Forecasting framework

Release 1

Gianluca Ferro

May 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, 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. Each block makes use of classes and functions from a corresponding file located in the *classes* folder, in order to implement the time series forecasting models features. For time series analysis and functions for loading/saving data there are two respective files in the *utils* folder. Below is the structure of the framework.

2.1 Main Code

main_code.**main**()

Main function to execute time series forecasting tasks based on user-specified arguments. This function handles the entire workflow from data loading, preprocessing, model training, testing, and evaluation, based on the configuration provided via command-line arguments.

The main code is divided into the following sections:

- **Data loading**

Loads data from the specified dataset path using the DataLoader class.

- **Preprocessing and dataset split**

Handling data preprocessing and dataset splitting based on the specified model type and runtime configurations. For test-only run mode, only the test set will be created.

- **Optional time series analysis**

Performs additional time series analysis if specified with the parser argument *–ts_analysis*. Functions used in this section include plotting the ACF and PACF diagrams to have an early estimate of the AR and MA parameters, as well as statistic tests for stationarity. For seasonal models, a seasonal-trend decomposition of the series can be performed.

- **Model loading for test or fine-tuning**

Depending on the run mode, loads a pre-trained model to perform fine-tuning or to handle the test-only mode. For statistical models, some operations on the index of the training and test set are required, to ensure that the endogenous and exogenous variables passed to the model during training have indexes that are contiguous to those of the pre-trained model.

- **Model training**

The training of the model is done using the corresponding method of the *ModelTraining* class. A buffer containing the last *buffer_size* elements of the training set is created, which can be used for further test or to create naive benchmark models. After the training phase data (including the model itself and validation metrics) are saved in the model's directory.

- **Model testing**

The class ModelTest is used in order to generate the predictions of the trained model.

- **Plot of predictions versus real data**
- **Performance measurement and saving**

After computing the metrics of the model, the performance is stored into a text file.

The *main* function orchestrates the loading, preprocessing, training/testing of models, optionally fine-tuning, and plotting of predictions.

2.2 Data Loader Module

class data_loader.**DataLoader**(*file_path*, *model_type*, *target_column*, *time_column_index=0*, *date_list=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
- **model_type** – Type of the machine learning model ('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)

load_data()

Load data from a file and convert the time column to the datetime format.

Returns

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

Class for loading data from various file formats. If the parameter –time_column_index is not specified, the code expects that the time column is the first of the dataset. For statistical models, the end date of the training set must be equal to the start date of the validation set; the same holds for validation and test set.

2.3 Data Preprocessing

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*, *seasonal_split=False*, *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
- **seasonal_split** – Boolean flag to use seasonal data splitting logic
- **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

create_time_features(*df*, *label=None*, *seasonal_model=None*, *lags=[1, 2, 3, 24]*, *rolling_window=24*)

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

Parameters

- **df** – DataFrame to modify with time-based features
- **label** – Label column name for generating features
- **seasonal_model** – Boolean indicating whether to add Fourier features for seasonal models
- **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

data_windowing(*train*, *valid*, *test*)

Prepare data windows for training, validation, and testing datasets, suitable for neural network models.

Parameters

- **train** – Training DataFrame
- **valid** – Validation DataFrame
- **test** – Testing DataFrame

Returns

Lists of input and target data arrays for training, validation, and test

`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`(*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

- `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 DataFrame using a rolling window and IQR method.

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

Returns
DataFrame with outliers replaced

`seasonal_split_data`(*df*)

Split the dataset into seasonal parts based on day conditions to form the training, validation, and testing datasets.

Parameters
`df` – DataFrame to be split

Returns
DataFrames for training, testing, and validation, based on the seasonal conditions applied

`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

Class for preprocessing data for machine learning models.

2.4 Training Module

class training_module.**ModelTraining**(*model_type: str*, *train*, *valid=None*, *target_column=None*, *verbose=False*)

Bases: object

Class for training various types of machine learning models based on the –model_type argument.

Parameters

- **model_type** – Specifies the type of model to train (e.g., 'ARIMA', 'SARIMAX', 'LSTM', 'XGB').
- **train** – Training dataset.
- **valid** – Optional validation dataset for model evaluation.
- **target_column** – The name of the target variable in the dataset.
- **verbose** – If True, enables verbose output during model training.

train_ARIMA_model()

Trains an ARIMA model using the training dataset.

Returns
A tuple containing the trained model and validation metrics.

train_LSTM_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.

train_SARIMAX_model(*target_train*, *exog_train*, *exog_valid=None*, *period=24*)

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

Parameters

- **target_train** – Training dataset containing the target variable.
- **exog_train** – Training dataset containing the exogenous variables.
- **exog_valid** – Optional validation dataset containing the exogenous variables for model evaluation.
- **period** – Seasonal period of the SARIMAX model.

Returns
A tuple containing the trained model and validation metrics.

train_XGB_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.

Class for training time series forecasting models.

2.5 Testing Module

class model_testing.**ModelTest**(*model_type*, *model*, *test*, *target_column*, *forecast_type*, *steps_ahead*)

Bases: object

A class for testing and visualizing the predictions of various types of forecasting models.

Parameters

- **model_type** – The type of model to test ('ARIMA', 'SARIMAX', etc.).
- **model** – The model object to be tested.
- **test** – The test set.
- **target_column** – The target column in the dataset.
- **forecast_type** – The type of forecasting to be performed ('ol-one', etc.).
- **steps_ahead** – Number of forecasting steps to perform.

ARIMA_plot_pred(*best_order*, *predictions*, *naive_predictions=None*)

Plots the ARIMA model predictions against the test data and optionally against naive predictions.

Parameters

- **best_order** – The order of the ARIMA model used.
- **predictions** – The predictions made by the ARIMA model.
- **naive_predictions** – Optional naive predictions for comparison.

SARIMAX_plot_pred(*best_order*, *naive_predictions=None*)

Plots the SARIMAX model predictions against the test data and optionally against naive predictions.

Parameters

- **best_order** – The order of the SARIMAX model used.
- **naive_predictions** – Optional naive predictions for comparison.

naive_forecast(*train*)

Performs a naive forecast using the last observed value from the training set.

Parameters
train – The training set.

Returns
A pandas Series of naive forecasts.

naive_seasonal_forecast(*train*, *target_test*, *period=24*)

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

Parameters

- **train** – The training set.
- **target_test** – The test set.
- **period** – The seasonal period to consider for the forecast.

Returns
A pandas Series of naive seasonal forecasts.

plot_pred(*test*, *predictions*, *time_values*)

Plots model predictions against the test data.

Parameters

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

test_ARIMA_model(*steps_jump=None*, *ol_refit=False*)

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

Parameters

- **steps_jump** – Optional parameter to skip steps in the forecasting.
- **ol_refit** – Boolean indicating whether to refit the model after each forecast.

Returns
A pandas Series of the predictions.

test_SARIMAX_model(*steps_jump=None*, *exog_test=None*, *ol_refit=False*)

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

Parameters

- **steps_jump** – Optional parameter to skip steps in the forecasting.
- **exog_test** – Optional exogenous variables for the test set.
- **ol_refit** – Boolean indicating whether to refit the model after each forecast.

Returns
A pandas Series of the predictions.

Class for testing models.

2.6 Performance Measurement

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

Bases: ModelTest

Class that extends ModelTest to provide additional methods for measuring and plotting performance metrics of forecasting models.

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.

Class for measuring the performance of forecasting models.

2.7 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.**moving_average_ST**(*dataframe*, *target_column*)

Decomposes a time series into its seasonal, trend, and residual components using moving averages.

Parameters

- **dataframe** – DataFrame containing the time series.
- **target_column** – The target column in the DataFrame to decompose.

Returns
The decomposed series with seasonal, trend, and residual attributes.

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.**time_s_analysis**(*df*, *target_column*, *seasonal_period*)

Functions for time series analysis, statistic tests and optimizing statistical models.

2.8 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*, *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** – 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.

Functions for loading or saving models and training data.

CHAPTER

THREE

APPENDIX

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

3.1 Parser Arguments

The command-line parser manages the framework settings, allowing different implementations and uses of the models. It provides many options to customize time series analysis or training and testing of models.

3.1.1 Parameters

–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.

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

–date_list
Provides start and end dates for training, validation, and test sets. Accepts multiple values.

–seasonal_split
If `True`, adjusts the data split to account for seasonality. Default: `False`.

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

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

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

–scaling
If `True`, scales the data. This parameter is required.

–validation
If `True`, includes validation set creation in the data preparation process (not applicable for ARIMA-SARIMAX models). Default: `False`.

–target_column
Specifies the column name to be used as the target variable for forecasting. This parameter is required.

–time_column_index
Specifies the index of the timestamp column in the dataset. Default: `0`.

–model_type
Specifies the type of model to be used. Options include 'ARIMA', 'SARIMAX', 'PROPHET', 'CONV', 'LSTM', 'CNN_LSTM'. This parameter is required.

–forecast_type
Specifies the type of forecast; options are 'ol-multi', 'ol-one', 'cl-multi'. Not necessary for 'PROPHET'.

–steps_ahead
Defines the number of time steps to forecast ahead. Default: `10`.

–steps_jump
Specifies the number of time steps to skip. Default: `50`.

–exog
Specifies exogenous columns for the SARIMAX model. Accepts multiple values.

–period
Defines the seasonality period for the SARIMAX model. Default: `24`.

–seasonal_model
If `True`, performs a seasonal decomposition, and the seasonal component is fed into the LSTM model.

–model_path
Specifies the path of the pre-trained model for fine-tuning. Default: `None`.

–ol_refit
For ARIMA and SARIMAX models, if specified, the model is retrained for each added observation in open-loop forecasts. Default: `False`.

3.1.2 Usage Example

To run the application in training mode with an ARIMA model with validation, use: ``python main_code.py –run_mode training –dataset_path '/path/to/dataset.csv' –target_column 'open' –model_type "ARIMA" –scaling –validation ``

CHAPTER

FOUR

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

FIVE

INDICES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

R

S

T

U