

COS30018

Intelligent Systems

Option B: Stock Prediction



Task B.2 – Data Processing

Name: Duc Thuan Tran
Student ID: 104330455
Tutor: Dr. Ru Jia
Tutorial: Friday 2:30 – 4:30

Table of Contents

<i>Introduction</i>	3
<i>Methodology</i>	3
<i>Code Analysis</i>	8
Model Execution and Outputs	12
Execution Process	12
GitHub Repository	13
<i>References</i>	13

Introduction

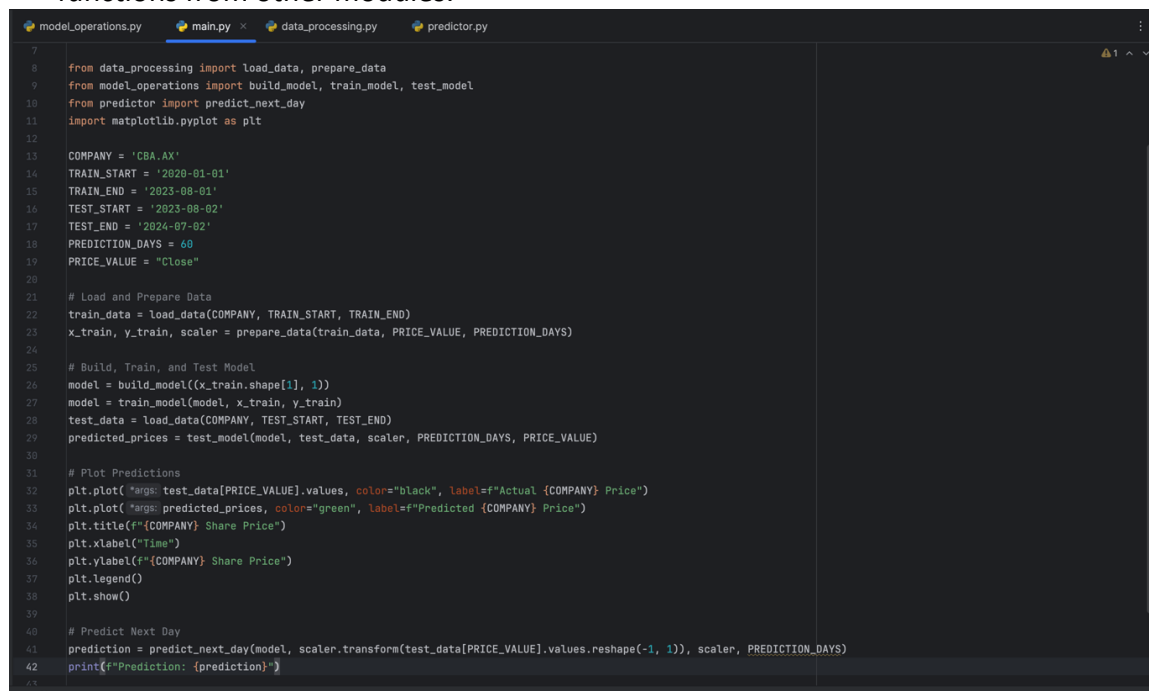
This task focuses on enhancing the v0.1 stock prediction code by addressing its limitations in data processing. The original version only utilized the 'Close' feature and required manual date selection for training and testing datasets. Additionally, it lacked effective handling of NaN values and flexible data splitting options.

To improve this, I developed a more versatile data processing function that supports multiple features, flexible date selection, robust NaN handling, and various data splitting methods. This work builds on the v0.1 code, integrating new functionalities for more accurate and efficient stock market predictions.

Methodology

To enhance the organization and maintainability of the stock prediction project, the original source code was divided into four distinct files: **main.py**, **model_operations.py**, **predictor.py**, and **data_processing.py**. Each file serves a specific purpose in the workflow:

- **main.py**: This file orchestrates the entire stock prediction workflow. It is responsible for integrating all components, including data loading, preprocessing, model building, training, testing, and prediction. It acts as the central script that coordinates the execution of various functions from other modules.



```
7
8 from data_processing import load_data, prepare_data
9 from model_operations import build_model, train_model, test_model
10 from predictor import predict_next_day
11 import matplotlib.pyplot as plt
12
13 COMPANY = 'CBA.AX'
14 TRAIN_START = '2020-01-01'
15 TRAIN_END = '2023-08-01'
16 TEST_START = '2023-08-02'
17 TEST_END = '2024-07-02'
18 PREDICTION_DAYS = 60
19 PRICE_VALUE = "Close"
20
21 # Load and Prepare Data
22 train_data = load_data(COMPANY, TRAIN_START, TRAIN_END)
23 x_train, y_train, scaler = prepare_data(train_data, PRICE_VALUE, PREDICTION_DAYS)
24
25 # Build, Train, and Test Model
26 model = build_model((x_train.shape[1], 1))
27 model = train_model(model, x_train, y_train)
28 test_data = load_data(COMPANY, TEST_START, TEST_END)
29 predicted_prices = test_model(model, test_data, scaler, PREDICTION_DAYS, PRICE_VALUE)
30
31 # Plot Predictions
32 plt.plot(*args: test_data[PRICE_VALUE].values, color="black", label=f"Actual {COMPANY} Price")
33 plt.plot(*args: predicted_prices, color="green", label=f"Predicted {COMPANY} Price")
34 plt.title(f"{COMPANY} Share Price")
35 plt.xlabel("Time")
36 plt.ylabel(f"{COMPANY} Share Price")
37 plt.legend()
38 plt.show()
39
40 # Predict Next Day
41 prediction = predict_next_day(model, scaler.transform(test_data[PRICE_VALUE].values.reshape(-1, 1)), scaler, PREDICTION_DAYS)
42 print(f"Prediction: {prediction}")
43
```

Figure 1. main.py

- **data_processing.py:** This module handles all tasks related to data loading, processing, and preparation. It includes functions for loading stock data, handling NaN values, splitting the data into training and testing sets, scaling features, and caching the data locally. The enhancements made in this file focus on providing flexibility and efficiency in data handling.

```

1  # File: data_processing.py
2  # Purpose: This module handles data loading and preprocessing tasks.
3  # It fetches the stock data from the specified source and prepares it
4  # for model training by scaling and structuring it appropriately.
5
6  import yfinance as yf
7  import numpy as np
8  from sklearn.preprocessing import MinMaxScaler
9
10
11  3 usages
12  def load_data(company, start_date, end_date):
13      return yf.download(company, start_date, end_date)
14
15  2 usages
16  def prepare_data(data, price_value, prediction_days):
17      scaler = MinMaxScaler(feature_range=(0, 1))
18      scaled_data = scaler.fit_transform(data[price_value].values.reshape(-1, 1))
19
20      x_train, y_train = [], []
21      scaled_data = scaled_data[:, 0]
22      for x in range(prediction_days, len(scaled_data)):
23          x_train.append(scaled_data[x - prediction_days:x])
24          y_train.append(scaled_data[x])
25
26      x_train, y_train = np.array(x_train), np.array(y_train)
27      x_train = np.reshape(x_train, (newshape: (x_train.shape[0], x_train.shape[1], 1)))
28
29      return x_train, y_train, scaler

```

Figure 2. data_processing.py

- **model_operations.py:** This file contains the functions responsible for building, training, and testing the predictive model. The architecture of the model, such as the LSTM layers and the training process, is defined here. By separating model-related operations into this module, the code becomes more modular and easier to manage.

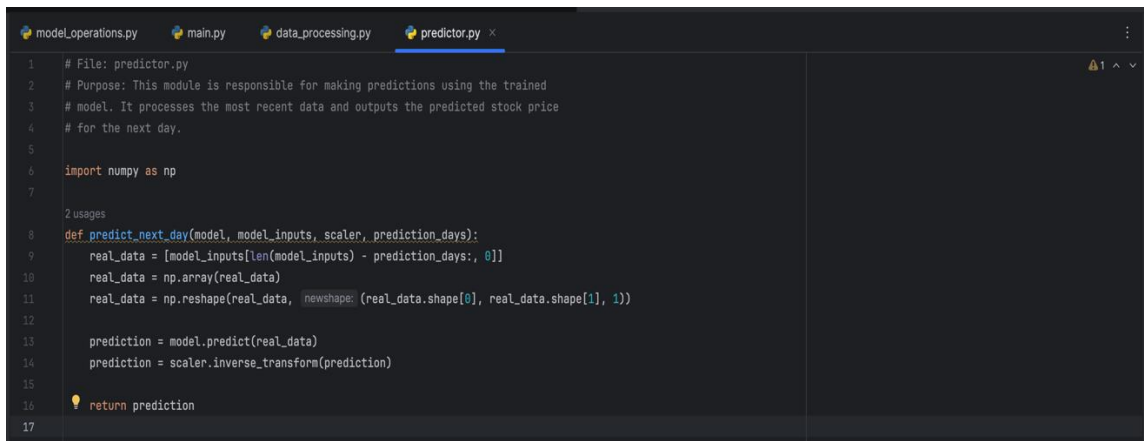
```

1  modelOperations.py x main.py data_processing.py predictor.py
2
3  5
4  from tensorflow.keras.models import Sequential
5  from tensorflow.keras.layers import Dense, Dropout, LSTM
6  import numpy as np
7  import pandas as pd
8
9  2 usages
10 def build_model(input_shape):
11     model = Sequential()
12
13     model.add(LSTM(units=50, return_sequences=True, input_shape=input_shape))
14     model.add(Dropout(0.2))
15     model.add(LSTM(units=50, return_sequences=True))
16     model.add(Dropout(0.2))
17     model.add(LSTM(units=50))
18     model.add(Dropout(0.2))
19     model.add(Dense(units=1))
20
21     model.compile(optimizer='adam', loss='mean_squared_error')
22
23     return model
24
25  2 usages
26 def train_model(model, x_train, y_train, epochs=25, batch_size=32):
27     model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size)
28     return model
29
30  2 usages
31 def test_model(model, data, scaler, prediction_days, price_value):
32     total_dataset = pd.concat([data[price_value]], axis=0)
33     model_inputs = total_dataset[len(total_dataset) - len(data) - prediction_days:].values
34     model_inputs = model_inputs.reshape(-1, 1)
35     model_inputs = scaler.transform(model_inputs)
36
37     v_fact = 1

```

Figure 3. model_operations.py

- **predictor.py:** The predictor.py file is dedicated to making predictions using the trained model. It includes functions for generating predictions based on the most recent data and performing any necessary inverse transformations to return the predictions to their original scale. This separation ensures that the prediction logic is isolated from other parts of the code, making it easier to maintain and update.



```
1 # File: predictor.py
2 # Purpose: This module is responsible for making predictions using the trained
3 # model. It processes the most recent data and outputs the predicted stock price
4 # for the next day.
5
6 import numpy as np
7
8 2 usages
9 def predict_next_day(model, model_inputs, scaler, prediction_days):
10     real_data = [model_inputs[len(model_inputs) - prediction_days:, 0]]
11     real_data = np.array(real_data)
12     real_data = np.reshape(real_data, newshape=(real_data.shape[0], real_data.shape[1], 1))
13
14     prediction = model.predict(real_data)
15     prediction = scaler.inverse_transform(prediction)
16
17     return prediction
```

Figure 4. predictor.py

The following sections outline the approach taken to address the key requirements: data loading and processing, handling NaN values, data splitting, data caching, and feature scaling.

1. Data Loading and Processing

To improve the flexibility of the data processing function, a new function was developed to allow users to specify the start and end dates for the dataset. This function loads stock data for the specified period and ensures that multiple features, such as 'Close' and 'Volume', are utilized, rather than relying solely on the 'Close' price as in the original version. The ability to specify the date range ensures that the model can be trained and tested on any desired period, making the data processing pipeline adaptable to various scenarios.

```
def load_data(company, start_date, end_date, nan_handling='drop', fill_value=0,
              cache_dir='data_cache', use_cache=True):
    """
    Load stock data, handle NaN values, and optionally cache the data locally.
    """
    os.makedirs(cache_dir, exist_ok=True)
    cache_file = f"{cache_dir}/{company}_{start_date}_{end_date}.csv"

    # Check if cached data exists
    if use_cache and os.path.exists(cache_file):
        data = pd.read_csv(cache_file, index_col=0, parse_dates=True)
        print(f"Loaded data from cache: {cache_file}")
    else:
        # Download the data
        data = yf.download(company, start_date, end_date)

    return data
```

Figure 5. Data loading and processing

2. Handling NaN Values

Data quality is critical for accurate predictions, and missing values (NaNs) in the dataset can negatively impact model performance. To address this, the function was designed to handle NaN values using multiple methods: dropping rows with NaNs, filling them with a specific value, or using forward/backward fill techniques. This flexibility allows the user to choose the most appropriate method based on the nature of the dataset and the importance of maintaining data continuity.

```

# Handle NaN values
if nan_handling == 'drop':
    data.dropna(inplace=True)
elif nan_handling == 'fill':
    data.fillna(fill_value, inplace=True)
elif nan_handling == 'ffill':
    data.ffill(inplace=True)
elif nan_handling == 'bfill':
    data.bfill(inplace=True)
else:
    raise ValueError("Invalid NaN handling method.")

```

Figure 6. Handling NaN values

3. Data Splitting

Effective data splitting is crucial for model evaluation. The function supports various methods for dividing the dataset into training and testing subsets. Users can choose to split the data by a specified ratio (e.g., 80% training, 20% testing), by a specific date (using `split_date`), or randomly. This flexibility in splitting methods ensures that the data can be appropriately divided based on the specific requirements of the analysis, whether it be time-based predictions or more generalized model training.

```

def prepare_data(data, feature_columns, prediction_days, split_method='ratio',
                split_ratio=0.8, split_date=None, random_split=False):
    """
    Prepare, scale, and split stock data for model training.
    """
    scalars = {}
    scaled_data = {}

    # Scale each feature column and store the scaler
    for feature in feature_columns:
        scaler = MinMaxScaler(feature_range=(0, 1))
        scaled_data[feature] = scaler.fit_transform(data[feature].values.reshape(-1, 1))
        scalars[feature] = scaler

    x_data, y_data = [], []
    for x in range(prediction_days, len(scaled_data[feature_columns[0]])):
        x_data.append(np.hstack([scaled_data[feature][x - prediction_days:x, 0] for feature in feature_columns]))
        y_data.append(scaled_data[feature_columns[0]][x, 0]) # Assuming 'Close' or first feature column for y_data

    x_data = np.array(x_data).reshape(-1, prediction_days, len(feature_columns))
    y_data = np.array(y_data)

    if split_method == 'date' and split_date:
        split_index = data.index.get_loc(split_date)
        x_train, x_test = x_data[:split_index], x_data[split_index:]
        y_train, y_test = y_data[:split_index], y_data[split_index:]
    elif split_method == 'ratio':
        if random_split:
            x_train, x_test, y_train, y_test = train_test_split(*arrays: x_data, y_data, train_size=split_ratio, random_state=42)
        else:
            split_index = int(len(x_data) * split_ratio)
            x_train, x_test = x_data[:split_index], x_data[split_index:]
            y_train, y_test = y_data[:split_index], y_data[split_index:]
    else:
        raise ValueError("Invalid split method.")

```

Figure 7. Data Splitting

4. Data Caching

To optimize the data retrieval process, especially when working with large datasets or conducting multiple experiments, a caching mechanism was implemented. The function allows the data to be saved locally after the initial download, and subsequent runs can load the data directly from the local cache if available. This reduces the need for repeated data downloads, significantly speeding up the data processing pipeline and making the workflow more efficient.

```
def load_data(company, start_date, end_date, nan_handling='drop', fill_value=0,
              cache_dir='data_cache', use_cache=True):
    """
    Load stock data, handle NaN values, and optionally cache the data locally.
    """
    os.makedirs(cache_dir, exist_ok=True)
    cache_file = f"{cache_dir}/{company}_{start_date}_{end_date}.csv"

    # Check if cached data exists
    if use_cache and os.path.exists(cache_file):
        data = pd.read_csv(cache_file, index_col=0, parse_dates=True)
        print(f"Loaded data from cache: {cache_file}")
    else:
        # Download the data
        data = yf.download(company, start_date, end_date)
```

Figure 8. data catching

5. Feature Scaling

Scaling the feature columns is a critical step in preparing data for machine learning models, particularly for models like LSTM that are sensitive to the scale of input data. The function was extended to scale multiple features (e.g., 'Close', 'Volume') using MinMaxScaler, which normalizes the data to a specified range (typically 0 to 1). The scalers used for each feature are stored in a dictionary, allowing for easy retrieval and inverse transformation during model evaluation and prediction. This ensures that predictions can be accurately compared to actual values on the original scale.

```
scalers = {}
scaled_data = {}

# Scale each feature column and store the scaler
for feature in feature_columns:
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data[feature] = scaler.fit_transform(data[feature].values.reshape(-1, 1))
    scalers[feature] = scaler
return x_train, y_train, x_test, y_test, scalers
```

Figure 9. scaling feature

Code Analysis

The enhancements made to the original stock prediction code required a thorough understanding of several key functions and programming concepts. This section provides a detailed analysis of the most critical components, particularly those that required additional research and careful consideration.

1. Data Loading and NaN Handling

The `load_data` function in `data_processing.py` is central to the entire workflow. It not only retrieves stock data from Yahoo Finance but also addresses potential issues with missing values (NaNs) in the dataset.

```
def load_data(company, start_date, end_date, nan_handling='drop', fill_value=0,
              cache_dir='data_cache', use_cache=True):
    """
    Load stock data, handle NaN values, and optionally cache the data locally.
    """
    os.makedirs(cache_dir, exist_ok=True)
    cache_file = f"{cache_dir}/{company}_{start_date}_{end_date}.csv"

    # Check if cached data exists
    if use_cache and os.path.exists(cache_file):
        data = pd.read_csv(cache_file, index_col=0, parse_dates=True)
        print(f"Loaded data from cache: {cache_file}")
    else:
        # Download the data
        data = yf.download(company, start_date, end_date)

        # Handle NaN values
        if nan_handling == 'drop':
            data.dropna(inplace=True)
        elif nan_handling == 'fill':
            data.fillna(fill_value, inplace=True)
        elif nan_handling == 'ffill':
            data.ffill(inplace=True)
        elif nan_handling == 'bfill':
            data.bfill(inplace=True)
        else:
            raise ValueError("Invalid NaN handling method.")

        # Save data to cache
        if use_cache:
            data.to_csv(cache_file)
            print(f"Saved data to cache: {cache_file}")

    return data
```

Figure 10. `load_data` function

- **Data Downloading:** The function uses `yfinance` to download stock data for a specified date range. The use of a `cache_dir` ensures that data is stored locally, reducing the need for repeated downloads.

- **NaN Handling:** The function offers multiple options for dealing with missing data, including dropping rows (dropna), filling with a specific value (fillna), and using forward or backward fill (ffill and bfill). The choice of method depends on the nature of the dataset and the specific analysis needs. This flexibility allows users to handle missing data in the way that best suits their analysis.

2. Data Splitting

The `prepare_data` function is responsible for scaling the data and splitting it into training and testing sets. The function supports splitting by a specified ratio, by a specific date, or randomly.

```
def prepare_data(data, feature_columns, prediction_days, split_method='ratio',
                split_date=None, split_ratio=0.8, random_state=42):
    """
    Prepare, scale, and split stock data for model training.
    """
    scalers = {}
    scaled_data = {}

    # Scale each feature column and store the scaler
    for feature in feature_columns:
        scaler = MinMaxScaler(feature_range=(0, 1))
        scaled_data[feature] = scaler.fit_transform(data[feature].values.reshape(-1, 1))
        scalers[feature] = scaler

    x_data, y_data = [], []
    for x in range(prediction_days, len(scaled_data[feature_columns[0]])):
        x_data.append(np.hstack([scaled_data[feature][x - prediction_days:x, 0] for feature in feature_columns]))
        y_data.append(scaled_data[feature_columns[0]][x, 0]) # Assuming 'Close' or first feature column for y_data

    x_data = np.array(x_data).reshape(-1, prediction_days, len(feature_columns))
    y_data = np.array(y_data)

    if split_method == 'date' and split_date:
        split_index = data.index.get_loc(split_date)
        x_train, x_test = x_data[:split_index], x_data[split_index:]
        y_train, y_test = y_data[:split_index], y_data[split_index:]
    elif split_method == 'ratio':
        if random_split:
            x_train, x_test, y_train, y_test = train_test_split(*arrays: x_data, y_data, train_size=split_ratio, random_state=42)
        else:
            split_index = int(len(x_data) * split_ratio)
            x_train, x_test = x_data[:split_index], x_data[split_index:]
            y_train, y_test = y_data[:split_index], y_data[split_index:]
    else:
        raise ValueError("Invalid split method.")

    return x_train, y_train, x_test, y_test, scalers
```

Figure 11. `prepare_data` function

- **Feature Scaling:** The function scales each feature using `MinMaxScaler`, which normalizes the data to a range between 0 and 1. This step is crucial for models like LSTM, which are sensitive to the scale of input data.
- **Data Splitting:** The function supports three methods of data splitting:
 - **Ratio-Based Split:** Divides the data into training and testing sets based on a specified ratio.
 - **Date-Based Split:** Uses a specific date (`split_date`) to separate the training and testing periods.
 - **Random Split:** Provides an option to randomly shuffle and split the data, ensuring that the model is not biased by the order of the data.

3. Model Building and Training

The `model_operations.py` file contains the logic for building, training, and evaluating the predictive model. The choice of architecture and training parameters is crucial for model performance.

```

2 usages
def build_model(input_shape):
    model = Sequential()

    model.add(LSTM(units=50, return_sequences=True, input_shape=input_shape))
    model.add(Dropout(0.2))
    model.add(LSTM(units=50, return_sequences=True))
    model.add(Dropout(0.2))
    model.add(LSTM(units=50))
    model.add(Dropout(0.2))
    model.add(Dense(units=1))

    model.compile(optimizer='adam', loss='mean_squared_error')

    return model

```

Figure 12. build_model function

- **LSTM Layers:** The model consists of three stacked LSTM layers, each with 50 units. Stacking multiple LSTM layers allows the model to capture complex temporal dependencies in the data, which is essential for accurate stock price predictions.
- **Dropout Layers:** Dropout is used after each LSTM layer to prevent overfitting by randomly setting a fraction of the input units to zero during training. This helps the model generalize better to unseen data.
- **Output Layer:** A Dense layer with a single unit is used for the final output, which predicts the stock price for the next day.

4. Prediction and Inverse Transformation

The predictor.py file is focused on making predictions using the trained model. It includes functions for generating predictions and transforming them back to the original scale.

```

def predict_next_day(model, model_inputs, scaler, prediction_days):
    real_data = [model_inputs[len(model_inputs) - prediction_days:, 0]]
    real_data = np.array(real_data)
    real_data = np.reshape(real_data, newshape=(real_data.shape[0], real_data.shape[1], 1))

    prediction = model.predict(real_data)
    prediction = scaler.inverse_transform(prediction)

    return prediction

```

Figure 13. predict_next_day function

- **Prediction Generation:** The function takes the last sequence of input data, uses the trained model to predict the next day's stock price, and then applies an inverse transformation to convert the prediction back to its original scale.
- **Inverse Transformation:** The inverse transformation is necessary because the data was scaled during preprocessing. Without this step, the predicted prices would remain in the normalized range (0-1), making them difficult to interpret.

5. Inclusion of New Variables in main.py

The main.py file now includes several new variables that enhance the flexibility and control over the data processing and model training process.

```

# COMPANY: The stock ticker symbol of the company to analyze.
COMPANY = 'CBA.AX'

# TRAIN_START, TRAIN_END: The start and end dates for the training data.
TRAIN_START, TRAIN_END = '2020-01-01', '2023-08-01'

# TEST_START, TEST_END: The start and end dates for the testing data.
TEST_START, TEST_END = '2023-08-02', '2024-07-02'

# PREDICTION_DAYS: Number of days to look back when creating input sequences for prediction.
PREDICTION_DAYS = 60

# FEATURE_COLUMNS: List of features (columns) from the dataset to be scaled and used for training.
FEATURE_COLUMNS = ["Close", "Volume"]

# NAN_METHOD: Method for handling missing data (NaN values). Options: 'drop', 'fill', 'ffill', 'bfill'.
# FILL_VALUE: Value to use for filling NaNs if 'fill' method is selected.
NAN_METHOD, FILL_VALUE = 'ffill', 0

# SPLIT_METHOD: Method for splitting the dataset into training and testing sets. Options: 'ratio', 'date'.
# SPLIT_RATIO: Ratio of the data to be used for training if 'ratio' method is selected.
# SPLIT_DATE: Specific date to split the data if 'date' method is selected.
SPLIT_METHOD = 'ratio'
SPLIT_RATIO = 0.8
SPLIT_DATE = '2023-01-01'

# RANDOM_SPLIT: Whether to split the data randomly (True) or sequentially (False).
RANDOM_SPLIT = False

# USE_CACHE: Whether to load data from a local cache if available, to save time on downloading.
USE_CACHE = True

# CACHE_DIR: Directory where cached data will be stored and loaded from.
CACHE_DIR = 'data_cache'

```

Figure 14. variables in main.py

Model Execution and Outputs

This section demonstrates how the enhanced stock prediction model runs and the outputs it produces. The goal is to provide a clear overview of the model's performance and its ability to predict stock prices based on the given input data.

Execution Process

The model was executed on the CBA.AX stock data, covering a training period from January 2020 to August 2023 and a testing period from August 2023 to July 2024. The training process involved 25 epochs, during which the model's loss steadily decreased, indicating effective learning.

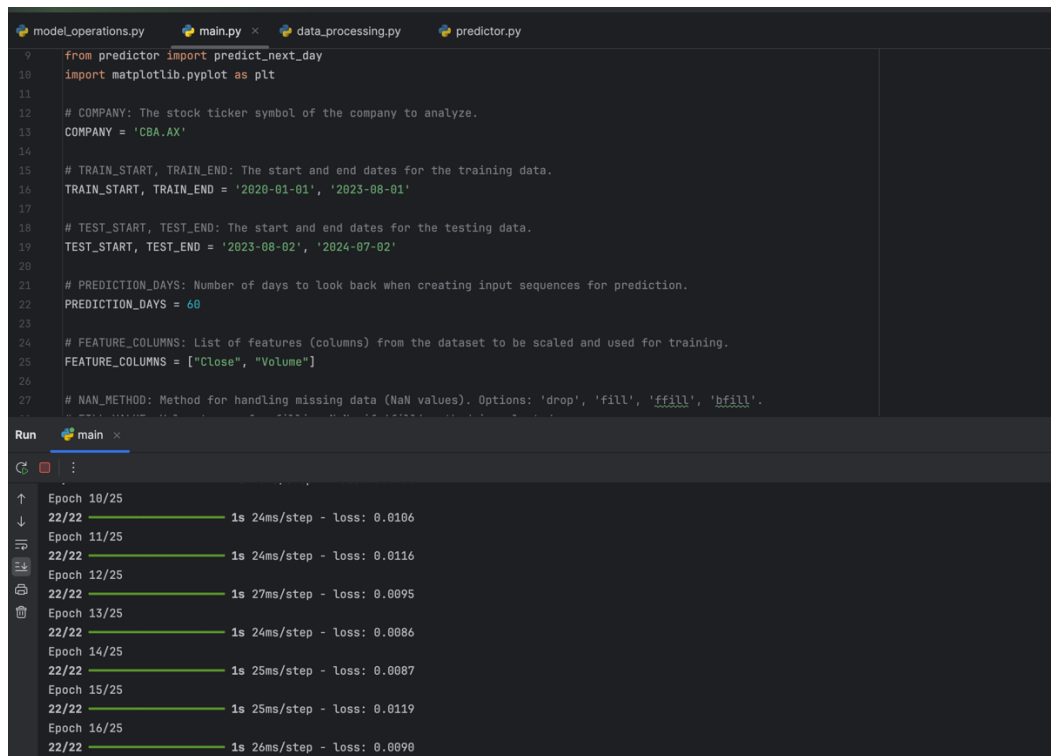


Figure 15. Training process

Visual Output

The following plot illustrates the model's predictions against the actual stock prices for the testing period:

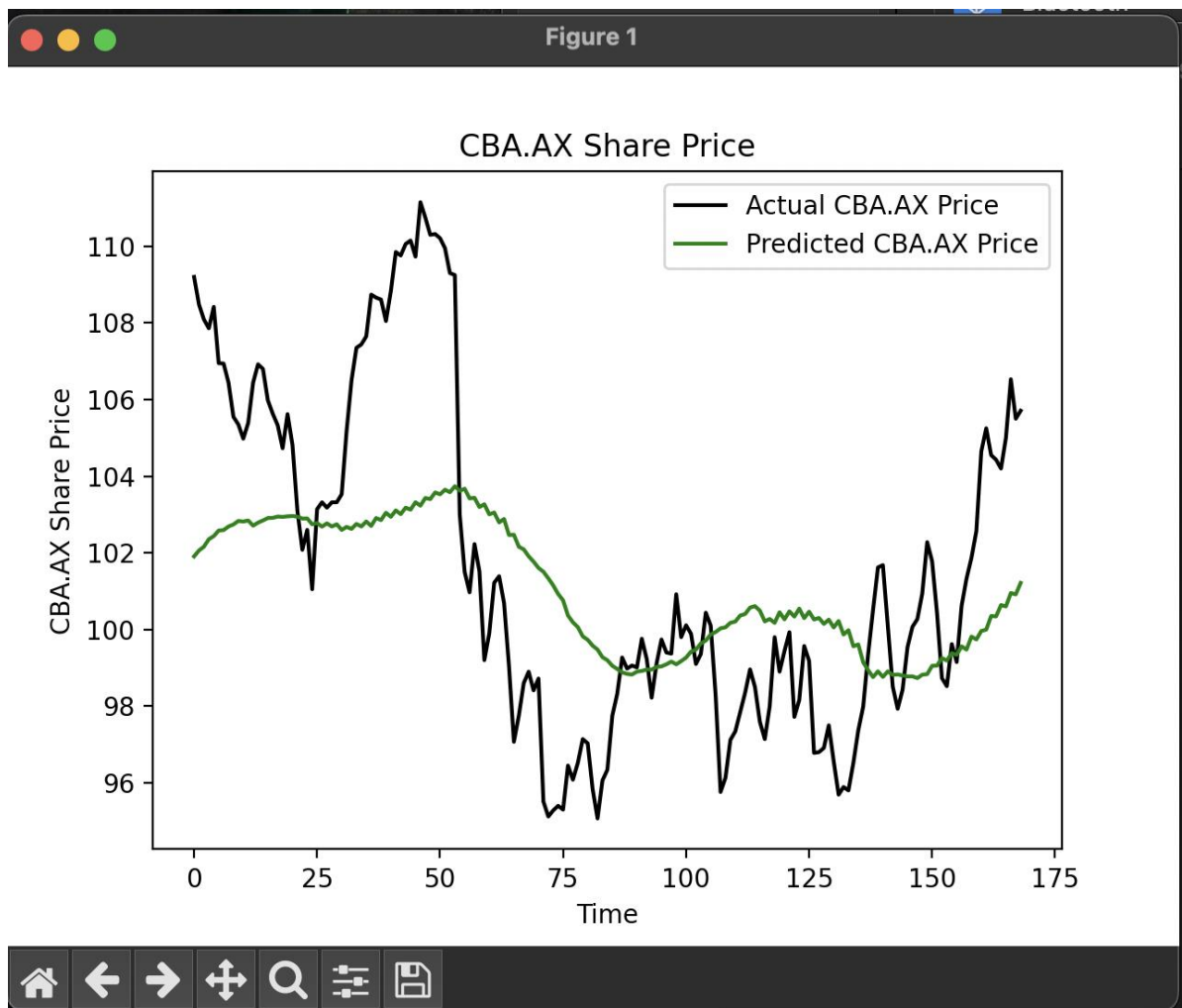


Figure 16. stock prediction result

- **Black Line:** Represents the actual CBA.AX share prices.
- **Green Line:** Represents the model's predicted prices.

This output shows that the model captures the general trend of the stock prices, although some discrepancies are observed during periods of high volatility. These discrepancies highlight areas where further tuning and enhancements could improve prediction accuracy.

GitHub Repository

The project's code base is hosted on GitHub for version control and review. Everyone can access the repository via the following link: [GitHub Repository](#). This repository contains all necessary files for the project and is available for the tutor to review the work.

References

- Simplilearn. (n.d.). *Stock Price Prediction Using Machine Learning*. Retrieved from <https://www.simplilearn.com/tutorials/machine-learning-tutorial/stock-price-prediction-using-machine-learning>
- GeeksforGeeks. (n.d.). *Stock Price Prediction using Machine Learning in Python*. Retrieved from <https://www.geeksforgeeks.org/stock-price-prediction-using-machine-learning-in-python/>
- NeuralNine. (2022, October 1). *Stock Price Prediction using LSTM in Python*. [Video]. YouTube. Retrieved from https://www.youtube.com/watch?v=CeTR_-ALdRw&list=PL7yh-TELLS1G9mmnBN3ZSY8hYgJ5kBOg-&index=4

- GeeksforGeeks. (n.d.). *Best Python Libraries for Machine Learning*. Retrieved from <https://www.geeksforgeeks.org/best-python-libraries-for-machine-learning/>