Intelligent System
Assignment 1 - Option B
Week 7 Report

Xuan Tuan Minh Nguyen - 103819212

October 20, 2024

Contents

1 Environment Setup

1.1 Create a new environment using Conda CLI

There are several different procedures to create an environment, but the given procedure below will encapsulate all of the required steps to create a safe and clean environment using Conda [1].

- Navigate to the Github repository that contains the source code for v0.5.
- Once navigated, download the source code by clicking on Code → Download ZIP or use the following command in the CLI (Terminal):

```
git clone https://github.com/cobeo2004/cos30018.git
```

- Once the source code is successfully cloned (downloaded), navigate to the Week 6/v0.5 folder and execute the file conda-config.sh using the following command:

```
bash conda-config.sh
```

- The given file config.sh will execute the following procedure:
  - Generate an environment with a pre-defined name (you can change the name if you want to) in Python 3.10.9 by using the command:

    ```
    conda create -n cos30018_env_w7_v0.5 python=3.10.9
    ```

  - Activate the created environment using:

    ```
    conda activate cos30018_env_w7_v0.5
    ```

  - Check and validate if the conda environment is successfully initialized by running `conda info --envs` for listing conda environments and see which environment that we are in and current Python version using `python --version`.

1.2 Installing required dependencies

Once the environment is successfully initialized, we can start installing the dependencies (libraries) that are required by the program. There are multiple pathways to install dependencies in Python, but the most popular steps are:

- Scan through the code to find out the required dependencies; for example, consider the file stock_prediction.py. We could see that there are quite a few required dependencies, such as: numpy matplotlib pandas tensorflow scikit-learn pandas-datareader yfinance TA-lib statsmodels, plotly and pmdarima. Especially, **statsmodels** will be used for finding parameters for ARIMA model, **plotly** for plotting and **pmdarima** for constructing ARIMA model.
- Once dependencies are scanned, use the following command to install the dependencies:

```
pip install numpy matplotlib pandas tensorflow scikit-learn
pandas-datareader yfinance TA-lib statsmodels plotly pmdarima
```

- Another step is to list all required libraries into a requirements.txt file, and using the following command to install the required dependencies:

```
pip install -U -r requirements.txt
```

Figure 1: Example of requirements.txt

2 Understanding the Machine Learning 3

2.1 Exploring the required parameters for the ARIMA model

2.1.1 Codebase

Figure 2: Codebase for function to display the plots related to ARIMA parameters

2.1.2 Parameters

- **data: pandas.DataFrame**: The processed data that is used for finding the autoregressive order, difference order and moving average order.

2.1.3 Functionalities

In general, the given code will be used for plotting three graphs that correlate to three different essential parameters that the ARIMA model used, which are the number of lag observations in the model (p), the differencing degree to make the time series stationary (d), and the moving average order (q).

- To observe the number of lags in the model (p), we will use the partial autocorrelation plot that is taken from **statsmodels** library and look for any crosses for the upper confidence interval, which is illustrated in the code and the plot below:

Figure 3: Codebase used for plotting PACF

After a short investigation, we could see that first two lags have a significant correlation, which stays at 1 while the others stay around 0. Thus, a suggested p value that we obtained is 2.

To observe the moving average order (q), we will use the autocorrelation function plot that is also acquired from **statsmodels** library and look for any crosses for the upper confidence interval, which is illustrated in the code and the plot below:

The ACF plot points out that there is a significant autocorrelation at the first lag of the plot, thus suggests that the moving average order (q) is 1.

To observe the differencing degree (d), we will inspect the original time series of the stock price and its first and second time series differences.

The given code above creates three subplots that sequently correlates to:

- The original time series of the stock price.
- The first difference of the stock price's time series.

Figure 4: PACF Results of TSLA stock from 2015-01-01 and 2023-08-25

Figure 5: Codebase used for plotting ACF

- The second difference of the stock price's time series.

Based on the given three plots, we could see that:

- The original time series observed a clear upward trend, especially from 2020 onwards. This trend indicates that the series is non-stationary, which suggests that differencing is necessary to achieve stationarity.
- The first-order differencing of the original data appears much more stable compared to the original series plot. The mean seems to be roughly constant around zero, and the variance looks more consistent throughout the time period. This is a significant improvement in terms of stationarity.
- The second-order differencing doesn't seem to be a substantial improvement over the first-order differencing. Thus, might be introducing unnecessary complexity or noise into the data.

Therefore, first-order differencing (d=1) is sufficient to achieve stationarity in this time series.

2.2 Function to create ARIMA model

2.2.1 Codebase

2.2.2 Parameters

- **train_data: pd.DataFrame**: The training dataset that contains historical stock price data.
- **test_data: pd.DataFrame**: The testing dataset that contains historical stock price data for evaluation.
- **p_range: Tuple[int, int]**: The range of p (number of lag) values to consider in the ARIMA model. Default is (0, 5).
- **d_range: Tuple[int, int]**: The range of d (differencing degree) values to consider in the ARIMA model. Default is (0, 2).
- **q_range: Tuple[int, int]**: The range of q (moving average) values to consider in the ARIMA model. Default is (0, 5).
- **m: int**: The number of periods in each season for seasonal ARIMA. Default is 7.
- **seasonal: bool**: Whether to include seasonal components in the ARIMA model. Default is True.

2.2.3 Functionalities

In general, the major purpose of this function is to use the power of Auto Regressive Integrated Moving Average (ARIMA) model to predict the price of the stock based on the close price. The function performs the following steps:

Figure 6: ACF Results of TSLA stock from 2015-01-01 and 2023-08-25

Figure 7: Codebase used for plotting time series

- **Data Extraction**: The function starts by extracting the value of the 'Close' price from both pre-processed train and test data, taken from the implemented `create_dataset()` function.
- **Modelling and training the ARIMA**: Once the data has been processed, the function will call to the `auto_arima()` function from the module `pmdarima` that helps creating the arima model with all of the required parameters and help selecting which model is the best parameters for the context. Specifically, the `auto_arima()` function will consider the specified ranges for the required p, d, and q parameters, and seasonal components if required. Then, the function will perform trainings with different configurations to find out which model fits the best.
- **Model Summary**: Displays general informations of the selected ARIMA model, including the parameters and fit statistics.
- **Prediction**: Then, the function will perform some prediction based on the length of the test data to cover all of the periods on the test data.
- **Calculate Root Mean Square Error (RMSE)**: Once the prediction is finished, the function will calculate the Root Mean Square Error (RMSE) to indicate the error between the predicted values and the actual test data.
- **Return**: Finally, the function wil return an array that contains the predicted stock prices for the test period and the Root Mean Square Error of the predictions compared to the actual test data.

2.3 Function to calculate the ensemble predictions

2.3.1 Codebase

2.3.2 Parameters

- **dl_pred: np.ndarray**: The prediction results array from the deep learning model, could be from LSTM, GRU or RNN.
- **arima_pred: np.ndarray**: The prediction results array from the ARIMA model.
- **dl_rmse: float**: Root Mean Square Error (RMSE) value from the deep learning model predictions.
- **arima_rmse: float**: Root Mean Square Error (RMSE) value from the ARIMA model predictions.
- **test_data: pd.DataFrame — np.ndarray**: The actual test data for comparison and error calculation.

2.3.3 Functionalities

The purpose of this function is to create an ensemble prediction by combining the predictions from a deep learning model with an ARIMA model. The function performs the following procedures:

Figure 8: Time series differences of TSLA stock from 2015-01-01 and 2023-08-25

Figure 9: Codebase for function to create ARIMA model

- **Data Preparation**: Firstly, the function will flatten the deep learning predictions array if needed and starts modifying the length of predictions and test data to ensure they have the same size as different size of data could result in conflictions when working with charts and further calculations.
- **Performing ensemble prediction**: It then converts the deep learning predictions and the ARIMA predictions into a `numpy.array`, then sum all values from both arrays and divide it by 2 to get a unified ensemble prediction array.
- **Data cleaning**: Then, to ensure that the data is cleaned to perform a smooth plotting experience and for scalability, we search for NaN values in the test data and replace it with 0. Once the NaN values are replaced, the function prints the data after replacing NaN and if there are any NaN values in the deep learning and ARIMA prediction arrays.
- **Error Calculation**: Once the data is cleaned, the function will calculate the Root Mean Square Error (RMSE) for the ensemble predictions array and the average RMSE from both deep learning model predictions and the ARIMA predictions.
- **Return data**: Finally, the function will return ensemble predictions result as an `numpy.ndarray`, the Root Mean Square Error (RMSE) value for the ensemble predictions and the average RMSE value for both deep learning predictions and the ARIMA predictions.

3 Deploying and Testing the Codebase

3.1 Codebase used for testing

Please note that, since the given image below that shows the codebase for testing is too long, please refer to the file `main.py` for the full testing code.

References

[1] Md Sabbirul Haque, Md Shahedul Amin, Jonayet Miah, Duc Minh Cao, and Ashiqul Haque Ahmed. Boosting stock price prediction with anticipated macro policy changes. *arXiv*, October 2023.

Figure 10: Codebase for calculate the ensemble predictions

Figure 11: Codebase for testing