Forecasting framework

Release 1

Gianluca Ferro

Sep 03, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

This framework is designed to provide the main blocks for implementing and using many types of machine learning models for time series forecasting, including statistical models (ARIMA and SARIMA), Long Short Term Memory (LSTM) neural networks, and Extreme Gradient Boosting (XGB) models. Other models can also be integrated, by incorporating the corresponding tools into each respective block of the framework. The data preprocessing block makes possible to train and test the models on datasets with varying structures and formats, allowing a robust support for handling NaN values and outliers. The framework comprises a main file that orchestrates the various implementation phases of the models, with initial settings provided as command-line arguments using a parser (whose parameters are presented in the Appendix). The code supports four distinct modes of operation: training, testing, combined training and testing, and fine tuning. Various configurations of the framework, using different terminal arguments, are present in the JSON files (*launch.json* for debug and *tasks.json* for code usage); however, using consistent command line arguments, it is possible to create custom configurations by passing parameters directly through the terminal.

CHAPTER

TWO

FRAMEWORK ARCHITECTURE

The main blocks of the framework are data loading, data preprocessing, training, testing, and performance measurement. Once the model is selected, the file located in the *Predictors* folder corresponding to that model will be used. Each of these files contains the classes that implement the training and testing phases of the model. The blocks of the main code also use classes and functions from a corresponding file located in the *tools* folder, that includes functionalities as data loading, data preprocessing, optional time series analysis and performance measurement.

CHAPTER

THREE

FRAMEWORK TOOLS

Below are reported the classes from the *tools* folder.

3.1 Data Loader

This module contains the *DataLoader* class, specifically crafted to load and preprocess datasets for various types of machine learning models, including LSTM, XGB, ARIMA and SARIMA. The class efficiently handles datasets from multiple file formats and prepares them for model-specific requirements. It is essential to specify the correct date format present in the dataset using the *–date_format* parser argument to avoid errors during data loading. If the date column is not the first column in the dataset, its index must be specified using the *–time_column_index* command to ensure accurate processing. The *load_data* method converts date columns to datetime objects and adjusts dataset indices to align with the chosen model type. Additionally, the class utilizes specific dates provided through the *–date_list* input, filtering and structuring the data accordingly.

class `data_loader.`**`DataLoader`**(*file_path*, *date_format*, *model_type*, *target_column*, *time_column_index=0*, *date_list=None*, *exog=None*)

Bases: `object`

Class for loading datasets from various file formats and preparing them for machine learning models.

Parameters

- **`file_path`** – Path to the dataset file.
- **`date_format`** – Format of the date in the dataset file, e.g., '%Y-%m-%d'.
- **`model_type`** – Type of the machine learning model. Supported models are 'LSTM', 'XGB', 'ARIMA', 'SARIMA', 'SARIMAX'.
- **`target_column`** – Name of the target column in the dataset.
- **`time_column_index`** – Index of the time column in the dataset (default is 0).
- **`date_list`** – List of specific dates to be filtered (default is None).
- **`exog`** – Name or list of exogenous variables (default is None).

`load_data`()

Loads data from a file, processes it according to the specified settings, and prepares it for machine learning models. This includes formatting date columns, filtering specific dates, and adjusting data structure based on the model type.

Returns

- A tuple containing the dataframe and the indices of the dates if provided in *date_list*.

3.2 Data Preprocessing

This module is equipped with the *DataPreprocessor* class, specifically designed for the preprocessing of time series data for machine learning models. A method for data splitting is present, that uses processed dates coming from the *DataLoader* object. The class contains also specific methods for tasks such as managing missing values, removing non-numeric columns, managing outliers, and appropriately scaling data. These methods are designed taking into account the sequential nature of time series data, providing moving windows for outlier detection and making sure that, if a dataset with a large sequence of NaN is detected, the code stops working due to the lack of useful data. Also, data scaling is applied to the test set by using statistics from the training set, avoiding thus data leakage. The class supports various operational modes like training, testing, and fine-tuning.

`class data_preprocessing.DataPreprocessor`(*file_ext*, *run_mode*, *model_type*, *df: DataFrame*, *target_column: str*, *dates=None*, *scaling=False*, *validation=None*, *train_size=0.7*, *val_size=0.2*, *test_size=0.1*, *folder_path=None*, *model_path=None*, *verbose=False*)

Bases: `object`

A class to handle operations of preprocessing, including tasks such as managing NaN values, removing non-numeric columns, splitting datasets, managing outliers, and scaling data.

Parameters

- **`file_ext`** – File extension for saving datasets.
- **`run_mode`** – Mode of operation ('train', 'test', 'train_test', 'fine_tuning').
- **`model_type`** – Type of machine learning model to prepare data for.
- **`df`** – DataFrame containing the data.
- **`target_column`** – Name of the target column in the DataFrame.
- **`dates`** – Indexes of dates given by command line with –date_list.
- **`scaling`** – Boolean flag to determine if scaling should be applied.
- **`validation`** – Boolean flag to determine if a validation set should be created.
- **`train_size`** – Proportion of data to be used for training.
- **`val_size`** – Proportion of data to be used for validation.
- **`test_size`** – Proportion of data to be used for testing.
- **`folder_path`** – Path to folder for saving data.
- **`model_path`** – Path to model file for loading or saving the model.
- **`verbose`** – Boolean flag for verbose output.

`conditional_print`(**args*, ***kwargs*)

Print messages conditionally based on the verbose attribute.

Parameters

- **`args`** – Non-keyword arguments to be printed
- **`kwargs`** – Keyword arguments to be printed

`detect_nan_hole`(*df*)

Detects the largest contiguous NaN hole in the target column.

Parameters

`df` – DataFrame in which to find the NaN hole

Returns
A dictionary with the start and end indices of the largest NaN hole in the target column

`manage_nan`(*df*, *max_nan_percentage=50*, *min_nan_percentage=10*, *percent_threshold=40*)

Manage NaN values in the dataset based on defined percentage thresholds and interpolation strategies.

Parameters

- `df` – Dataframe to analyze
- `max_nan_percentage` – Maximum allowed percentage of NaN values for a column to be interpolated or kept
- `min_nan_percentage` – Minimum percentage of NaN values for which linear interpolation is applied
- `percent_threshold` – Threshold percentage of NaNs in the target column to decide between interpolation and splitting the dataset

Returns
A tuple (df, exit), where df is the DataFrame after NaN management, and exit is a boolean flag indicating if the dataset needs to be split

`preprocess_data`()

Main method to preprocess the dataset according to specified configurations.

Returns
Depending on the mode, returns the splitted dataframe and an exit flag.

`print_stats`(*train*)

Print statistics for the selected feature in the training dataset.

Parameters
`train` – DataFrame containing the training data

`replace_outliers`(*df*)

Replaces outliers in the dataset based on the Interquartile Range (IQR) method. Instead of analyzing the entire dataset at once, this method focuses on a window of data points at a time. The window moves through the data series step by step. For each step, it includes the next data point in the sequence while dropping the oldest one, thus maintaining a constant window size. For each position of the window, the function calculates the first (Q1) and third (Q3) quartiles of the data within the window. These quartiles are used to determine the Interquartile Range (IQR), from which lower and upper bounds for outliers are derived.

Parameters
`df` – DataFrame from which to remove and replace outliers

Returns
DataFrame with outliers replaced

`split_data`(*df*)

Split the dataset into training, validation, and test sets. If a list with dates is given, each set is created within the respective dates, otherwise the sets are created following the given percentage sizes.

Parameters
`df` – DataFrame to split

Returns
Tuple of DataFrames for training, testing, and validation

`split_file_at_nanhole`(*nan_hole*)

Splits the dataset at a significant NaN hole into two separate files.

Parameters
`nan_hole` – Dictionary containing start and end indices of the NaN hole in the target column

3.3 Performance Measurement

class `performance_measurement.`**`PerfMeasure`**(*model_type*, *model*, *test*, *target_column*, *forecast_type*)

Bases: `object`

`get_performance_metrics`(*test*, *predictions*, *naive=False*)

Calculates a set of performance metrics for model evaluation.

Parameters

- **`test`** – The actual test data.
- **`predictions`** – Predicted values by the model.
- **`naive`** – Boolean flag to indicate if the naive predictions should be considered.

Returns
A dictionary of performance metrics including MSE, RMSE, MAPE, MSPE, MAE, and R-squared.

3.4 Time Series Analysis

`time_series_analysis.`**`ARIMA_optimizer`**(*train*, *target_column=None*, *verbose=False*)

Determines the optimal parameters for an ARIMA model based on the Akaike Information Criterion (AIC).

Parameters

- **`train`** – The training dataset.
- **`target_column`** – The target column in the dataset that needs to be forecasted.
- **`verbose`** – If set to True, prints the process of optimization.

Returns
The best (p, d, q) order for the ARIMA model.

`time_series_analysis.`**`SARIMAX_optimizer`**(*train*, *target_column=None*, *period=None*, *exog=None*, *verbose=False*)

Identifies the optimal parameters for a SARIMAX model.

Parameters

- **`train`** – The training dataset.
- **`target_column`** – The target column in the dataset.
- **`period`** – The seasonal period of the dataset.
- **`exog`** – The exogenous variables included in the model.
- **`verbose`** – Controls the output of the optimization process.

Returns
The best (p, d, q, P, D, Q) parameters for the SARIMAX model.

`time_series_analysis.adf_test(df, alpha=0.05, verbose=False)`

Performs the Augmented Dickey-Fuller test to determine if a series is stationary and provides detailed output.

Parameters

- **df** – The time series data as a DataFrame.
- **alpha** – The significance level for the test to determine stationarity.
- **verbose** – Boolean flag that determines whether to print detailed results.

Returns

The number of differences needed to make the series stationary.

`time_series_analysis.conditional_print(verbose, *args, **kwargs)`

Prints messages conditionally based on a verbosity flag.

Parameters

- **verbose** – Boolean flag indicating whether to print messages.
- **args** – Arguments to be printed.
- **kwargs** – Keyword arguments to be printed.

`time_series_analysis.ljung_box_test(model)`

Conducts the Ljung-Box test on the residuals of a fitted time series model to check for autocorrelation.

Parameters

model – The time series model after fitting to the data.

`time_series_analysis.multiple_STL(dataframe, target_column)`

Performs multiple seasonal decomposition using STL on specified periods.

Parameters

- **dataframe** – The DataFrame containing the time series data.
- **target_column** – The column in the DataFrame to be decomposed.

`time_series_analysis.optimize_ARIMA(endog, order_list)`

Optimizes ARIMA parameters by iterating over a list of (p, d, q) combinations to find the lowest AIC.

Parameters

- **endog** – The endogenous variable.
- **order_list** – A list of (p, d, q) tuples representing different ARIMA configurations to test.

Returns

A DataFrame containing the AIC scores for each parameter combination.

`time_series_analysis.optimize_SARIMAX(endog, order_list, s, exog=None)`

Optimizes SARIMAX parameters by testing various combinations and selecting the one with the lowest AIC.

Parameters

- **endog** – The dependent variable.
- **order_list** – A list of order tuples (p, d, q, P, D, Q) for the SARIMAX.
- **s** – The seasonal period of the model.
- **exog** – Optional exogenous variables.

Returns

A DataFrame with the results of the parameter testing.

`time_series_analysis.prepare_seasonal_sets(train, valid, test, target_column, period)`

Decomposes the datasets into seasonal and residual components based on the specified period.

Parameters

- **train** – Training dataset.
- **valid** – Validation dataset.
- **test** – Test dataset.
- **target_column** – The target column in the datasets.
- **period** – The period for seasonal decomposition.

Returns

Decomposed training, validation, and test datasets.

`time_series_analysis.time_s_analysis(df, target_column, seasonal_period)`

Performs a comprehensive time series analysis including plotting, stationarity testing, and decomposition.

Parameters

- **df** – The DataFrame containing the time series data.
- **target_column** – The column in the DataFrame representing the time series to analyze.
- **seasonal_period** – The period to consider for seasonal decomposition and autocorrelation analysis.

3.5 Utilities

`utilities.conditional_print(verbose, *args, **kwargs)`

Prints provided arguments if the verbose flag is set to True.

Parameters

- **verbose** – Boolean, controlling whether to print.
- **args** – Arguments to be printed.
- **kwargs** – Keyword arguments to be printed.

`utilities.load_trained_model(model_type, folder_name)`

Loads a trained model and its configuration from the selected directory.

Parameters

- **model_type** – Type of the model to load ('ARIMA', 'SARIMAX', etc.).
- **folder_name** – Directory from which the model and its details will be loaded.

Returns

A tuple containing the loaded model and its order (if applicable).

`utilities.save_buffer(folder_path, df, target_column, size=20, file_name='buffer.json')`

Saves a buffer of the latest data points to a JSON file.

Parameters

- **folder_path** – Directory path where the file will be saved.
- **df** – DataFrame from which data will be extracted.

- **target_column** – Column whose data is to be saved.
- **size** – Number of rows to save from the end of the DataFrame.
- **file_name** – Name of the file to save the data in.

utilities.**save_data**(*save_mode*, *validation*, *path*, *model_type*, *model*, *dataset*, *performance=None*, *naive_performance=None*, *best_order=None*, *end_index=None*, *valid_metrics=None*)

Saves various types of data to files based on the specified mode.

Parameters

- **save_mode** – String, 'training' or 'test', specifying the type of data to save.
- **validation** – Boolean, indicates if validation metrics should be saved.
- **path** – Path where the data will be saved.
- **model_type** – Type of model used.
- **model** – Model object to be saved.
- **dataset** – Name of the dataset used.
- **performance** – model performance metrics to be saved.
- **naive_performance** – naive model performance metrics to be saved.
- **best_order** – best model order to be saved.
- **end_index** – index of the last training point.
- **valid_metrics** – validation metrics to be saved.

CHAPTER

FOUR

FRAMEWORK MODELS

In the following are documented the models used inside the framework.

4.1 ARIMA Model

This module is designed to make forecasts on time series through ARIMA (Autoregressive Integrated Moving Average) models, encapsulated in the *ARIMA_Predictor* class. The model used comes from the *Statsmodels* library, and the optimization of the hyperparameters is done through grid search, by finding the model with the best AIC score. This class provides methods for predictive analysis of univariate time series, and is designed for both one-step ahead or multi-step ahead forecasts. One-step ahead predictions can be done in open loop mode, i.e. by updating at each forecast the model with the present observation, in order to make the model suitable for online learning settings.

4.1.1 ARIMA_Predictor

class ARIMA_model.**ARIMA_Predictor**(*run_mode*, *target_column=None*, *verbose=False*)

Bases: `Predictor`

A class used to predict time series data using the ARIMA model.

plot_predictions(*predictions*)

Plots the ARIMA model predictions against the test data.

Parameters
predictions – The predictions made by the ARIMA model

test_model(*model*, *last_index*, *forecast_type*, *ol_refit=False*)

Tests an ARIMA model by performing one-step ahead predictions and optionally refitting the model.

Parameters

- **model** – The ARIMA model to be tested
- **last_index** – Index of last training/validation timestep
- **forecast_type** – Type of forecasting ('ol-one' for open-loop one-step ahead, 'cl-multi' for closed-loop multi-step)
- **ol_refit** – Boolean indicating whether to refit the model after each forecast

Returns
A pandas Series of the predictions

train_model()

Trains an ARIMA model using the training dataset.

Returns
A tuple containing the trained model, validation metrics, and the index of the last training/validation timestep

unscale_predictions(*predictions*, *folder_path*)

Unscales the predictions using the scaler saved during model training.

Parameters

- **predictions** – The scaled predictions that need to be unscaled
- **folder_path** – Path to the folder containing the scaler object

4.2 SARIMA Model

This module implements Seasonal ARIMA (SARIMA) models for time series forecasting. It features the *SARIMA_Predictor* class, which leverages seasonal differencing and potentially exogenous variables, like Fourier terms for capturing seasonality. The model used comes from the *Statsmodels* library, and the optimization of the hyperparameters is done through grid search, by finding the model with the best AIC score. This model is used for univariate time series, and is designed for both one-step ahead or multi-step ahead forecasts. Like the ARIMA model, one-step ahead predictions can be done in open loop mode, i.e. by updating at each forecast the model with the present observation, in order to make the model suitable for online learning settings. For datasets with high frequency, like solar panel production with 15 min timesteps, the model may not be able to be trained or optimized with daily or weekly seasonality, due to high memory requirements. For this reason, a setting with additional Fourier terms as exogenous variable is present, and the choice of the hyperparameters could be done by inspection of the ACF and PACF plots. An optional rolling window cross validation technique is also implemented in the class.

4.2.1 SARIMA_Predictor

class SARIMA_model.**SARIMA_Predictor**(*run_mode*, *target_column=None*, *period=24*, *verbose=False*, *set_fourier=False*)

Bases: `Predictor`

A class used to predict time series data using Seasonal ARIMA (SARIMA) models.

plot_predictions(*predictions*)

Plots the SARIMA model predictions against the test data.

Parameters
predictions – The predictions made by the SARIMA model

test_model(*model*, *last_index*, *forecast_type*, *ol_refit=False*, *period=24*, *set_Fourier=False*)

Tests a SARIMAX model by performing one-step or multi-step ahead predictions, optionally using exogenous variables or applying refitting.

Parameters

- **model** – The SARIMAX model to be tested
- **last_index** – Index of the last training/validation timestep
- **forecast_type** – Type of forecasting ('ol-one' for open-loop one-step ahead, 'cl-multi' for closed-loop multi-step)

- **ol_refit** – Boolean indicating whether to refit the model after each forecast
- **period** – The period for Fourier terms if set_fourier is true
- **set_fourier** – Boolean flag to determine if Fourier terms should be included

Returns
A pandas Series of the predictions

train_model()

Trains a SARIMAX model using the training dataset and exogenous variables, if specified.

Returns
A tuple containing the trained model, validation metrics, and the index of the last training/validation timestep

unscale_predictions(*predictions*, *folder_path*)

Unscales the predictions using the scaler saved during model training.

Parameters

- **predictions** – The scaled predictions that need to be unscaled
- **folder_path** – Path to the folder containing the scaler object

4.3 XGB Model

This module encapsulates the XGB_Predictor class, which leverages the XGBoost machine learning library to forecast time series data. The class is specifically designed to incorporate extensive feature engineering including lag features, rolling window statistics, and optional Fourier transformations to capture seasonal patterns. It focuses on using these enhanced datasets to train and evaluate XGBoost models for precise predictions, offering functionalities for scaling, plotting, and performance assessment tailored to time series forecasting.

4.3.1 XGB_Predictor

class XGB_model.**XGB_Predictor**(*run_mode*, *target_column=None*, *verbose=False*, *seasonal_model=False*, *set_fourier=False*)

Bases: Predictor

A class used to predict time series data using XGBoost, a gradient boosting framework.

create_time_features(*df*, *lags=[1, 2, 3, 24]*, *rolling_window=24*)

Creates time-based features for a DataFrame, optionally including Fourier features and rolling window statistics.

Parameters

- **df** – DataFrame to modify with time-based features
- **lags** – List of integers representing lag periods to generate features for
- **rolling_window** – Window size for generating rolling mean and standard deviation

Returns
Modified DataFrame with new features, optionally including target column labels

`plot_predictions`(*predictions*, *test*, *time_values*)

Plots predictions made by an XGBoost model against the test data.

Parameters

- **predictions** – Predictions made by the XGBoost model
- **test** – The actual test data
- **time_values** – Time values corresponding to the test data

`train_model`(*X_train*, *y_train*, *X_valid*, *y_valid*)

Trains an XGBoost model using the training and validation datasets.

Parameters

- **X_train** – Input data for training
- **y_train** – Target variable for training
- **X_valid** – Input data for validation
- **y_valid** – Target variable for validation

Returns

A tuple containing the trained XGBoost model and validation metrics

`unscale_data`(*predictions*, *y_test*, *folder_path*)

Unscales the predictions and test data using the scaler saved during model training.

Parameters

- **predictions** – The scaled predictions that need to be unscaled
- **y_test** – The scaled test data that needs to be unscaled
- **folder_path** – Path to the folder containing the scaler object

4.4 LSTM Model

This module provides functionality for forecasting time series data using Long Short-Term Memory (LSTM) networks. It includes the *LSTM_Predictor* class, which employs a Keras (Tensorflow) model to make forecasts. The class supports advanced features such as input and output sequence length customization, optional Fourier transformation for seasonality, and the capability to handle multi-step forecasts with seasonality adjustments. A data windowing method is also included, in order to give to the neural network the correct input data shape. By correct setting of *input_len* and *output_len* command line parameters, both one-step or multi-step ahead predictions can be done.

4.4.1 LSTM Model Structure

The model consists of the following sequence of layers:

1. **Input Layer**:
2. **LSTM Layer**: - LSTM with 40 units
3. **Dropout Layer**: - First dropout layer with rate of 0.15
4. **LSTM Layer**: - Second LSTM layer with 40 units
5. **Dropout Layer**: - Second dropout layer with rate of 0.15
6. **LSTM Layer**: - Third LSTM layer with 40 units

7. **Dropout Layer**: - Third dropout layer with rate of 0.15
8. **Dense Layer**: - Dense layer for the final output

This structure of the LSTM model is configured to process temporal sequences, followed by multiple LSTM and dropout layers to prevent overfitting, concluding with a dense layer for output.

4.4.2 LSTM_Predictor

class `LSTM_model.`**`LSTM_Predictor`**(*run_mode*, *target_column=None*, *verbose=False*, *input_len=None*, *output_len=None*, *seasonal_model=False*, *set_fourier=False*)

Bases: `Predictor`

A class used to predict time series data using Long Short-Term Memory (LSTM) networks.

`data_windowing`()

Creates data windows suitable for input into LSTM models, optionally incorporating Fourier features for seasonality.

Returns
Arrays of input and output data windows for training, validation, and testing

`plot_predictions`(*predictions*, *y_test*)

Plots LSTM model predictions against actual test data for each data window in the test set.

Parameters

- **predictions** – Predictions made by the LSTM model
- **y_test** – Actual test values corresponding to the predictions

`train_model`(*X_train*, *y_train*, *X_valid*, *y_valid*)

Trains an LSTM model using the training and validation datasets.

Parameters

- **X_train** – Input data for training
- **y_train** – Target variable for training
- **X_valid** – Input data for validation
- **y_valid** – Target variable for validation

Returns
A tuple containing the trained LSTM model and validation metrics

`unscale_data`(*predictions*, *y_test*, *folder_path*)

Unscales the predictions and test data using the scaler saved during model training.

Parameters

- **predictions** – The scaled predictions that need to be unscaled
- **y_test** – The scaled test data that needs to be unscaled
- **folder_path** – Path to the folder containing the scaler object

4.5 NAIVE Model

This module introduces the *NAIVE_Predictor* class, designed to implement naive forecasting techniques for time series data to be used as a benchmark for analyzing the performance of the machine learning models. The class is constructed to provide basic prediction strategies, such as using the last observed value, the mean of the dataset, or the last observed seasonal value as forecasts. It is also useful for initial assessments of time series forecasting tasks, offering various naive methods to quickly generate forecasts without complex modeling.

4.5.1 NAIVE_Predictor

class `NAIVE_model.`**`NAIVE_Predictor`**(*run_mode*, *target_column*, *verbose=False*)

Bases: `object`

A class used to predict time series data using simple naive methods.

`forecast`(*forecast_type*)

Performs a naive forecast using the last observed value from the training set or the immediate previous value from the test set.

Parameters

`forecast_type` – Type of forecasting ('cl-multi' for using the training set mean, else uses the last known values)

Returns

A pandas Series of naive forecasts.

`mean_forecast`()

Performs a naive forecast using the mean value of the training set.

Returns

A pandas Series of naive forecasts using the mean.

`plot_predictions`(*naive_predictions*)

Plots naive predictions against the test data.

Parameters

`naive_predictions` – The naive predictions to plot.

`prepare_data`(*train=None*, *valid=None*, *test=None*)

Prepares the data for the naive forecasting model.

Parameters

- **`train`** – Training dataset
- **`valid`** – Validation dataset (optional)
- **`test`** – Testing dataset

`seasonal_forecast`(*period=24*)

Performs a seasonal naive forecast using the last observed seasonal cycle.

Parameters

`period` – The seasonal period to consider for the forecast.

Returns

A pandas Series of naive seasonal forecasts.

`unscale_predictions`(*predictions*, *folder_path*)

Unscales the predictions using the scaler saved during model training.

Parameters

- `predictions` – The scaled predictions that need to be unscaled
- `folder_path` – Path to the folder containing the scaler object

CHAPTER

FIVE

APPENDIX

Here are presented all the parameters that can be given to the argument parser, specifying their function.

5.1 Parser Arguments

The command-line parser in the forecasting framework configures settings for various model implementations and applications. It provides multiple options to tailor the time series analysis, model training, and testing processes.

5.1.1 General Arguments

–verbose
Minimizes the additional information provided during the program's execution if specified. Default: `False`.

–ts_analysis
If `True`, performs an analysis on the time series. Default: `False`.

–run_mode
Specifies the running mode, which must be one of 'training', 'testing', 'both', or 'fine tuning'. This parameter is required.

5.1.2 Dataset Arguments

–dataset_path
Specifies the file path to the dataset. This parameter is required.

–date_format
Specifies the date format in the dataset, crucial for correct datetime parsing. This parameter is required.

–date_list
Provides a list of dates defining the start and end for training, validation, and testing phases, tailored to the model's needs.

–train_size
Sets the proportion of the dataset to be used for training. Default: `0.7`.

–val_size
Sets the proportion of the dataset to be used for validation. Default: `0.2`.

–test_size
Sets the proportion of the dataset to be used for testing. Default: `0.1`.

–scaling
If `True`, scales the data. This is essential for models sensitive to the magnitude of data.

–validation
If `True`, includes a validation set in the data preparation process. Default: `False`.

–target_column
Specifies the column to be forecasted. This parameter is required.

–time_column_index
Specifies the index of the column containing timestamps. Default: `0`.

5.1.3 Model Arguments

–model_type
Indicates the type of model to be used. This parameter is required.

–forecast_type
Defines the forecast strategy: 'ol-multi' (open-loop multi-step), 'ol-one' (open loop one-step), or 'cl-multi' (closed-loop multi-step).

–valid_steps
Number of time steps to use during the validation phase. Default: `10`.

–steps_jump
Specifies the number of time steps to skip during open-loop multi-step predictions. Default: `50`.

–exog
Defines one or more exogenous variables for models like SARIMAX, enhancing model predictions.

–period
Sets the seasonality period, critical for models handling seasonal variations. Default: `24`.

–set_fourier
If `True`, incorporates Fourier terms as exogenous variables, useful for capturing seasonal patterns in data.

5.1.4 Other Arguments

–seasonal_model
Activates the inclusion of a seasonal component in models like LSTM or XGB.

–input_len
Specifies the number of timesteps for input in models like LSTM. Default: `24`.

–output_len
Defines the number of timesteps to predict in each window for LSTM models. Default: `1`.

–model_path
Provides the path to a pre-trained model, facilitating fine-tuning or continued training from a saved state.

–ol_refit
For ARIMA and SARIMA models, allows the model to be retrained for each new observation during open-loop forecasts. Default: `False`.

–unscale_predictions
If specified, predictions and test data are unscaled, essential for interpreting results in their original scale.

5.2 Usage Examples

1. **Training an ARIMA Model with Data Scaling and Validation**

   This example sets up the framework to train an ARIMA model, applying data scaling and including a validation dataset. Use the command below to execute the training process:

```
python main_code.py --run_mode training --dataset_path '/path/to/dataset.csv' --
↪date_format '%Y-%m-%d' --target_column 'sales' --model_type 'ARIMA' --scaling --
↪validation --date_list '2022-01-01' '2022-06-30' '2022-07-01' '2022-08-31' '2022-
↪09-01' '2022-09-30'
```

2. **Fine-Tuning a Pre-trained LSTM Model for Multi-step Forecasting**

   Demonstrates how to fine-tune an LSTM model for multi-step forecasting, specifying the length of input and output sequences. This setup also includes enabling a seasonal component for the LSTM model:

```
python main_code.py --run_mode fine_tuning --dataset_path '/path/to/dataset.csv' --
↪date_format '%Y-%m-%d' --target_column 'temperature' --model_type 'LSTM' --model_
↪path '/path/to/pretrained_model' --input_len 24 --output_len 3 --seasonal_model
```

3. **Testing an XGBoost Model with Feature Engineering**

   This example showcases testing an XGBoost model that incorporates Fourier features as part of its feature engineering process to capture seasonal patterns.

```
python main_code.py --run_mode testing --dataset_path '/path/to/dataset.csv' --date_
↪format '%Y-%m-%d' --target_column 'energy_consumption' --model_type 'XGB' --
↪seasonal_model --set_fourier --model_path '/path/to/pretrained_model' --unscale_
↪predictions
```

CHAPTER

SIX

REFERENCES

https://www.statsmodels.org/stable/examples/notebooks/generated/statespace_forecasting.html#Cross-validation
https://www.statsmodels.org/stable/generated/statsmodels.tsa.ar_model.AutoRegResults.append.html#statsmodels.tsa.ar_model.AutoRegResults.append

CHAPTER

SEVEN

INDICES

- genindex
- modindex
- search

PYTHON MODULE INDEX