Intelligent System

Assignment 1 - Option B

Week 7 Report

Xuan Tuan Minh Nguyen - 103819212

Environment Setup

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.

- 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
```

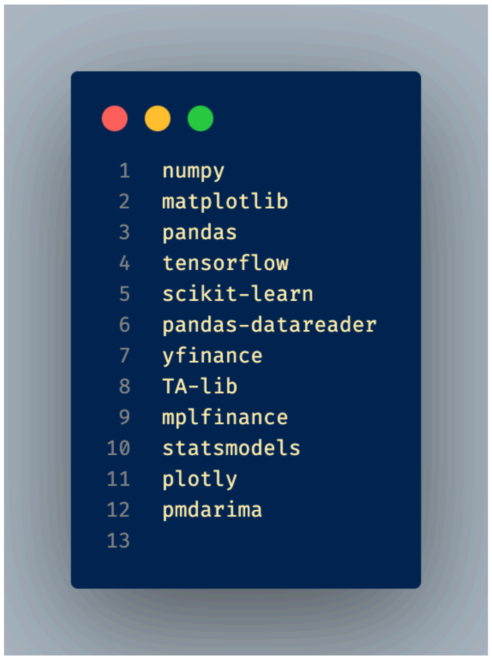
- The given file config.sh will execute the following procedure:
  - Generate an environment with a pre-defined name (you can change the name if you want to) in Python 3.10.9 by using the command: **conda create -n cos30018_env_w7_v0.5 python=3.10.9**
  - Activate the created environment using: **conda 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.**

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 nnumpy 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.



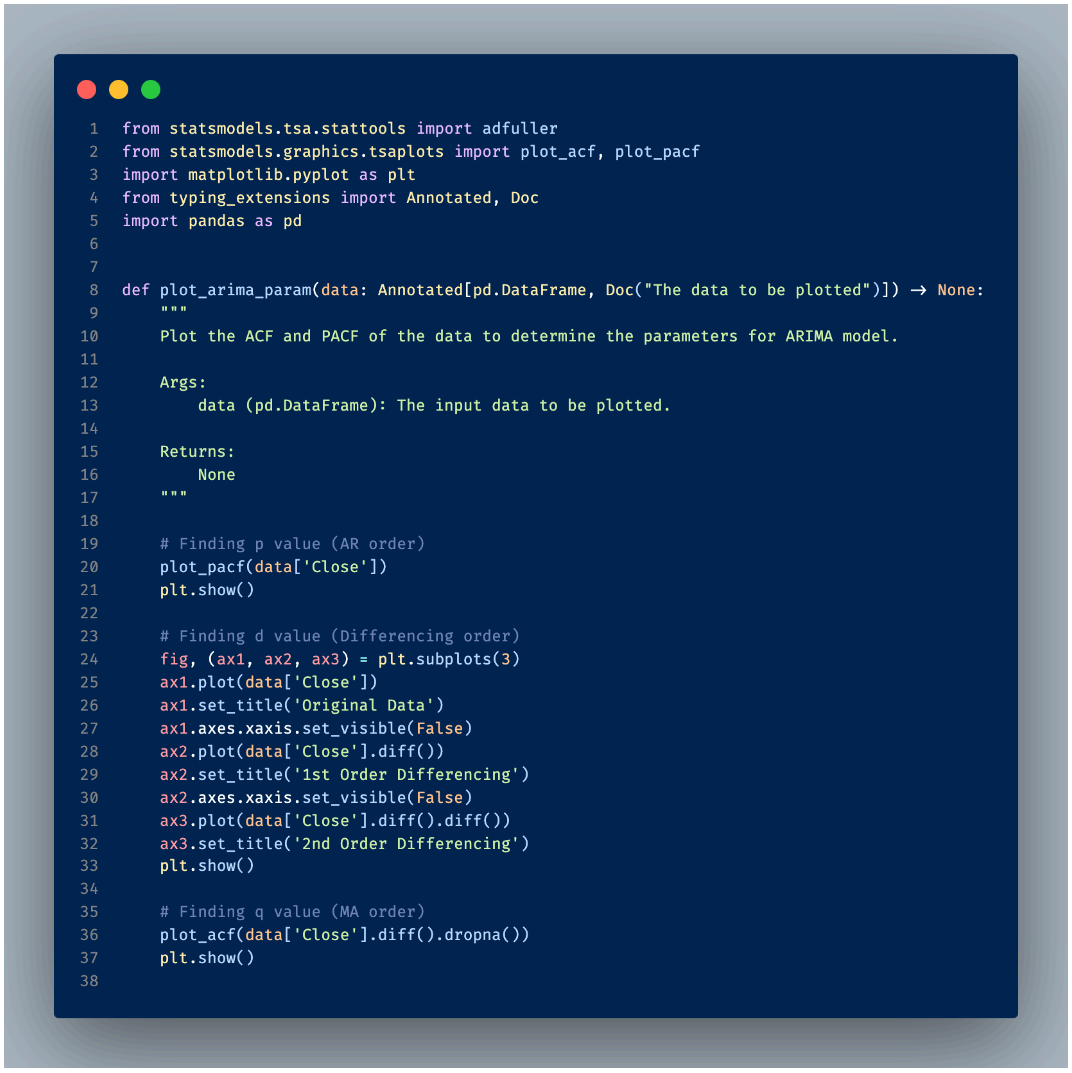
```
numpy
matplotlib
pandas
tensorflow
scikit-learn
pandas-datareader
yfinance
TA-lib
mplfinance
statsmodels
plotly
pmdarima

```

Figure 2: Example of requirements.txt

Understanding the Machine Learning 3

1. Exploring the required parameters for the ARIMA model Codebase



```python
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import matplotlib.pyplot as plt
from typing_extensions import Annotated, Doc
import pandas as pd


def plot_arima_param(data: Annotated[pd.DataFrame, Doc("The data to be plotted")]) → None:
    """
    Plot the ACF and PACF of the data to determine the parameters for ARIMA model.

    Args:
        data (pd.DataFrame): The input data to be plotted.

    Returns:
        None
    """

    # Finding p value (AR order)
    plot_pacf(data['Close'])
    plt.show()

    # Finding d value (Differencing order)
    fig, (ax1, ax2, ax3) = plt.subplots(3)
    ax1.plot(data['Close'])
    ax1.set_title('Original Data')
    ax1.axes.xaxis.set_visible(False)
    ax2.plot(data['Close'].diff())
    ax2.set_title('1st Order Differencing')
    ax2.axes.xaxis.set_visible(False)
    ax3.plot(data['Close'].diff().diff())
    ax3.set_title('2nd Order Differencing')
    plt.show()

    # Finding q value (MA order)
    plot_acf(data['Close'].diff().dropna())
    plt.show()

```

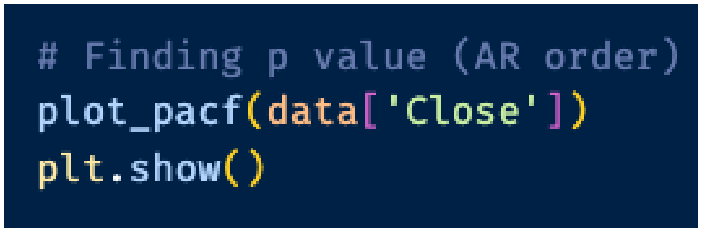
Figure 3: Codebase for function to display the plots related to ARIMA parameters

Parameters

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

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:



```python
# Finding p value (AR order)
plot_pacf(data['Close'])
plt.show()
```

Figure 4: Codebase used for plotting PACF

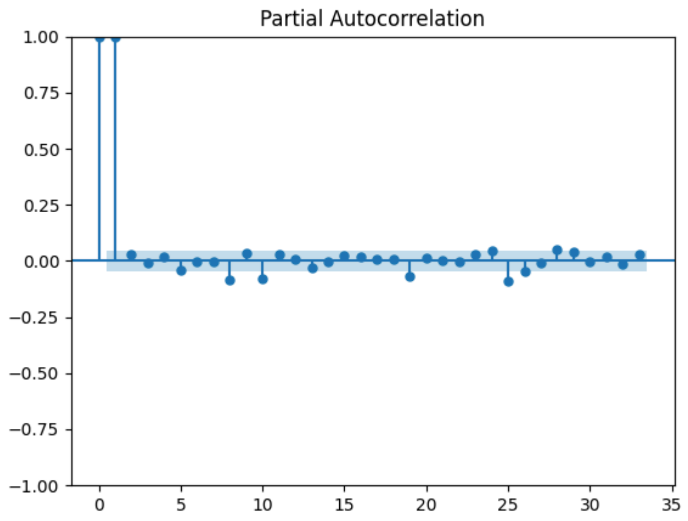
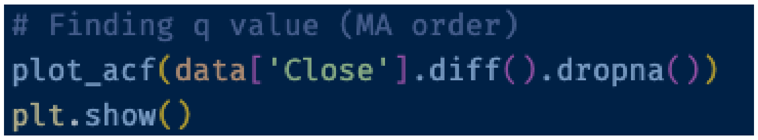


Figure 5: PACF Results of TSLA stock from 2015-01-01 and 2023-08-25

+ 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:



```python
# Finding q value (MA order)
plot_acf(data['Close'].diff().dropna())
plt.show()
```

Figure 6: Codebase used for plotting ACF

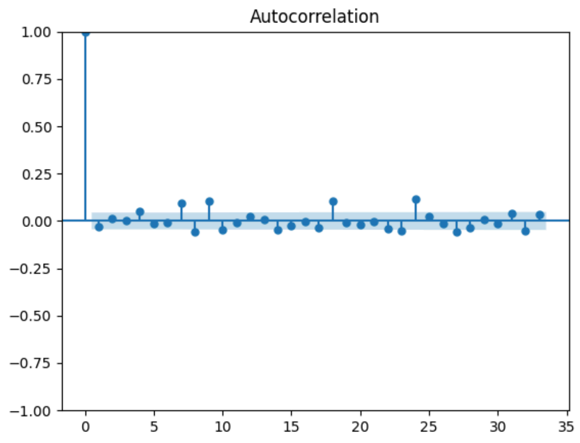
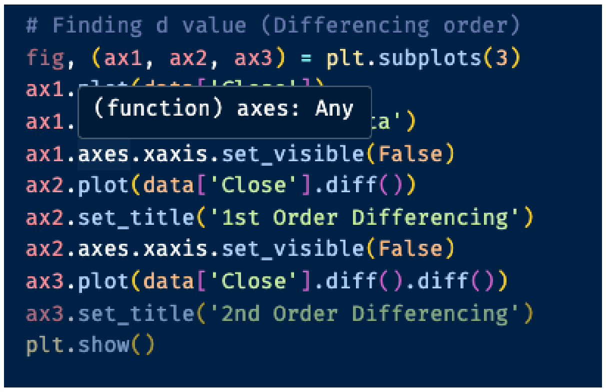


Figure 7: ACF Results of TSLA stock from 2015-01-01 and 2023-08-25

+ 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.



```
# Finding d value (Differencing order)
fig, (ax1, ax2, ax3) = plt.subplots(3)
ax1.plot(data['Close'])
ax1. (function) axes: Any a')
ax1.axes.xaxis.set_visible(False)
ax2.plot(data['Close'].diff())
ax2.set_title('1st Order Differencing')
ax2.axes.xaxis.set_visible(False)
ax3.plot(data['Close'].diff().diff())
ax3.set_title('2nd Order Differencing')
plt.show()
```

Figure 8: Codebase used for plotting time series

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

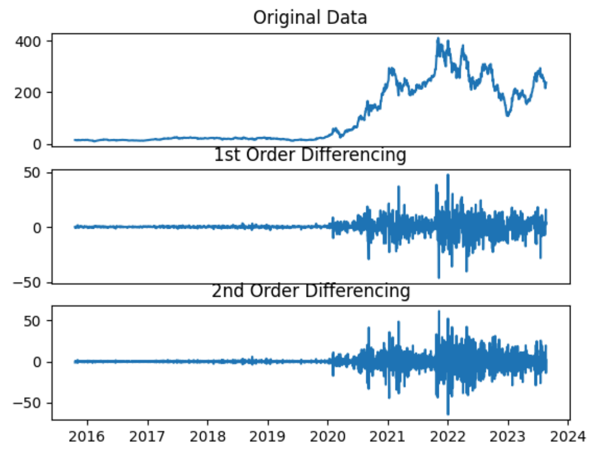
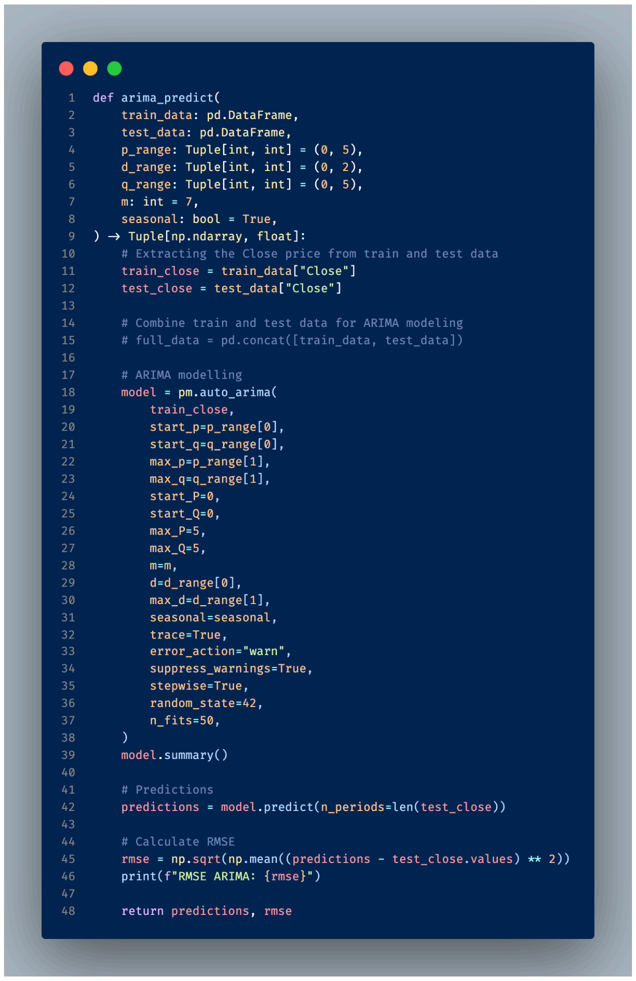


Figure 9: Time series differences of TSLA stock from 2015-01-01 and 2023-08-25

- 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. Function to create ARIMA model

Codebase



```python
def arima_predict(
    train_data: pd.DataFrame,
    test_data: pd.DataFrame,
    p_range: Tuple[int, int] = (0, 5),
    d_range: Tuple[int, int] = (0, 2),
    q_range: Tuple[int, int] = (0, 5),
    m: int = 7,
    seasonal: bool = True,
) → Tuple[np.ndarray, float]:
    # Extracting the Close price from train and test data
    train_close = train_data["Close"]
    test_close = test_data["Close"]

    # Combine train and test data for ARIMA modeling
    # full_data = pd.concat([train_data, test_data])

    # ARIMA modelling
    model = pm.auto_arima(
        train_close,
        start_p=p_range[0],
        start_q=q_range[0],
        max_p=p_range[1],
        max_q=q_range[1],
        start_P=0,
        start_Q=0,
        max_P=5,
        max_Q=5,
        m=m,
        d=d_range[0],
        max_d=d_range[1],
        seasonal=seasonal,
        trace=True,
        error_action="warn",
        suppress_warnings=True,
        stepwise=True,
        random_state=42,
        n_fits=50,
    )
    model.summary()

    # Predictions
    predictions = model.predict(n_periods=len(test_close))

    # Calculate RMSE
    rmse = np.sqrt(np.mean((predictions - test_close.values) ** 2))
    print(f"RMSE ARIMA: {rmse}")

    return predictions, rmse
```

Figure 10: Codebase for function to create ARIMA model

Parameters

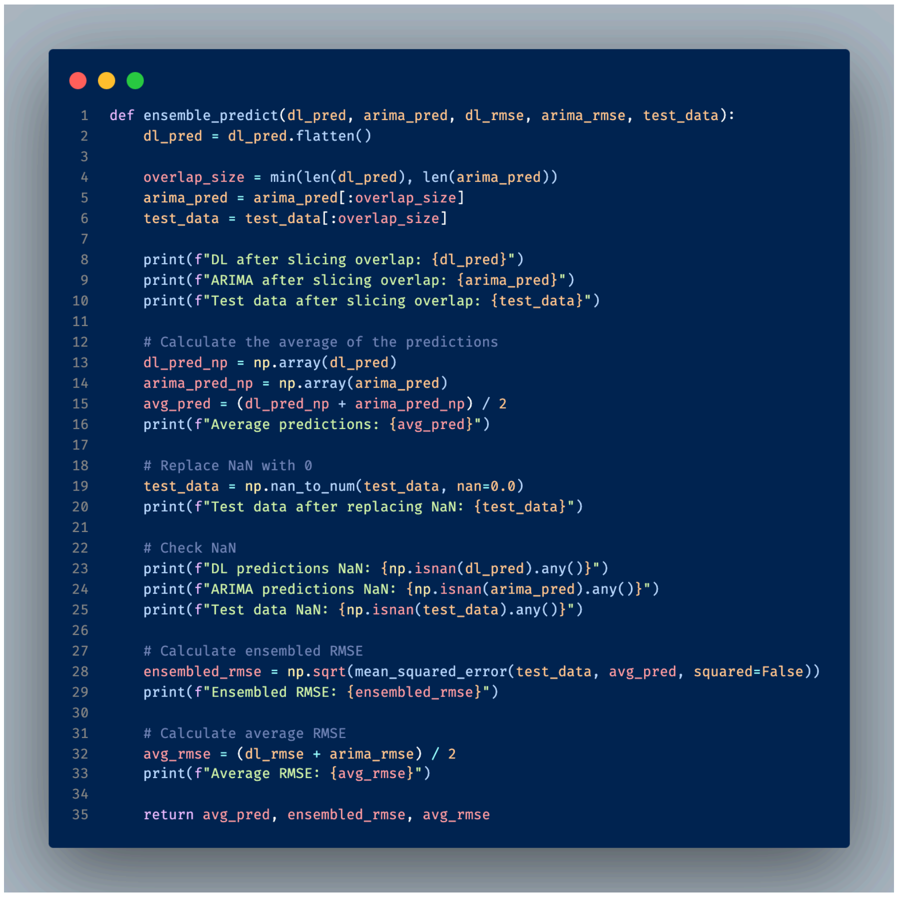
- `train_data: pd.DataFrame`: The training dataset that contains historical stock price data.
- `test_data: pd.DataFrame`: The testing dataset that contains historical stock price data for evaluation.
- `p_range: Tuple[int, int]`: The range of p (number of lag) values to consider in the ARIMA model. 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.

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:
    + **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.

3. Function to calculate the ensemble predictions

Codebase



```python
def ensemble_predict(dl_pred, arima_pred, dl_rmse, arima_rmse, test_data):
    dl_pred = dl_pred.flatten()

    overlap_size = min(len(dl_pred), len(arima_pred))
    arima_pred = arima_pred[:overlap_size]
    test_data = test_data[:overlap_size]

    print(f"DL after slicing overlap: {dl_pred}")
    print(f"ARIMA after slicing overlap: {arima_pred}")
    print(f"Test data after slicing overlap: {test_data}")

    # Calculate the average of the predictions
    dl_pred_np = np.array(dl_pred)
    arima_pred_np = np.array(arima_pred)
    avg_pred = (dl_pred_np + arima_pred_np) / 2
    print(f"Average predictions: {avg_pred}")

    # Replace NaN with 0
    test_data = np.nan_to_num(test_data, nan=0.0)
    print(f"Test data after replacing NaN: {test_data}")

    # Check NaN
    print(f"DL predictions NaN: {np.isnan(dl_pred).any()}")
    print(f"ARIMA predictions NaN: {np.isnan(arima_pred).any()}")
    print(f"Test data NaN: {np.isnan(test_data).any()}")

    # Calculate ensembled RMSE
    ensembled_rmse = np.sqrt(mean_squared_error(test_data, avg_pred, squared=False))
    print(f"Ensembled RMSE: {ensembled_rmse}")

    # Calculate average RMSE
    avg_rmse = (dl_rmse + arima_rmse) / 2
    print(f"Average RMSE: {avg_rmse}")

    return avg_pred, ensembled_rmse, avg_rmse
```

Figure 11: Codebase for calculate the ensemble predictions

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 fromthe ARIMA model predictions.
- `test_data: pd.DataFrame | np.ndarray`: The actual test data for comparison and error calculation.

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:
    + **Data Preparation**: Firstly, the function will flatten the deep learning predictions array if needed and starts modifying the length of

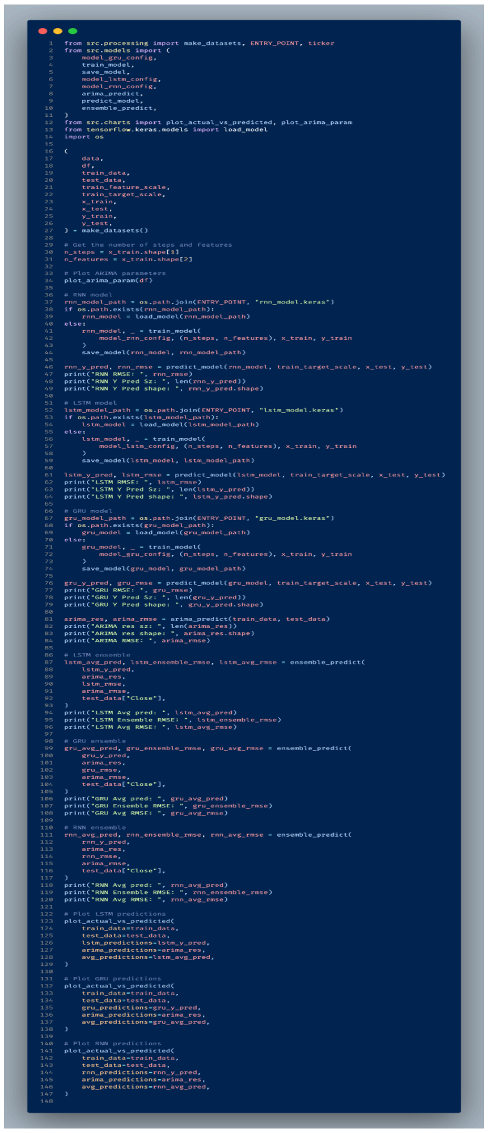
predictions and test data to ensure they have the same size as different size of data could result in conflictions when working with charts and further calculations.

+ **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.

Deploying and Testing the Codebase

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.



```python
from src.processing import make_datasets, ENTRY_POINT, ticker
from src.models import (
    model_gru_config,
    train_model,
    save_model,
    model_lstm_config,
    model_rnn_config,
    arima_predict,
    predict_model,
    ensemble_predict,
)
from src.charts import plot_actual_vs_predicted, plot_arima_param
from tensorflow.keras.models import load_model
import os

(
    data,
    df,
    train_data,
    test_data,
    train_feature_scale,
    train_target_scale,
    x_train,
    x_test,
    y_train,
    y_test,
) = make_datasets()

# Get the number of steps and features
n_steps = x_train.shape[1]
n_features = x_train.shape[2]

# Plot ARIMA parameters
plot_arima_param(df)

# RNN model
rnn_model_path = os.path.join(ENTRY_POINT, "rnn_model.keras")
if os.path.exists(rnn_model_path):
    rnn_model = load_model(rnn_model_path)
else:
    rnn_model, _ = train_model(
        model_rnn_config, (n_steps, n_features), x_train, y_train
    )
    save_model(rnn_model, rnn_model_path)

rnn_y_pred, rnn_rmse = predict_model(rnn_model, train_target_scale, x_test, y_test)
print("RNN RMSE: ", rnn_rmse)
print("RNN Y Pred Sz: ", len(rnn_y_pred))
print("RNN Y Pred shape: ", rnn_y_pred.shape)

# LSTM model
lstm_model_path = os.path.join(ENTRY_POINT, "lstm_model.keras")
if os.path.exists(lstm_model_path):
    lstm_model = load_model(lstm_model_path)
else:
    lstm_model, _ = train_model(
        model_lstm_config, (n_steps, n_features), x_train, y_train
    )
    save_model(lstm_model, lstm_model_path)

lstm_y_pred, lstm_rmse = predict_model(lstm_model, train_target_scale, x_test, y_test)
print("LSTM RMSE: ", lstm_rmse)
print("LSTM Y Pred Sz: ", len(lstm_y_pred))
print("LSTM Y Pred shape: ", lstm_y_pred.shape)

# GRU model
gru_model_path = os.path.join(ENTRY_POINT, "gru_model.keras")
if os.path.exists(gru_model_path):
    gru_model = load_model(gru_model_path)
else:
    gru_model, _ = train_model(
        model_gru_config, (n_steps, n_features), x_train, y_train
    )
    save_model(gru_model, gru_model_path)

gru_y_pred, gru_rmse = predict_model(gru_model, train_target_scale, x_test, y_test)
print("GRU RMSE: ", gru_rmse)
print("GRU Y Pred Sz: ", len(gru_y_pred))
print("GRU Y Pred shape: ", gru_y_pred.shape)

arima_res, arima_rmse = arima_predict(train_data, test_data)
print("ARIMA res sz: ", len(arima_res))
print("ARIMA res shape: ", arima_res.shape)
print("ARIMA RMSE: ", arima_rmse)

# LSTM ensemble
lstm_avg_pred, lstm_ensemble_rmse, lstm_avg_rmse = ensemble_predict(
    lstm_y_pred,
    arima_res,
    lstm_rmse,
    arima_rmse,
    test_data["Close"],
)
print("LSTM Avg pred: ", lstm_avg_pred)
print("LSTM Ensemble RMSE: ", lstm_ensemble_rmse)
print("LSTM Avg RMSE: ", lstm_avg_rmse)

# GRU ensemble
gru_avg_pred, gru_ensemble_rmse, gru_avg_rmse = ensemble_predict(
    gru_y_pred,
    arima_res,
    gru_rmse,
    arima_rmse,
    test_data["Close"],
)
print("GRU Avg pred: ", gru_avg_pred)
print("GRU Ensemble RMSE: ", gru_ensemble_rmse)
print("GRU Avg RMSE: ", gru_avg_rmse)

# RNN ensemble
rnn_avg_pred, rnn_ensemble_rmse, rnn_avg_rmse = ensemble_predict(
    rnn_y_pred,
    arima_res,
    rnn_rmse,
    arima_rmse,
    test_data["Close"],
)
print("RNN Avg pred: ", rnn_avg_pred)
print("RNN Ensemble RMSE: ", rnn_ensemble_rmse)
print("RNN Avg RMSE: ", rnn_avg_rmse)

# Plot LSTM predictions
plot_actual_vs_predicted(
    train_data=train_data,
    test_data=test_data,
    lstm_predictions=lstm_y_pred,
    arima_predictions=arima_res,
    avg_predictions=lstm_avg_pred,
)

# Plot GRU predictions
plot_actual_vs_predicted(
    train_data=train_data,
    test_data=test_data,
    gru_predictions=gru_y_pred,
    arima_predictions=arima_res,
    avg_predictions=gru_avg_pred,
)

# Plot RNN predictions
plot_actual_vs_predicted(
    train_data=train_data,
    test_data=test_data,
    rnn_predictions=rnn_y_pred,
    arima_predictions=arima_res,
    avg_predictions=rnn_avg_pred,
)

```

Figure 12: Codebase for testing

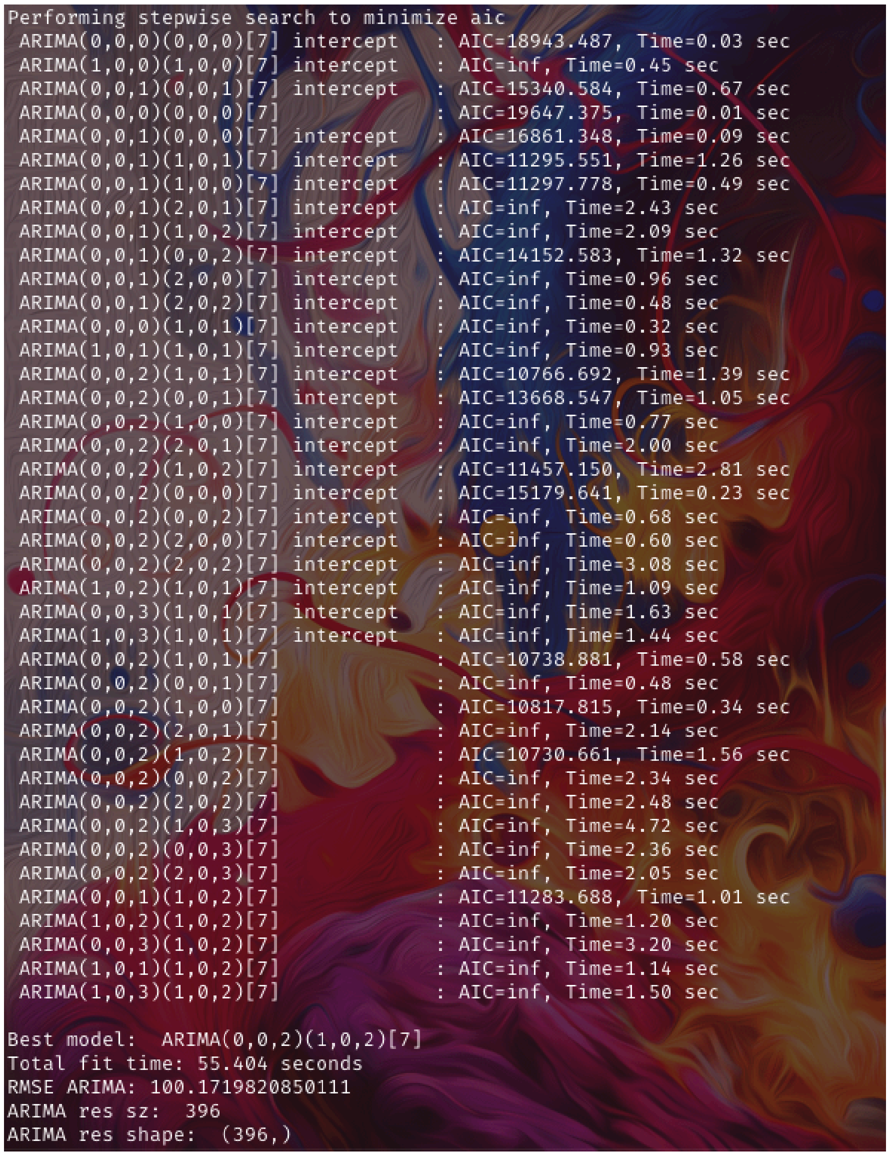
2. *Hyperparameter configurations*

For the ARIMA model, here is the hyperparameter configuration that I have chosen to test the capability of the `auto_arima()` function:

- Autoregressive range (p): From 0 to 5 [0, 5]
- Differencing degree (d): From 0 to 2 [0, 2]
- Moving average term (q): From 0 to 5 [0, 5]
- Number of period for each season: 7
- Is seasonal required?: True

3. *Result*

- Training progress performed by the `auto_arima()` function.

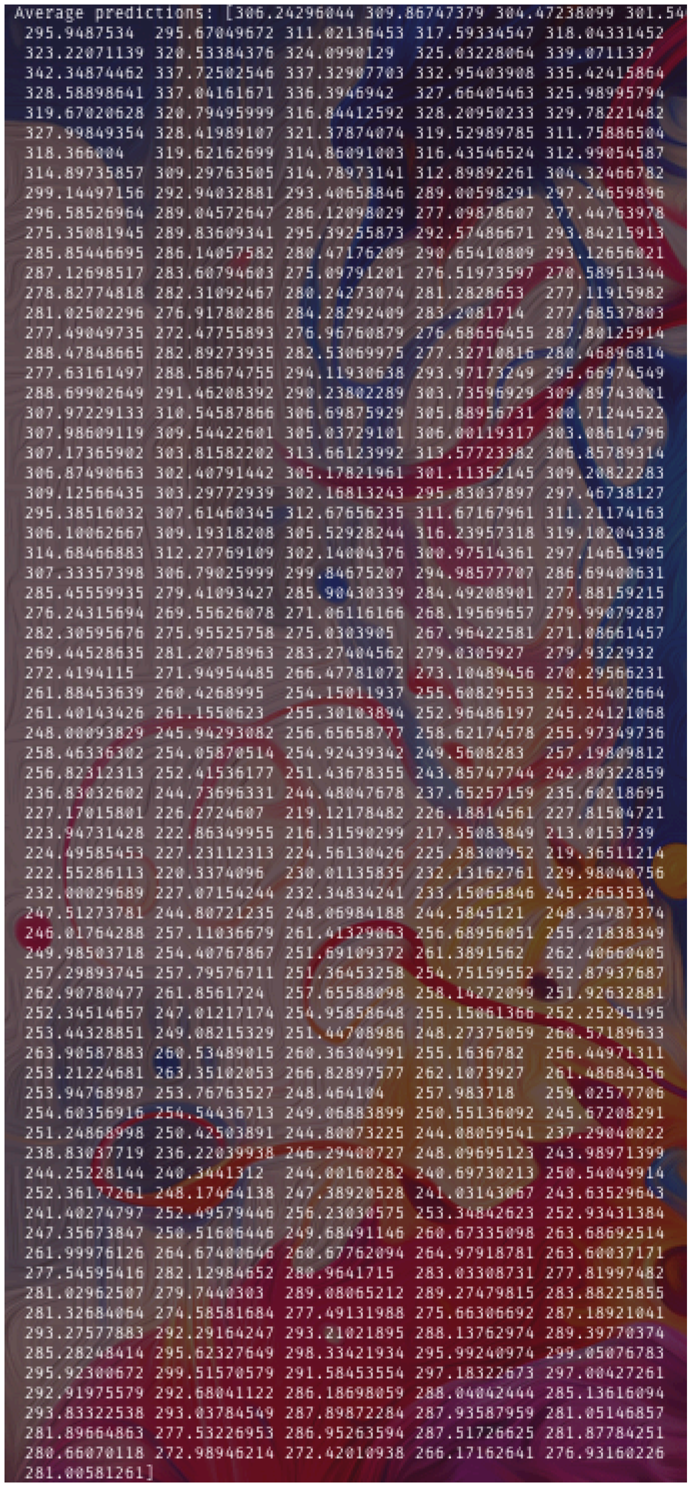


```
Performing stepwise search to minimize aic
 ARIMA(0,0,0)(0,0,0)[7] intercept   : AIC=18943.487, Time=0.03 sec
 ARIMA(1,0,0)(1,0,0)[7] intercept   : AIC=inf, Time=0.45 sec
 ARIMA(0,0,1)(0,0,1)[7] intercept   : AIC=15340.584, Time=0.67 sec
 ARIMA(0,0,0)(0,0,0)[7]             : AIC=19647.375, Time=0.01 sec
 ARIMA(0,0,1)(0,0,0)[7] intercept   : AIC=16861.348, Time=0.09 sec
 ARIMA(0,0,1)(1,0,1)[7] intercept   : AIC=11295.551, Time=1.26 sec
 ARIMA(0,0,1)(1,0,0)[7] intercept   : AIC=11297.778, Time=0.49 sec
 ARIMA(0,0,1)(2,0,1)[7] intercept   : AIC=inf, Time=2.43 sec
 ARIMA(0,0,1)(1,0,2)[7] intercept   : AIC=inf, Time=2.09 sec
 ARIMA(0,0,1)(0,0,2)[7] intercept   : AIC=14152.583, Time=1.32 sec
 ARIMA(0,0,1)(2,0,0)[7] intercept   : AIC=inf, Time=0.96 sec
 ARIMA(0,0,1)(2,0,2)[7] intercept   : AIC=inf, Time=0.48 sec
 ARIMA(0,0,0)(1,0,1)[7] intercept   : AIC=inf, Time=0.32 sec
 ARIMA(1,0,1)(1,0,1)[7] intercept   : AIC=inf, Time=0.93 sec
 ARIMA(0,0,2)(1,0,1)[7] intercept   : AIC=10766.692, Time=1.39 sec
 ARIMA(0,0,2)(0,0,1)[7] intercept   : AIC=13668.547, Time=1.05 sec
 ARIMA(0,0,2)(1,0,0)[7] intercept   : AIC=inf, Time=0.77 sec
 ARIMA(0,0,2)(2,0,1)[7] intercept   : AIC=inf, Time=2.00 sec
 ARIMA(0,0,2)(1,0,2)[7] intercept   : AIC=11457.150, Time=2.81 sec
 ARIMA(0,0,2)(0,0,0)[7] intercept   : AIC=15179.641, Time=0.23 sec
 ARIMA(0,0,2)(0,0,2)[7] intercept   : AIC=inf, Time=0.68 sec
 ARIMA(0,0,2)(2,0,0)[7] intercept   : AIC=inf, Time=0.60 sec
 ARIMA(0,0,2)(2,0,2)[7] intercept   : AIC=inf, Time=3.08 sec
 ARIMA(1,0,2)(1,0,1)[7] intercept   : AIC=inf, Time=1.09 sec
 ARIMA(0,0,3)(1,0,1)[7] intercept   : AIC=inf, Time=1.63 sec
 ARIMA(1,0,3)(1,0,1)[7] intercept   : AIC=inf, Time=1.44 sec
 ARIMA(0,0,2)(1,0,1)[7]             : AIC=10738.881, Time=0.58 sec
 ARIMA(0,0,2)(0,0,1)[7]             : AIC=inf, Time=0.48 sec
 ARIMA(0,0,2)(1,0,0)[7]             : AIC=10817.815, Time=0.34 sec
 ARIMA(0,0,2)(2,0,1)[7]             : AIC=inf, Time=2.14 sec
 ARIMA(0,0,2)(1,0,2)[7]             : AIC=10730.661, Time=1.56 sec
 ARIMA(0,0,2)(0,0,2)[7]             : AIC=inf, Time=2.34 sec
 ARIMA(0,0,2)(2,0,2)[7]             : AIC=inf, Time=2.48 sec
 ARIMA(0,0,2)(1,0,3)[7]             : AIC=inf, Time=4.72 sec
 ARIMA(0,0,2)(0,0,3)[7]             : AIC=inf, Time=2.36 sec
 ARIMA(0,0,2)(2,0,3)[7]             : AIC=inf, Time=2.05 sec
 ARIMA(0,0,1)(1,0,2)[7]             : AIC=11283.688, Time=1.01 sec
 ARIMA(1,0,2)(1,0,2)[7]             : AIC=inf, Time=1.20 sec
 ARIMA(0,0,3)(1,0,2)[7]             : AIC=inf, Time=3.20 sec
 ARIMA(1,0,1)(1,0,2)[7]             : AIC=inf, Time=1.14 sec
 ARIMA(1,0,3)(1,0,2)[7]             : AIC=inf, Time=1.50 sec

Best model:  ARIMA(0,0,2)(1,0,2)[7]
Total fit time: 55.404 seconds
RMSE ARIMA: 100.1719820850111
ARIMA res sz:  396
ARIMA res shape:  (396,)
```

Figure 13: Training process and result for ARIMA model

- Ensemble prediction results for both GRU and ARIMA predictions.



```
Average predictions: [306.24296044 309.86747379 304.47238099 301.54
 295.9487534  295.67049672 311.02136453 317.59334547 318.04331452
 323.22071139 320.53384376 324.0990129  325.03228064 339.0711337
 342.34874462 337.72502546 337.32907703 332.95403908 335.42415864
 328.58898641 337.04161671 336.3946942  327.66405463 325.98995794
 319.67020628 320.79495999 316.84412592 328.20950233 329.78221482
 327.99849354 328.41989107 321.37874074 319.52989785 311.75886504
 318.366004   319.62162699 314.86091003 316.43546524 312.99054587
 314.89735857 309.29763505 314.78973141 312.89892261 304.32466782
 299.14497156 292.94032881 293.40658846 289.00598291 297.24659896
 296.58526964 289.04572647 286.12098029 277.09878607 277.44763978
 275.35081945 289.83609341 295.39255873 292.57486671 293.84215913
 285.85446695 286.14057582 280.47176209 290.65410809 293.12656021
 287.12698517 283.60794603 275.09791201 276.51973597 270.58951344
 278.82774818 282.31092467 280.24273074 281.2828653  277.11915982
 281.02502296 276.91780286 284.28292409 283.2081714  277.68537803
 277.49049735 272.47755893 276.96760879 276.68656455 287.80125914
 288.47848665 282.89273935 282.53069975 277.32710816 280.46896814
 277.63161497 288.58674755 294.11930638 293.97173249 295.66974549
 288.69902649 291.46208392 290.23802289 303.73596929 309.89743001
 307.97229133 310.54587866 306.69875929 305.88956731 300.71244522
 307.98609119 309.54422601 305.03729101 306.00119317 303.08614796
 307.17365902 303.81582202 313.66123992 313.57723382 306.85789314
 306.87490663 302.40791442 305.17821961 301.11352145 309.20822283
 309.12566435 303.29772939 302.16813243 295.83037897 297.46738127
 295.38516032 307.61460345 312.67656235 311.67167961 311.21174163
 306.10062667 309.19318208 305.52928244 316.23957318 319.10204338
 314.68466883 312.27769109 302.14004376 300.97514361 297.14651905
 307.33357398 306.79025999 299.84675207 294.98577707 286.69400631
 285.45559935 279.41093427 285.90430339 284.49208901 277.88159215
 276.24315694 269.55626078 271.06116166 268.19569657 279.99079287
 282.30595676 275.95525758 275.0303905  267.96422581 271.08661457
 269.44528635 281.20758963 283.27404562 279.0305927  279.9322932
 272.4194115  271.94954485 266.47781072 273.10489456 270.29566231
 261.88453639 260.4268995  254.15011937 255.60829553 252.55402664
 261.40143426 261.1550623  255.30103894 252.96486197 245.24121068
 248.00093829 245.94293082 256.65658777 258.62174578 255.97349736
 258.46336302 254.05870514 254.92439342 249.5608283  257.19809812
 256.82312313 252.41536177 251.43678355 243.85747744 242.80322859
 236.83032602 244.73696331 244.48047678 237.65257159 235.60218695
 227.17015801 226.2724607  219.12178482 226.18814561 227.81504721
 223.94731428 222.86349955 216.31590299 217.35083849 213.0153739
 224.49585453 227.23112313 224.56130426 225.38300952 219.36511214
 222.55286113 220.3374096  230.01135835 232.13162761 229.98040756
 232.00029689 227.07154244 232.34834241 233.15065846 245.2653534
 247.51273781 244.80721235 248.06984188 244.5845121  248.34787374
 246.01764288 257.11036679 261.41329063 256.68956051 255.21838349
 249.98503718 254.40767867 251.69109372 261.3891562  262.40660405
 257.29893745 257.79576711 251.36453258 254.75159552 252.87937687
 262.90780477 261.8561724  257.65588098 258.14272099 251.92632881
 252.34514657 247.01217174 254.95858648 255.15061366 252.25295195
 253.44328851 249.08215329 251.44708986 248.27375059 260.57189633
 263.90587883 260.53489015 260.36304991 255.1636782  256.44971311
 253.21224681 263.35102053 266.82897577 262.1073927  261.48684356
 253.94768987 253.76763527 248.464104   257.983718   259.02577706
 254.60356916 254.54436713 249.06883899 250.55136092 245.67208291
 251.24868998 250.42503891 244.80073225 244.08059541 237.29040022
 238.83037719 236.22039938 246.29400727 248.09695123 243.98971399
 244.25228144 240.3441312  244.00160282 240.69730213 250.54049914
 252.36177261 248.17464138 247.38920528 241.03143067 243.63529643
 241.40274797 252.49579446 256.23030575 253.34842623 252.93431384
 247.35673847 250.51606446 249.68491146 260.67335098 263.68692514
 261.99976126 264.67400646 260.67762094 264.97918781 263.60037171
 277.54595416 282.12984652 280.9641715  283.03308731 277.81997482
 281.02962507 279.7440303  289.08065212 289.27479815 283.88225855
 281.32684064 274.58581684 277.49131988 275.66306692 287.18921041
 293.27577883 292.29164247 293.21021895 288.13762974 289.39770374
 285.28248414 295.62327649 298.33421934 295.99240974 299.05076783
 295.92300672 299.51570579 291.58453554 297.18322673 297.00427261
 292.91975579 292.68041122 286.18698059 288.04042444 285.13616094
 293.83322538 293.03784549 287.89872284 287.93587959 281.05146857
 281.89664863 277.53226953 286.95263594 287.51726625 281.87784251
 280.66070118 272.98946214 272.42010938 266.17162641 276.93160226
 281.00581261]
```

Figure 14: Result for the test

- Plots that display the prediction results, including the ensemble prediction results, between ARIMA and GRU, RNN and LSTM models.

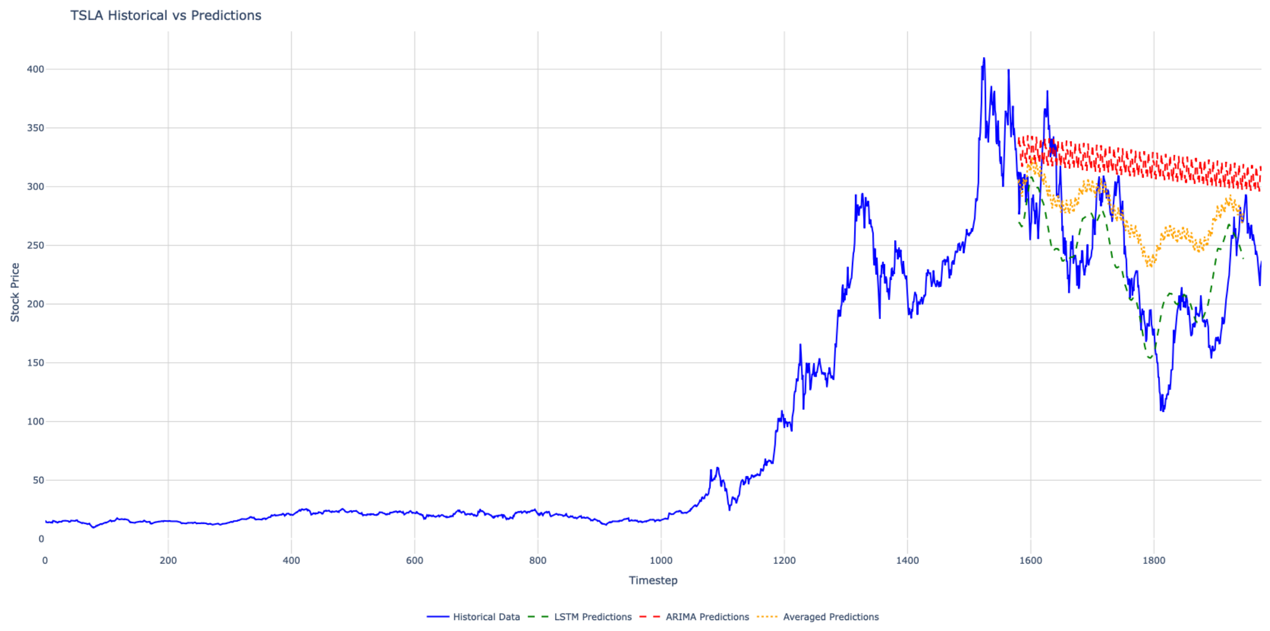


Figure 15: ARIMA and LSTM

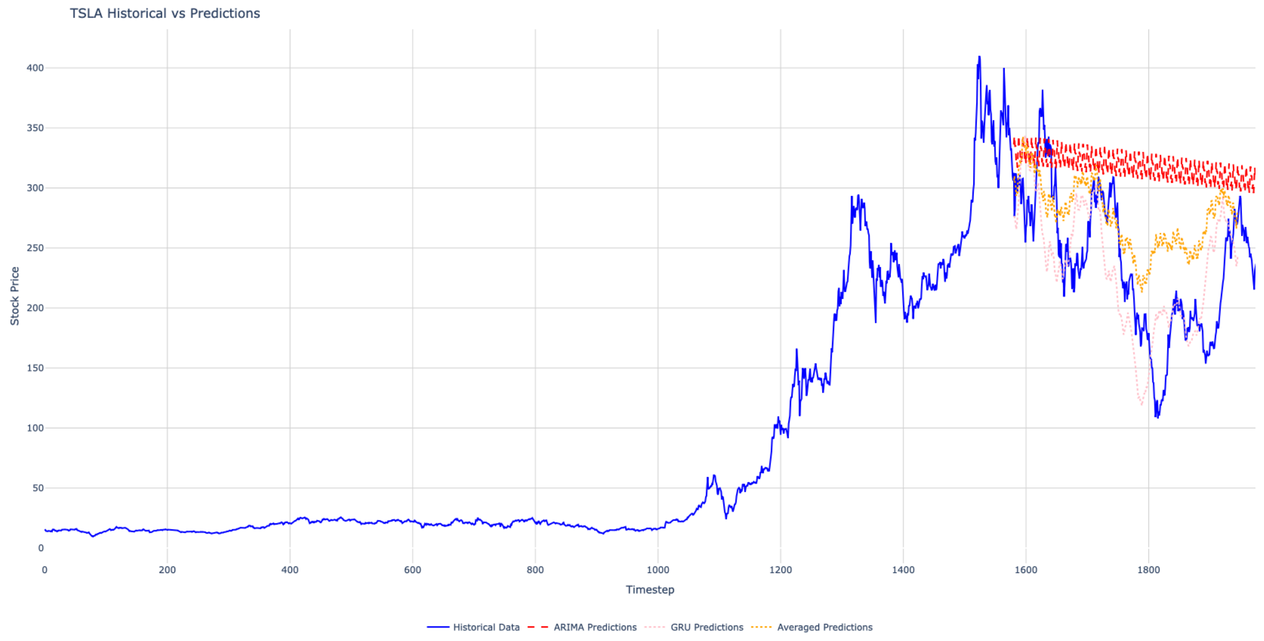


Figure 16: ARIMA and GRU

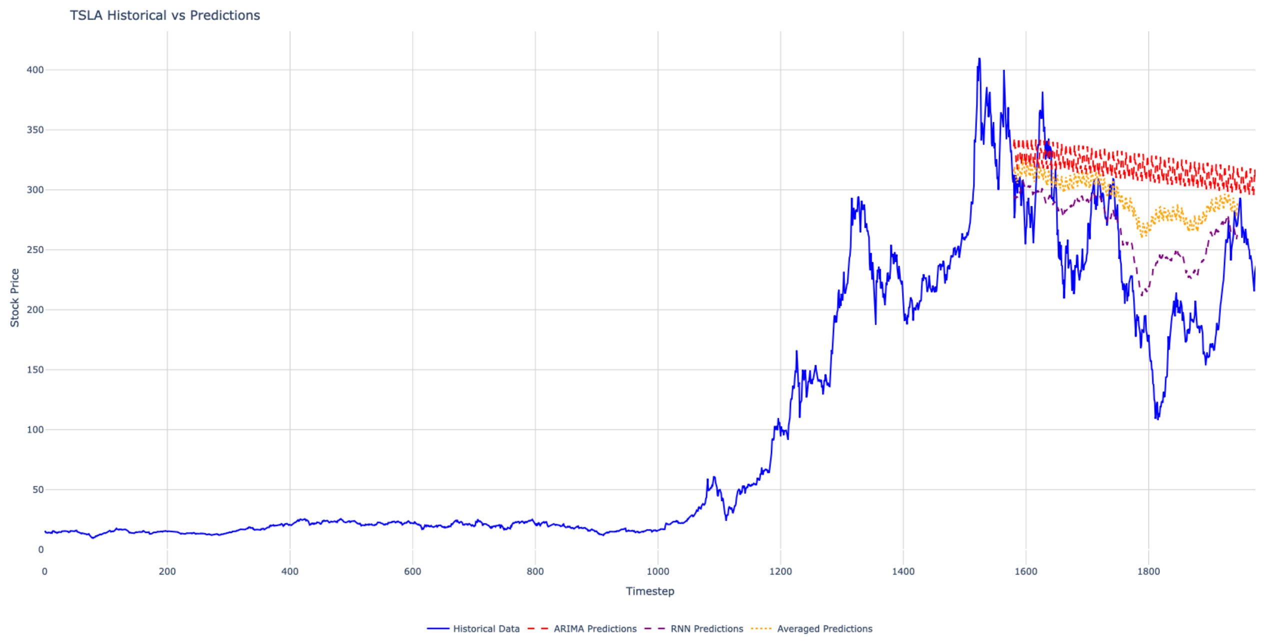


Figure 17: ARIMA and RNN