Stella_copy['Vwd'] = pd.to_numeric(df_Stella_copy['Vwd'], downcast='integer', errors='coerce')
```

During the resampling phase, it is important to change the time interval to align with the tests' purposes. In the original testing, the prediction time intervals were chosen as one hour intervals. To choose minute-based time intervals, the “resample()” method should be used with the string “_T” like 30T (thirty minutes) or 10T (ten minutes).

```
# Resample to 1 hour intervals, taking the mean of available values
df_resampled = df_Stella_copy.resample('1H').mean()

# Interpolate missing values linearly
df_resampled = df_resampled.interpolate(method='linear')

# Display the first few rows
print(df_resampled.head())
```

II. PVGIS Data

Pre-processing of the PVGIS data is a lot simpler since the data has no irregularities or NULL values within it, making it possible to forgo a lot of the previous steps. One important aspect of this step is, once again, to choose the actual file path for the script to read, and modify the “skiprows” and “skipfooter” parameters to fit the interval of the actual data within the .csv file for the code to read. Then, the features to be predicted can be chosen easily with the “.iloc()” method.

```
# Reading the datasets and creating the dataframes
# Change the skiprows and skipfooter depending on the length of your .csv file
dataset = pd.read_csv("PVGIS_2023.csv", skiprows=8, skipfooter=10, engine='python') # Change to your actual file path
df_Compiègne = pd.DataFrame(dataset)

print(df_Compiègne.head())

   time  G(i)  H_sun  T2m  WS10m  Int
0 20220101:0010  0.0  0.0  10.99  2.55  0.0
1 20220101:0110  0.0  0.0  11.01  2.41  0.0
2 20220101:0210  0.0  0.0  11.09  2.21  0.0
3 20220101:0310  0.0  0.0  11.05  2.14  0.0
4 20220101:0410  0.0  0.0  10.80  2.34  0.0

df_Compiègne['time'] = pd.to_datetime(df_Compiègne['time'].str.replace(':', ''), format='%Y%m%d%H%M', infer_datetime_format=True)
df_Compiègne = df_Compiègne.set_index(['time'])

df_Compiègne_Temp = df_Compiègne.iloc[:, [0, 2]] # Features to be predicted/trained on

print(df_Compiègne_Temp.head())
```

Necessary Functions

Most of the functions do not require anything to be changed, but a general overview on what a specific function does will be written here. These functions are strictly used for the training of the LSTM model, and they are not needed to be initialized for the implementation of the ARIMA model.

- **to_supervized():** This function creates supervised datasets from the input data so that the model will be able to guide itself while training by drawing input-output correlations between results. In this function, only the following line needs to be changed if multivariate tests are being conducted:

```
if out_end <= len(data):
    X.append(data[in_start:in_end, :])
    y.append(data[in_end:out_end, 0]) # Whichever feature needs to be predicted
```

Here, the value of 0 needs to be changed to the column number of the feature that is being predicted. If univariate tests are being conducted, then this line does not need to be modified.

- **forecast():** This function is the main function that uses the trained models to predict future values.
- **reframe_datasets():** This function reframes the datasets into adjusted lengths to later convert them into supervised datasets.
- **build_model():** The main function that builds an ML model by utilizing the training data to train the model. Every parameter within this function can be modified to optimize and tune the model to the test's specific needs; the epoch number of the model can be increased to improve the model's accuracy at the cost of training duration, etc. Detailed information on the parameters is given below:
 - **Batch Size:** Number of samples per gradient update. Increasing it decreases training duration at the cost of prediction accuracy.
 - **Epochs:** Number of epochs to train the model. An epoch is an iteration over the entire x and y data provided. Increasing it increases prediction accuracy at the cost of training duration.
 - **Verbose:** Verbosity mode. Prints the progress of the fitting of the model.
 - **LSTM Hidden Layer Units:** Used to generate the weight matrix. The model will read the input sequence and will output an element vector with this many elements (one output per unit) that captures features from the input sequence. Needs to be finely tuned as both increasing and decreasing it may affect the training durations and model accuracy negatively.
 - **Activation:** Activation function to use. 'tanh' activation function is assumed to work better with longer datasets, but the most common one is "relu".
 - **Loss:** Loss function to use.
 - **Optimizer:** String (name of optimizer) or optimizer instance.
- **evaluate_model():** The main function that takes the historical training data and the normalized testing data, and conducts forecasting until the end of the specified date / the end of the testing data. The "forecast()" function is used within this function, and the function uses the model that was built / trained by the "build_model()" function to make predictions by considering the past "n_input" number of time steps. The number of consecutive predictions made by the model is given to the function with "n_output". Finally, after the predictions are made, both the prediction values and the testing data are converted to real values by using the original scaler that was used to normalize the datasets. While the function is long, only a small part of the function should be changed depending on the test's specifications: If the tests are being conducted with multivariate data, the target scaler within the function should be modified to represent the index number of the predicted feature. In the original testing, if solar irradiance and ambient temperature were fed into the model in that order and the ambient temperature was chosen to be predicted, the index number of the column should have been modified to 1 instead of 0.

```

# Initialize its internal parameters using the ones from the full scaler
# The scaler must belong to the targeted feature's column
target_scaler.min_ = scaler.min_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.scale_ = scaler.scale_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.data_min_ = scaler.data_min_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.data_max_ = scaler.data_max_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.data_range_ = scaler.data_range_[0].reshape(1,) # Change 0 to desired feature's index

```

Initializations & Plotting

The training and testing dataset initializations, along with the plotting of real values and prediction results, are mostly identical between the two models. Firstly, the features to be used for training the model, along with a testing data with a single feature, is selected. Then, the input value (how many past real values to consider while predicting the next outputs) and the output value (how many future timesteps to predict) is specified. Finally, the date interval to use to train the models is selected both for the training and testing datasets.

```

# Select the features to train the model on
df_resampled_tamb = df_resampled['Tamb']
df_resampled_g = df_resampled['g']

# Select the features to predict with the model
df_Compiegne_tamb = df_Compiegne['T2m']
df_Compiegne_g = df_Compiegne['G(i)']

# Change the following variables depending on how many predictions you want/how many previous timesteps to consider
n_input = 336 # How many past real values (timesteps) to consider while predicting the next outputs
n_output = 24 # How many future timesteps to predict

```

```

# Training data - The date interval can be chosen as desired, but it is necessary for its structure to be similar
train_df = df_Compiegne_tamb[(df_Compiegne_tamb.index < '2023-06-23') & (df_Compiegne_tamb.index >= '2023-01-06')]
train_df = train_df.to_frame()

# Testing data - The date interval can be chosen as desired, but it is necessary for its structure to be similar
test_df = df_resampled_tamb[(df_resampled_tamb.index < '2024-06-24') & (df_resampled_tamb.index >= '2024-06-23')]
test_df = test_df.to_frame()

```

After this step, the code normalizes both datasets, runs and builds an LSTM model depending on the specifications written in the **build_model()** function, and makes predictions with the **evaluate_model()** function. Nothing should be modified until the plotting step, where the only things necessary to be changed should be the column names to represent the predicted feature.

```

# evaluate forecasts
rmse = np.sqrt(mean_squared_error(test_df['Tamb'].values, flattened_predictions_tamb)) # Change the column name for different predictions

```

```
plt.plot(test_df['Tamb'].values, 'g', label='Original Dataset') # Change the column name for different predictions
```

Finally, the predicted values are output into a text file. The “time_interval” variable should be changed depending on the resampling interval, and the output file’s name can be changed depending on the test’s specifications.

```
# Output the results into a text file
start_time = datetime.strptime("00:00:00", "%H:%M:%S")
time_interval = timedelta(minutes=60) # Should be the same as the resampling interval

with open("23062024-LSTM.txt", "w") as f: # Change the name of the text file depending on the day
    for i in range(n_output):
        current_time = (start_time + i * time_interval).strftime("%H:%M:%S")
        line = f"{current_time},{flattened_predictions_tamb[i]},{flattened_predictions_g[i]}\n"
        f.write(line)
```

ARIMA Model

For the ARIMA model’s implementation and training, the initialization phase is identical to that of the LSTM model, but the model is built and utilized to make predictions in the same cell, not using any other functions. The ARIMA model is initialized and fit in just two lines, with the following parameters being open to modifications based on the test’s specifications:

- **Lag Order (p):** Number of lag observations in the model, which identifies how many previous time steps will be considered to conduct forecasting. The previous “n_input” variable is normally used here, so it should not require any modifications.
- **Degree of Differencing (d):** Number of times the raw observations are differenced. If the data is non-stationary, this parameter is used to convert the data into a stationary dataset for the model to accurately be trained.
- **Order of Moving Average (q):** The size of the moving average window. Lets the model consider the error margins of previous predictions to reach a future prediction.

```
model_arima = ARIMA(history_arima, order=(n_steps,0,0))
model_fit = model_arima.fit()
```

Other parameters exist for different ARIMA versions (Auto ARIMA, SARIMAX, etc.) but only p, d, q values are important for a simple ARIMA model like what this script implements.

Before the plotting step of the predictions, both the testing and prediction datasets are converted into their real values by using the original scaler for the normalization step. This step is identical to the rescaling step of the LSTM model.


```
# Initialize its internal parameters using the ones from the full scaler
# The scaler must belong to the targeted feature's column
target_scaler.min_ = scaler.min_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.scale_ = scaler.scale_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.data_min_ = scaler.data_min_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.data_max_ = scaler.data_max_[0].reshape(1,) # Change 0 to desired feature's index
target_scaler.data_range_ = scaler.data_range_[0].reshape(1,) # Change 0 to desired feature's index
```

The plotting steps, along with the text output steps, are identical to the LSTM implementation, so if anything is changed in those previous steps, the ARIMA steps should also be modified similarly.

Stella_Testing

While most aspects of the file “Stella_Testing” are identical to “Certain_Day_Predictions”, like the data pre-processing, there is also a significantly different method of forecasting values that is implemented in this file. Here, both models shift how many timesteps further they predict as they forecast, creating dynamically shifting prediction intervals. For example, a model starts by predicting what the ambient temperature will be 10 minutes later from now, but when it makes a certain amount of predictions, it starts to predict what the values will be 30 minutes later, and eventually an hour later. This method was an interesting thought experiment that has the potential to alleviate execution durations of the models, along with creating more organized plots.

In this code, every step is identical to what is being done on “Certain_Day_Predictions” until the necessary functions, which adds two more functions with a few changes from their original counterparts:

- ***forecast_dynamic()***: A version of the function “forecast()” that supports dynamic intervals. Nothing should be changed in this function.
- ***evaluate_models_dynamic()***: A version of the function “evaluate_models()” that runs with dynamic intervals. This function not only makes the predictions and rescales the results like previously, but it also creates a dynamic list for the model to consider the testing values dynamically. Two important variables can be modified in the dynamic list creation loop:
 - The “**k**” values represent how many timesteps to jump forward, representing the intervals between prediction points. In the original testing, the data had 10 minutes between every row, and the dynamically adjusted intervals were chosen to be 10 minutes, 30 minutes, and 60 minutes. So the variable “k” represented every 10 minutes.

- The “if-elif-else” thresholds represent how many predictions to make before jumping to another interval. In the original testing, the first 12 hours into the future were chosen to be forecast in 10 minute intervals, and the next 36 hours were chosen to be forecast in 30 minute intervals. So the thresholds were represented with 72 and 144.

```
dynamic_values = list()
dynamic_count = 0
k = 0

while (k <= len(test_dynamic)):
    if dynamic_count <= 72:
        dynamic_values.append(test_dynamic[k:k+1])
        k += 1
    elif dynamic_count <= (72 + 72):
        dynamic_values.append(test_dynamic[k:k+1])
        k += 3
    else:
        dynamic_values.append(test_dynamic[k:k+1])
        k += 6
    dynamic_count += 1

dynamic_values = array(dynamic_values)
dynamic_values = dynamic_values.astype('float32')
dynamic_values = dynamic_values.reshape((dynamic_values.shape[0]*dynamic_values.shape[1], dynamic_values.shape[2]))
scaler_dynamic = MinMaxScaler(feature_range=(0, 1))
test_dynamic = scaler_dynamic.fit_transform(dynamic_values)
```

The same modifications are also located in the main forecasting loop to predict dynamically.

```
while (i <= len(test_resaped)):

    # Determine n_output dynamically
    if (prediction_count <= 72):
        interval_count = 1 # 10 minute predictions
    elif (prediction_count <= (72 + 72)):
        interval_count = 3 # Every 30 minutes
    else:
        interval_count = 6 # Every 1 hour

    # predict the day
    yhat_sequence = forecast_dynamic(model, history, n_input, interval_count)
```

Finally, the rescaling step also needs modifications depending on the amount of features the code is being run with:

```
# Initialize its internal parameters using the ones from the full scaler
# The scaler must belong to the targeted feature's column
target_scaler.min_ = scaler_dynamic.min_[0].reshape(1,)
target_scaler.scale_ = scaler_dynamic.scale_[0].reshape(1,)
target_scaler.data_min_ = scaler_dynamic.data_min_[0].reshape(1,)
target_scaler.data_max_ = scaler_dynamic.data_max_[0].reshape(1,)
target_scaler.data_range_ = scaler_dynamic.data_range_[0].reshape(1,)
```

After the necessary functions are initialized, the model trains and makes predictions like usual. However, since the predictions represent dynamic intervals, plotting these results also require dynamic plot intervals. In the following part, the “start_date” variable should be equal to the first timestamp of the forecasting results. Then, to generate the corresponding time index, the parameters of “periods” and “freq” should be changed depending on the intervals and the number of predictions made in a specific interval.

```
# Assume 'start_date' is the first timestamp of your forecast
start_date = pd.Timestamp("2024-05-01")

# Generate corresponding time index
hourly_dates = pd.date_range(start=start_date, periods=72, freq='10T') # First 72 points (10 mins)
six_hour_dates = pd.date_range(start=hourly_dates[-1] + pd.Timedelta(hours=3), periods=72, freq='30T') # Next 72 points (half-hour)
daily_dates = pd.date_range(start=six_hour_dates[-1] + pd.Timedelta(days=6), periods=len(predictions_dynamic)-144, freq='H') # Remaining points (hourly)
```

The same modifications should also be made in the following part:

```
# Manually create a modified x-axis index for readability
x_ticks = np.concatenate([
    np.linspace(0, 72, len(hourly_dates)),
    np.linspace(72, 144, len(six_hour_dates)),
    np.linspace(144, 288, len(daily_dates))
])
```

The dynamic forecasting steps, along with the dynamic plotting, are identical in both the LSTM and ARIMA models. If a change is made in one of them, the same change should also be done on the other implementation.