COS30049 - Intelligent System

MACHINE LEARNING

TRAN DUC ANH DANG

Student ID: 103995439

Requirements

Virtual Environment

- Google Colab
- Jupyter Notebook

Decencies

- numpy
- matplotlib
- mplfinance
- pandas
- scikit-learn
- pandas-datareader
- yfinance
- pandas_ta
- joblib

Installation

*Note: Anaconda is required unless Google Collab is being used

Anaconda/Virtual Environment

1. Download Anaconda: Go to the Anaconda website (https://www.anaconda.com/products/distribution) and download the appropriate version for your operating system.
2. Install Anaconda: Follow the installation instructions for the OS from the Anaconda website.
3. Open Anaconda Navigator: Launch Anaconda Navigator from your installed applications.
4. Create a New Environment (Required): Create a new environment to isolate Jupyter installation on each project. Click on "Environments" in Navigator and then "Create" to make a new environment.
5. Install Jupyter Notebook: In the selected environment, click on the environment name and select "Open Terminal". In the terminal, type: conda install jupyter.

Dependencies

In Google Colab or Jupyter Notebook, it can directly install the required dependencies using the !pip command in code cells. Here's an example of how to install the dependencies:

!pip install <package> or !pip install -r <text file>

Machine Learning 3

arima_predict_test

predict.py

```python
def arima_predict_test(train_data, test_data, start_p=0, max_p=5, start_q=0, max_q=5, m=7, seasonal=True):
    # Extracting the Close price from train and test data
    train_close = train_data['Close']
    test_close = test_data['Close']

    x_train = list(range(len(train_close)))
    x_test = list(range(len(train_close), len(train_close) + len(test_close)))

    # Visualization of training and testing data
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x_train, y=train_close, mode='lines+markers', marker=dict(size=4),  name='train', marker_color='#39304A'))
    fig.add_trace(go.Scatter(x=x_test, y=test_close, mode='lines+markers', marker=dict(size=4), name='test', marker_color='#A98D75'))
    fig.update_layout(legend_orientation="h",
                  legend=dict(x=.5, xanchor="center"),
                  plot_bgcolor='#FFFFFF',
                  xaxis=dict(gridcolor = 'lightgrey'),
                  yaxis=dict(gridcolor = 'lightgrey'),
                  title_text = f'{ticker} ARIMA data', title_x = 0.5,
                  xaxis_title="Timestep",
                  yaxis_title="Stock price",
                  margin=dict(l=0, r=0, t=30, b=0))
    fig.show()

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

    # ARIMA modeling
    model = pm.auto_arima(train_close,
                          start_p=start_p,
                          d=None,
                          start_q=start_q,
                          max_p=max_p,
                          max_d=5,
                          max_q=max_q,
                          start_P=0,
                          D=1,
                          start_Q=0,
                          max_P=5,
                          max_D=5,
                          max_Q=5,
                          m=m,
                          seasonal=seasonal,
                          error_action='warn',
                          trace=True,
                          supress_warnings=True,
                          stepwise=True,
                          random_state=20,
                          n_fits=50)

    model.summary()

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

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

    # Visualization of historical vs predicted values
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x_test, y=test_close, mode='lines+markers', name='historical', marker_color='#39304A'))
    fig.add_trace(go.Scatter(x=x_test, y=predictions, mode='lines+markers', name='predictions', marker_color='#FFAA00'))
    fig.update_layout(legend_orientation="h",
                  legend=dict(x=.5, xanchor="center"),
                  plot_bgcolor='#FFFFFF',
                  xaxis=dict(gridcolor = 'lightgrey'),
                  yaxis=dict(gridcolor = 'lightgrey'),
                  title_text = f'{ticker} ARIMA prediction', title_x = 0.5,
                  xaxis_title="Timestep",
                  yaxis_title="Stock price",
                  margin=dict(l=0, r=0, t=30, b=0))
    fig.show()

    return predictions, rmse
```

Snipped

The function `arima_predict_test` takes the following parameters:

1. `train_data`: A DataFrame containing training data which includes stock prices and other relevant information.
2. `test_data`: A DataFrame containing testing data.

3. `start_p`: Starting value of the autoregressive term 'p' (Default: 0).
4. `max_p`: Maximum value of the autoregressive term 'p' (Default: 5).
5. `start_q`: Starting value of the moving average term 'q' (Default: 0).
6. `max_q`: Maximum value of the moving average term 'q' (Default: 5).
7. `m`: Seasonality parameter (Default: 7).
8. seasonal`: A boolean indicating if seasonal terms should be considered (Default: True).

This function performs the following operations:

1. Data Extraction: Extracts the 'Close' price from both training and testing datasets.
2. Data Visualization: Plots the training and testing data for visualization.
3. Data Preparation: Combines the training and testing data for ARIMA modeling.
4. ARIMA Modeling: Constructs an ARIMA model using the `pm.auto_arima` method.
5. Predictions: Uses the trained ARIMA model to predict the 'Close' prices for the test data.
6. RMSE Calculation: Computes the RMSE between the predicted and actual 'Close' prices.
7. Prediction Visualization: Plots the historical (actual) vs. predicted values for the test data.

Return:
- Returns the predicted values (`predictions`) and the RMSE of the predictions.

sarimax_predict_test

predict.py

```python
def sarimax_predict_test(train_data, test_data, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12)):
    # Extracting the Close price and exogenous variables from train and test data
    train_close = train_data['Close']
    test_close = test_data['Close']
    exo_train_data = train_data['Volume']
    exo_test_data = test_data['Volume']

    # Setting frequency for the time series
    train_close.index = pd.DatetimeIndex(train_close.index).to_period('D')
    test_close.index = pd.DatetimeIndex(test_close.index).to_period('D')
    exo_train_data.index = pd.DatetimeIndex(exo_train_data.index).to_period('D')
    exo_test_data.index = pd.DatetimeIndex(exo_test_data.index).to_period('D')

    x_train = list(range(len(train_close)))
    x_test = list(range(len(train_close), len(train_close) + len(test_close)))

    # Visualization of training and testing data
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x_train, y=train_close, mode='lines+markers', marker=dict(size=4),  name='train', marker_color='#39304A'))
    fig.add_trace(go.Scatter(x=x_test, y=test_close, mode='lines+markers', marker=dict(size=4), name='test', marker_color='#A98D75'))
    fig.update_layout(legend_orientation="h",
                  legend=dict(x=.5, xanchor="center"),
                  plot_bgcolor='#FFFFFF',
                  xaxis=dict(gridcolor = 'lightgrey'),
                  yaxis=dict(gridcolor = 'lightgrey'),
                  title_text = f'{ticker} ARIMA data', title_x = 0.5,
                  xaxis_title="Timestep",
                  yaxis_title="Stock price",
                  margin=dict(l=0, r=0, t=30, b=0))
    fig.show()

    # SARIMAX modeling
    model = SARIMAX(train_close, exog=exo_train_data, order=order, seasonal_order=seasonal_order)
    results = model.fit(disp=-1, maxiter=200, method='nm')
    print(results.summary())

    # Predictions
    predictions = results.predict(start=len(train_close),
                                  end=len(train_close) + len(test_close) - 1,
                                  exog=exo_test_data)

    rmse = np.sqrt(mean_squared_error(test_close, predictions))
    print(f'RMSE SARIMAX: {rmse}')

    # Visualization of historical vs predicted values
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x_test, y=test_close, mode='lines+markers', name='historical', marker_color='#39304A'))
    fig.add_trace(go.Scatter(x=x_test, y=predictions, mode='lines+markers', name='predictions', marker_color='#FFAA00'))
    fig.update_layout(legend_orientation="h",
                  legend=dict(x=.5, xanchor="center"),
                  plot_bgcolor='#FFFFFF',
                  xaxis=dict(gridcolor='lightgrey'),
                  yaxis=dict(gridcolor='lightgrey'),
                  title_text=f'{ticker} SARIMAX prediction', title_x=0.5,
                  xaxis_title="Timestep",
                  yaxis_title="Stock price",
                  margin=dict(l=0, r=0, t=30, b=0))
    fig.show()

    return predictions, rmse
```

Snipped

The function `sarimax_predict_test` takes the following parameters:

1. `train_data`: A DataFrame containing training data which includes stock prices and other relevant information.
2. `test_data`: A DataFrame containing testing data.
3. `order`: A tuple representing the ARIMA order (p, d, q) (Default: (1, 1, 1)).
4. `seasonal_order`: A tuple representing the seasonal ARIMA order (P, D, Q, S) (Default: (1, 1, 1, 12)).

This function performs the following operations:

1. Data Extraction: Retrieves the 'Close' price and exogenous variable 'Volume' from both training and testing datasets.
2. Time Series Frequency Setting: Sets the frequency of the time series data to daily.

3. Data Visualization: Plots the training and testing data for visualization.
4. SARIMAX Modeling: Constructs a SARIMAX model using the provided order and seasonal_order parameters.
5. Predictions: Uses the trained SARIMAX model to predict the 'Close' prices for the test data.
6. RMSE Calculation: Computes the RMSE between the predicted and actual 'Close' prices.
7. Prediction Visualization: Plots the historical (actual) vs. predicted values for the test data.

Return:
- Returns the predicted values (`predictions`) and the RMSE of the predictions.

average_prediction

```
def average_predictions(lstm_predictions, arima_predictions, sarimax_predictions, lstm_rmse, arima_rmse, sarimax_
    # Ensure lstm_predictions is a 1D array
    lstm_predictions = lstm_predictions.flatten()

    # Get overlapping period
    print('Get Overlapping Period')
    min_length = min(len(lstm_predictions), len(arima_predictions), len(sarimax_predictions))  # Since LSTM has t
    arima_predictions = arima_predictions[:min_length]
    sarimax_predictions = sarimax_predictions[:min_length]
    test_data = test_data[:min_length]

    print(f'LSTM shape after slicing: {lstm_predictions.shape}')
    print(f'ARIMA shape after slicing: {arima_predictions.shape}')
    print(f'SARIMAX shape after slicing: {sarimax_predictions.shape}')

    # Average predictions
    print('Average Predictions')
    avg_predictions = (lstm_predictions[:, np.newaxis] + arima_predictions[:, np.newaxis] + sarimax_predictions[:
    avg_predictions = avg_predictions.flatten()

    print(f'avg_predictions shape: {avg_predictions.shape}')
    print(f'test_data shape: {test_data.shape}')

    # 0 for NaN in Test
    test_data = np.nan_to_num(test_data)  # This replaces NaN with 0
    print(f'Test Data; {len(test_data)}')

    # Checking for NaN
    print("NaN in LSTM predictions:", np.isnan(lstm_predictions).any())
    print("NaN in ARIMA predictions:", np.isnan(arima_predictions).any())
    print("NaN in SARIMAX predictions:", np.isnan(sarimax_predictions).any())
    print("NaN in test data:", np.isnan(test_data).any())

    # Calculate RMSE for the ensemble predictions
    print('Ensemble RMSE')
    ensemble_rmse = np.sqrt(mean_squared_error(test_data, avg_predictions))

    # Calculate the average RMSE for the three models
    print('Avergage RMSE Model')
    avg_model_rmse = (lstm_rmse + arima_rmse + sarimax_rmse) / 3

    return avg_predictions, ensemble_rmse, avg_model_rmse
```

The function `average_predictions` takes the following parameters:
1. `lstm_predictions`: A NumPy array containing the predicted values from the LSTM model.
2. `arima_predictions`: A NumPy array containing the predicted values from the ARIMA model.

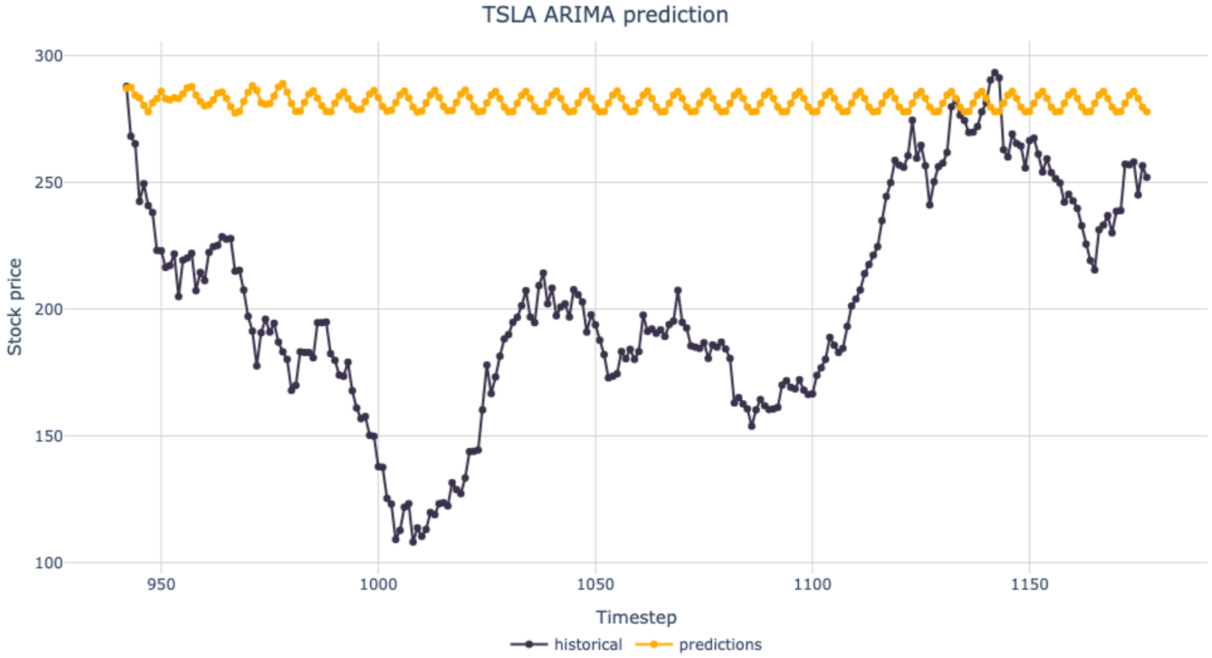
3. `sarimax_predictions`: A NumPy array containing the predicted values from the SARIMAX model.
4. `lstm_rmse`: A float representing the RMSE value of the LSTM predictions.
5. `arima_rmse`: A float representing the RMSE value of the ARIMA predictions.
6. `sarimax_rmse`: A float representing the RMSE value of the SARIMAX predictions.
7. `test_data`: A NumPy array containing the actual test values against which predictions are compared.

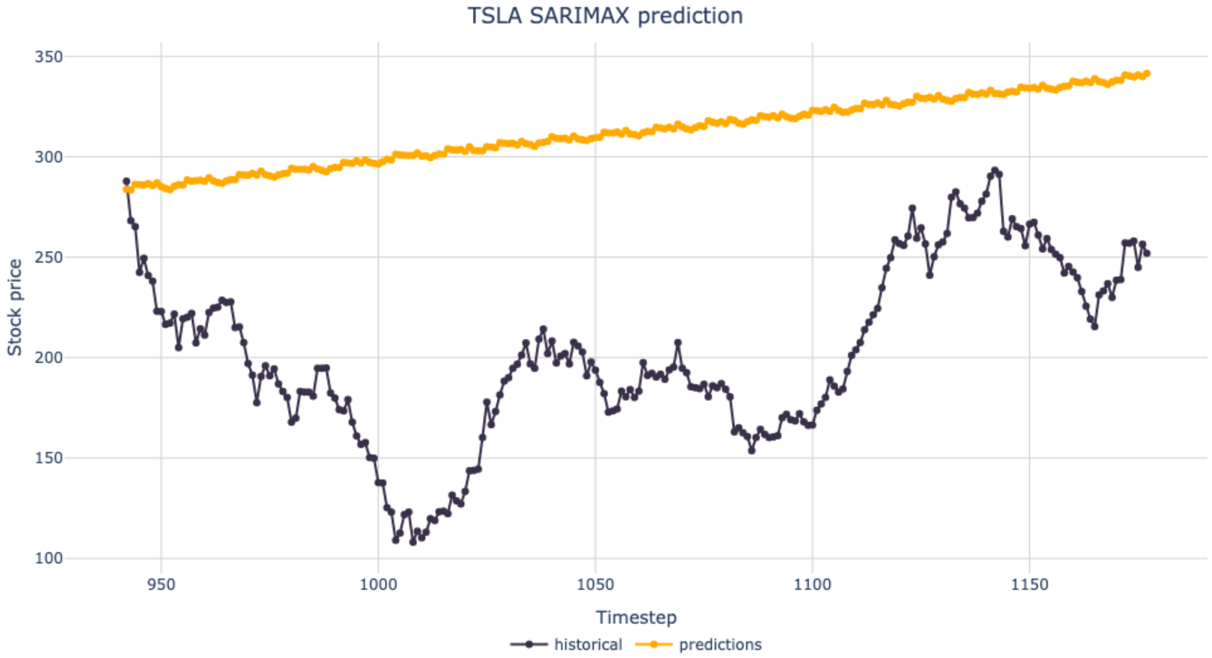
This function performs the following operations:
1. Data Preparation: Ensures that the LSTM predictions are in a flattened 1D array.
2. Overlap Determination: Finds the overlapping period among the LSTM, ARIMA, and SARIMAX predictions.
3. Averaging Predictions: Computes the average of the LSTM, ARIMA, and SARIMAX predictions for each time step.
4. Data Cleaning: Replaces any NaN values in the test data with 0.
5. NaN Check: Verifies if there are any NaN values in the predictions and the test data.
6. RMSE Calculation for Ensemble: Computes the RMSE of the ensemble predictions (average of the three models) against the test data.
7. Average RMSE Calculation: Computes the average RMSE of the individual LSTM, ARIMA, and SARIMAX models.

Return:
- Returns the ensemble predictions (`avg_predictions`), the RMSE of the ensemble predictions (`ensemble_rmse`), and the average RMSE of the three individual models (`avg_model_rmse`).

ARIMA and SARIMAX with Different Hyperameter Configs:

Default Config

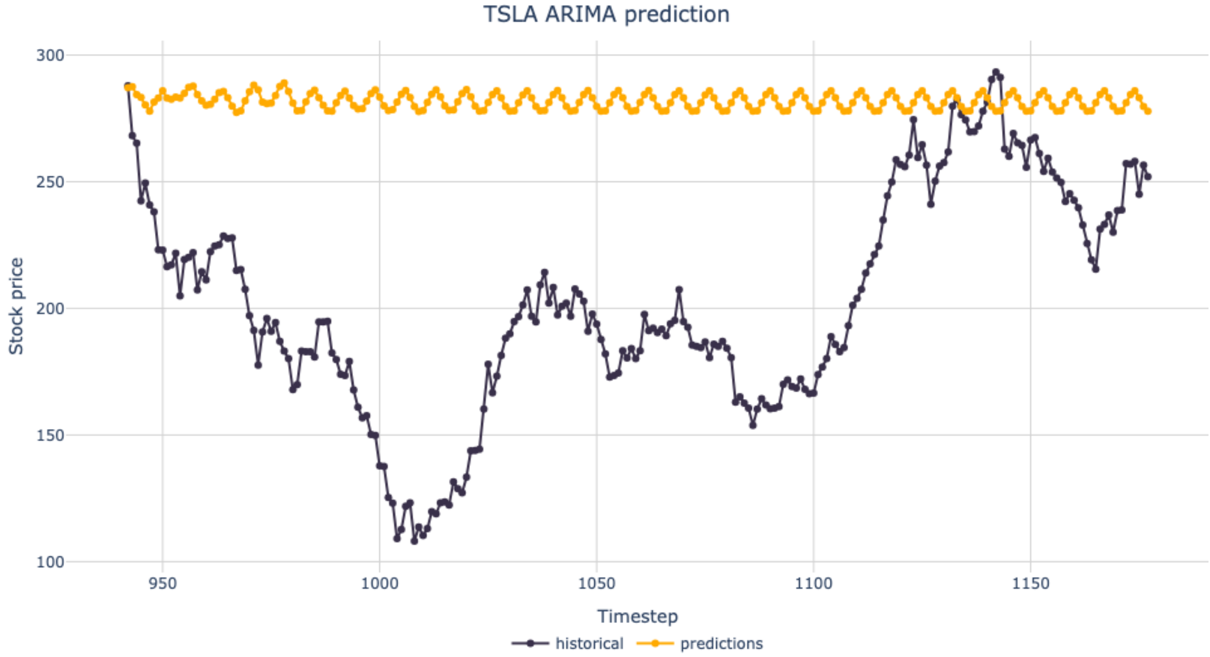




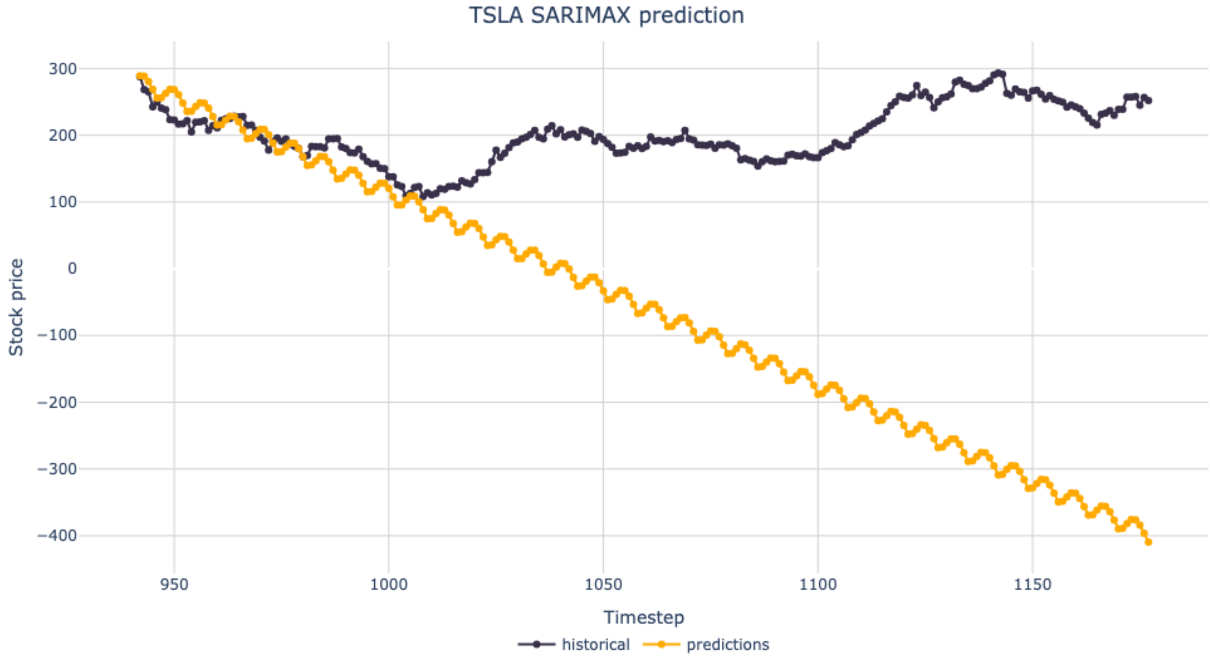

TSLA Historical vs Predictions
Stock Price
Timestep
0
50
100
150
200
250
300
350
400
0
200
400
600
800
1000
Historical Data
LSTM Predictions
ARIMA Predictions
SARIMAX Predictions
Averaged Predictions

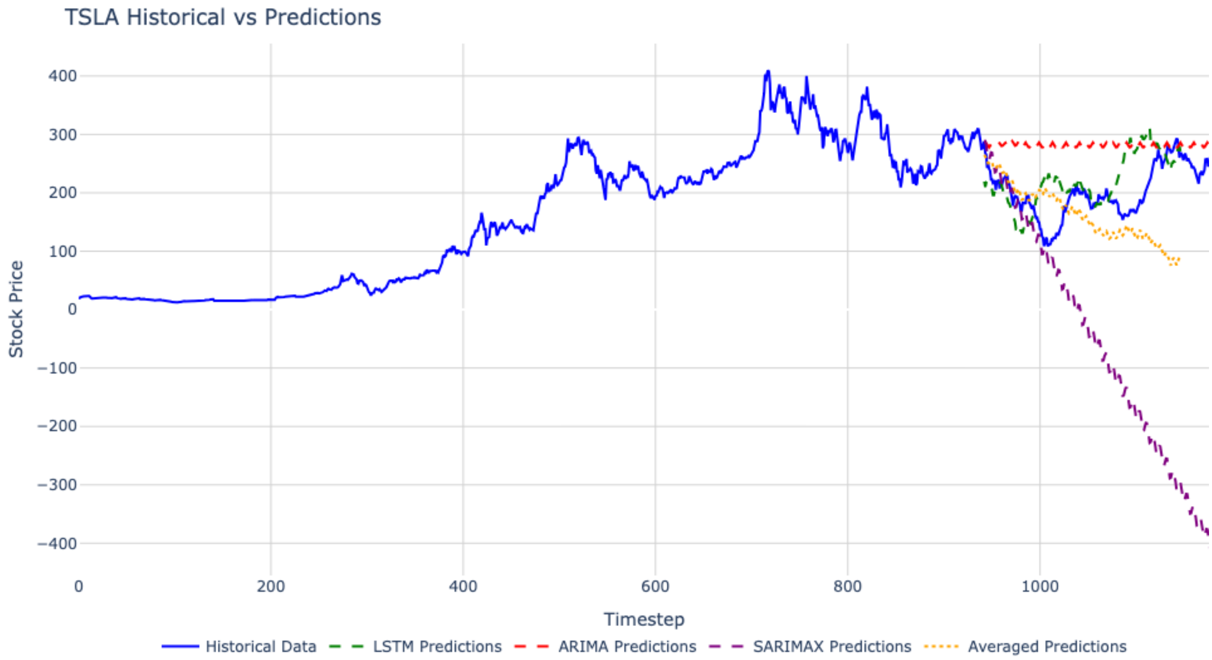
Config 1:

- arima_predictions_1, arima_rmse_1 = arima_predict_test(train_data, test_data, start_p=0, max_p=1, start_q=0, max_q=1)



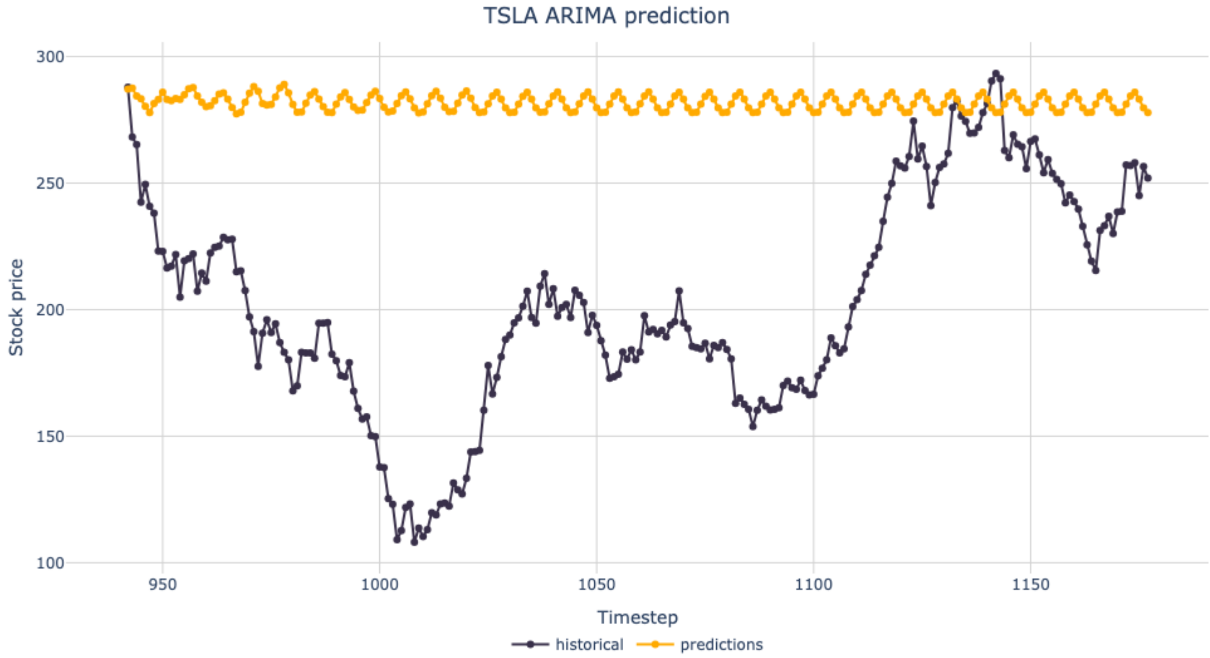
- sarimax_predictions_1, sarimax_rmse_1 = sarimax_predict_test(train_data, test_data, order=(0, 1, 0), seasonal_order=(0, 1, 0, 7))



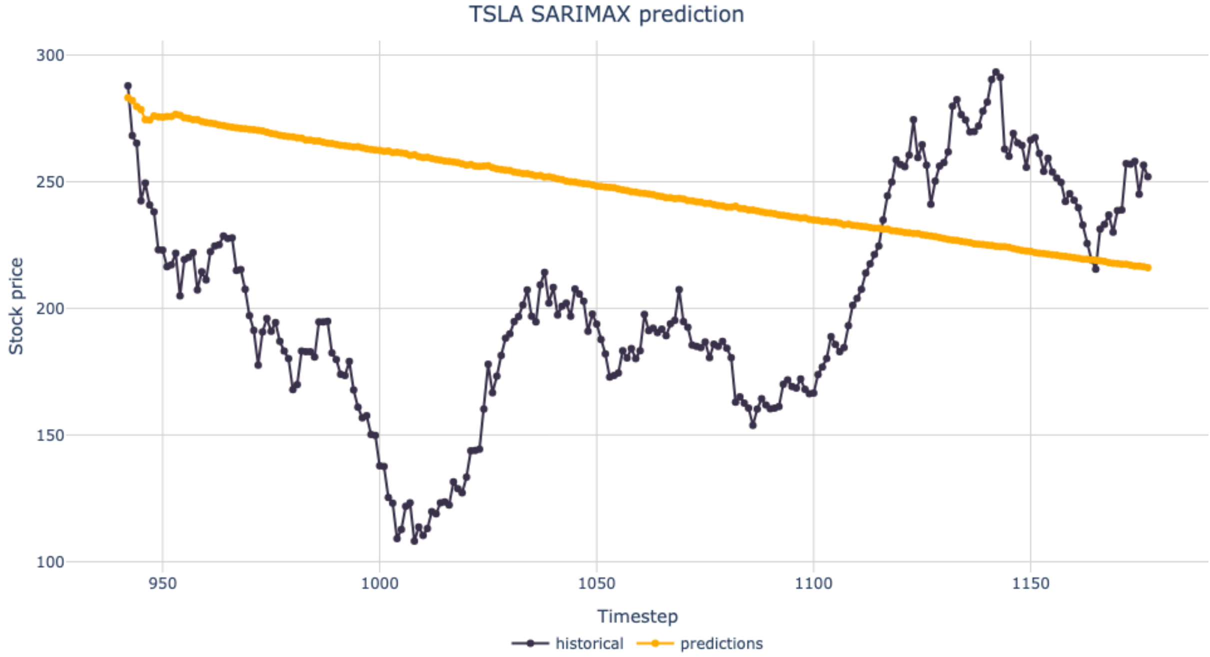


Config 2:

- arima_predictions_2, arima_rmse_2 = arima_predict_test(train_data, test_data, start_p=1, max_p=2, start_q=1, max_q=2)

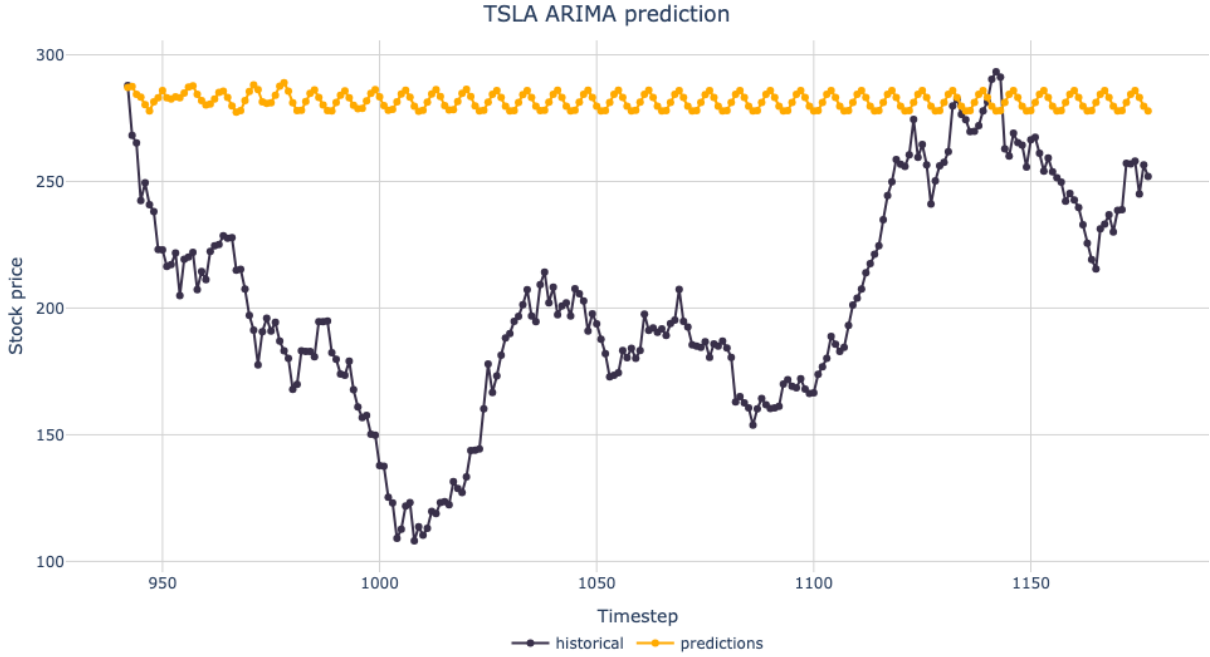


- sarimax_predictions_2, sarimax_rmse_2 = sarimax_predict_test(train_data, test_data, order=(1, 0, 1), seasonal_order=(1, 0, 1, 7))

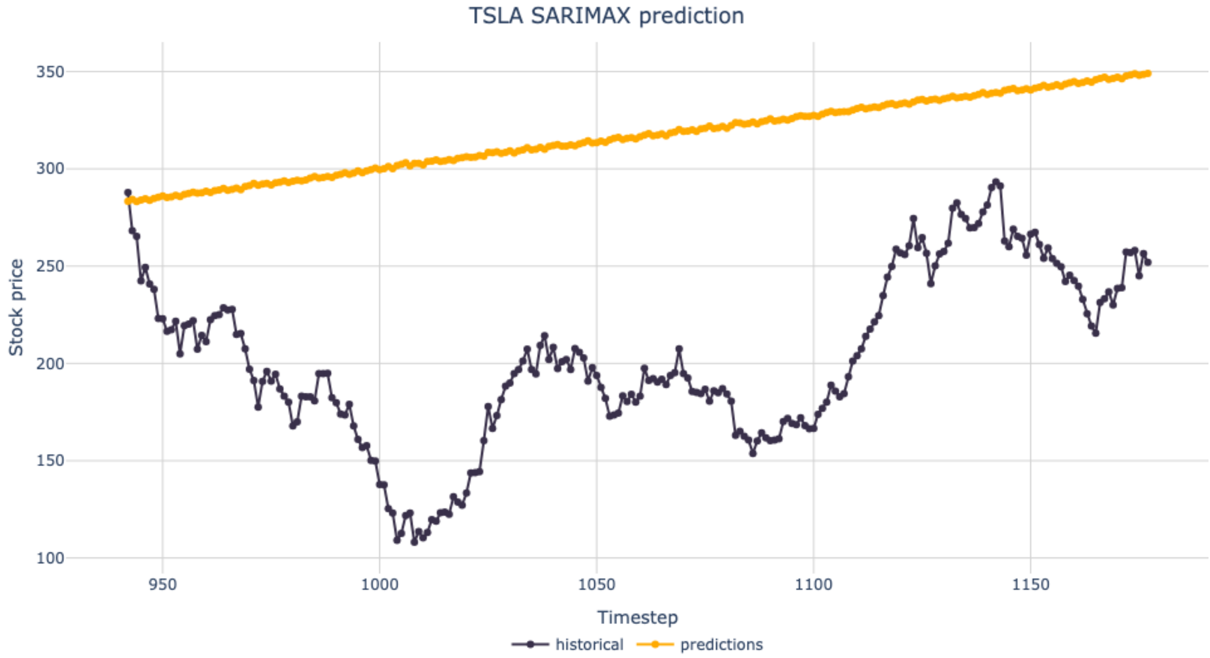


Config 3:

- arima_predictions_3, arima_rmse_3 = arima_predict_test(train_data, test_data, start_p=0, max_p=2, start_q=0, max_q=2)



- sarimax_predictions_3, sarimax_rmse_3 = sarimax_predict_test(train_data, test_data, order=(0, 1, 1), seasonal_order=(0, 1, 1, 7))



Overall, the ARIMA predictions are quite similar to each other. Only on some configs, SARIMAX are fairly accurate, but in the end most of the average price are predicting quite good beside of Config 1.