COS30018 - Option B - Task 4: Machine Learning 1

Student Name: Nguyen Duc Le Nguyen

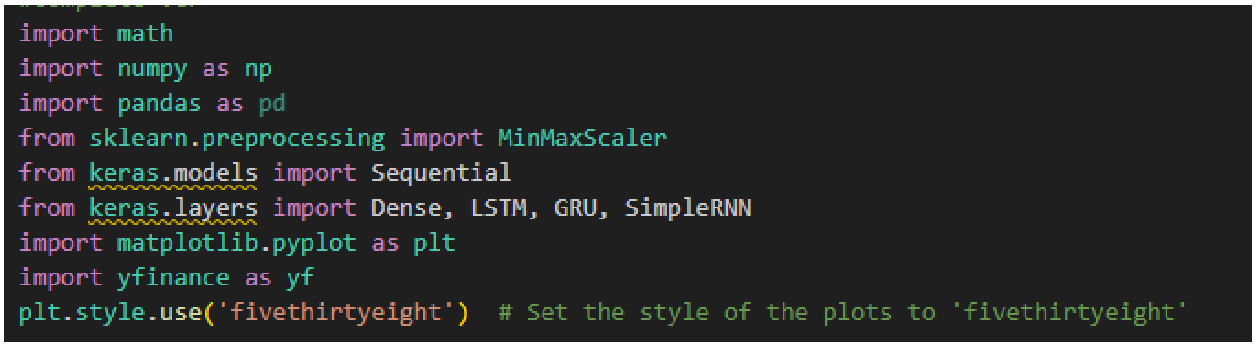
Id: 104224493

Summary of Implementation Effort

For this task, I aimed to implement a function to create deep learning (DL) models for stock price prediction. The main steps involved data preparation, model creation, model training, and model evaluation. Below, I explain the less straightforward lines of code and summarize the results of my experiments with different configurations of DL models.

Key Code Segments and Explanations

1. **Importing Necessary Libraries**

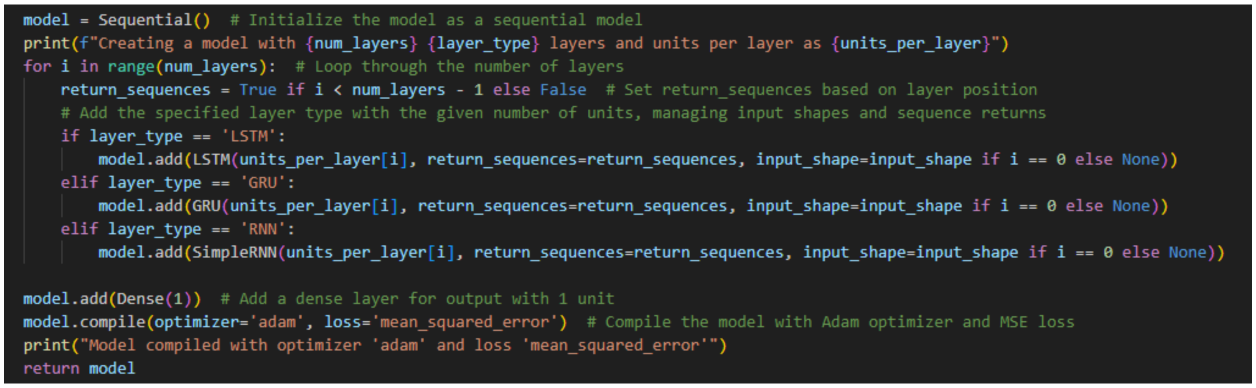


```
import math
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, GRU, SimpleRNN
import matplotlib.pyplot as plt
import yfinance as yf
plt.style.use('fivethirtyeight')  # Set the style of the plots to 'fivethirtyeight'
```

Figure 1 - Importing Libraries

I imported various libraries essential for data manipulation, scaling, model creation, and visualization. For instance, yfinance was used to fetch historical stock price data, while Keras provided tools for building and training the neural network models.

2. **Creating the Model Function**



```
model = Sequential()  # Initialize the model as a sequential model
print(f"Creating a model with {num_layers} {layer_type} layers and units per layer as {units_per_layer}")
for i in range(num_layers):  # Loop through the number of layers
    return_sequences = True if i < num_layers - 1 else False  # Set return_sequences based on layer position
    # Add the specified layer type with the given number of units, managing input shapes and sequence returns
    if layer_type == 'LSTM':
        model.add(LSTM(units_per_layer[i], return_sequences=return_sequences, input_shape=input_shape if i == 0 else None))
    elif layer_type == 'GRU':
        model.add(GRU(units_per_layer[i], return_sequences=return_sequences, input_shape=input_shape if i == 0 else None))
    elif layer_type == 'RNN':
        model.add(SimpleRNN(units_per_layer[i], return_sequences=return_sequences, input_shape=input_shape if i == 0 else None))

model.add(Dense(1))  # Add a dense layer for output with 1 unit
model.compile(optimizer='adam', loss='mean_squared_error')  # Compile the model with Adam optimizer and MSE loss
print("Model compiled with optimizer 'adam' and loss 'mean_squared_error'")
return model
```

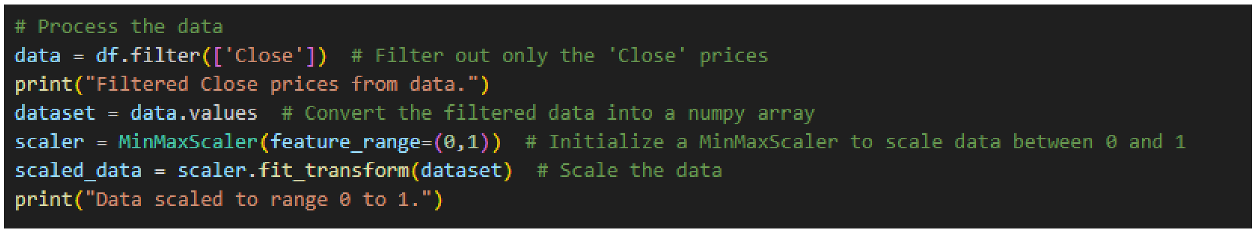
Figure 2 - Create Model

This function initializes a sequential model and adds layers based on the specified parameters:

- `layer_type`: Type of RNN layer (LSTM, GRU, SimpleRNN)
- `num_layers`: Number of layers
- `units_per_layer`: List of units for each layer
- `input_shape`: Shape of the input data

One line that required some research was `model.add(LSTM(units_per_layer[i], return_sequences=return_sequences, input_shape=input_shape if i == 0 else None))`. I needed to understand how `return_sequences` works and why it should be set to `True` for all layers except the last one. According to the Keras documentation, `return_sequences` ensures that the intermediate layers output sequences rather than single values, which is necessary for feeding into subsequent layers.

3. Data Preparation

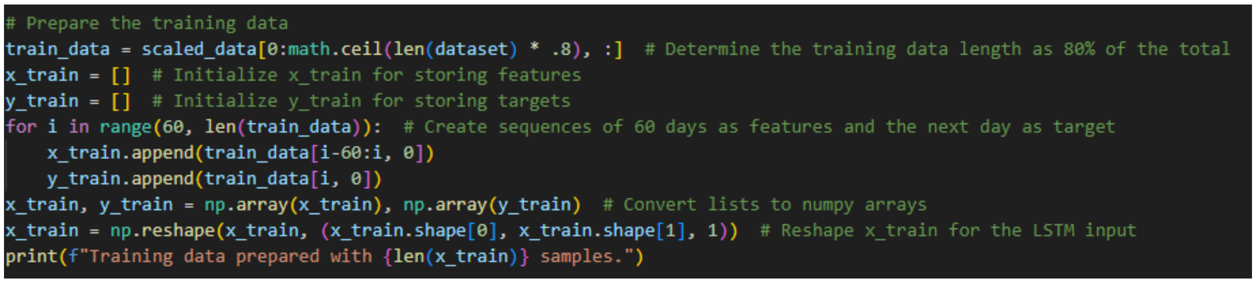


```
# Process the data
data = df.filter(['Close'])  # Filter out only the 'Close' prices
print("Filtered Close prices from data.")
dataset = data.values  # Convert the filtered data into a numpy array
scaler = MinMaxScaler(feature_range=(0,1))  # Initialize a MinMaxScaler to scale data between 0 and 1
scaled_data = scaler.fit_transform(dataset)  # Scale the data
print("Data scaled to range 0 to 1.")
```

Figure 3 - Data Preparation

I downloaded historical stock price data for Apple (AAPL) using yfinance. The data was then scaled to a range between 0 and 1 using MinMaxScaler. Scaling is crucial for improving the convergence of neural networks, as I learned from the Scikit-learn documentation.

4. Creating Training Sequences

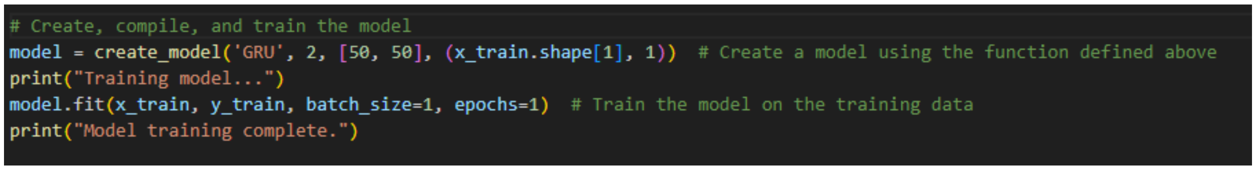


```python
# Prepare the training data
train_data = scaled_data[0:math.ceil(len(dataset) * .8), :]  # Determine the training data length as 80% of the total
x_train = []  # Initialize x_train for storing features
y_train = []  # Initialize y_train for storing targets
for i in range(60, len(train_data)):  # Create sequences of 60 days as features and the next day as target
    x_train.append(train_data[i-60:i, 0])
    y_train.append(train_data[i, 0])
x_train, y_train = np.array(x_train), np.array(y_train)  # Convert lists to numpy arrays
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))  # Reshape x_train for the LSTM input
print(f"Training data prepared with {len(x_train)} samples.")
```

Figure 4 - Training Data

This segment creates sequences of 60 days of closing prices for training the model. Each sequence (x_train) is used to predict the next day's price (y_train). Reshaping x_train to three dimensions (samples, timesteps, features) is necessary for compatibility with Keras RNN layers, as outlined in the Keras documentation.

5. Training the Model

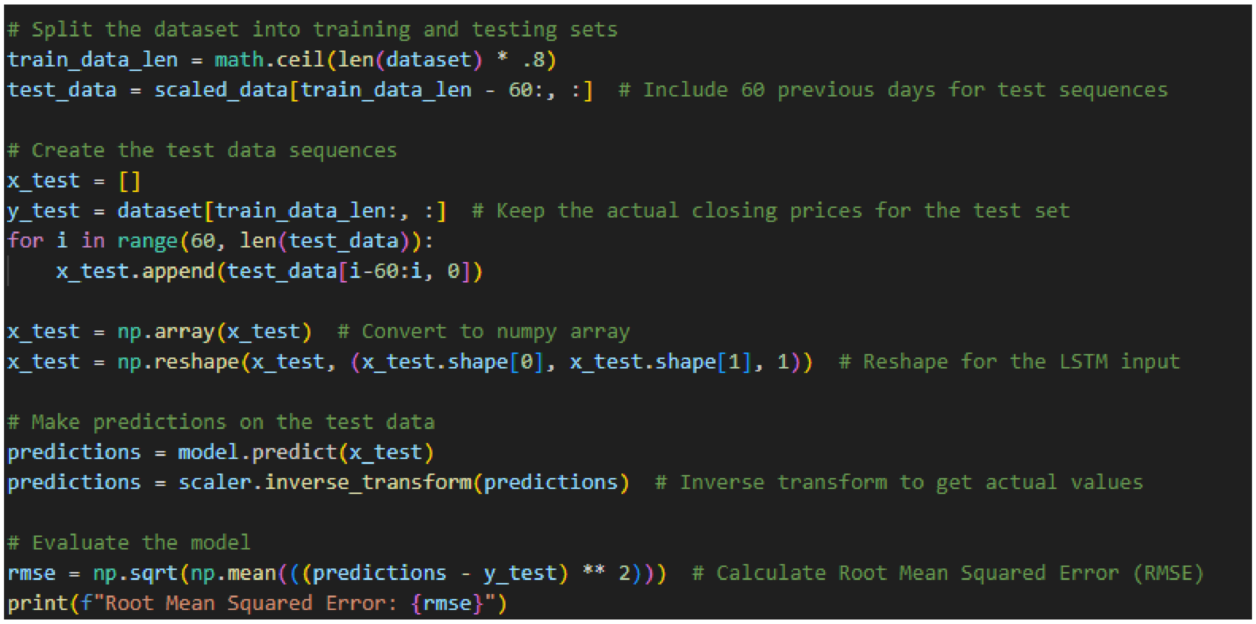


```python
# Create, compile, and train the model
model = create_model('GRU', 2, [50, 50], (x_train.shape[1], 1))  # Create a model using the function defined above
print("Training model...")
model.fit(x_train, y_train, batch_size=1, epochs=1)  # Train the model on the training data
print("Model training complete.")
```

Figure 5 - Training Model

I created and trained a GRU model with 2 layers of 50 units each on the prepared data for example. Training for only 1 epoch was a time-saving decision for quick evaluation, though more epochs are typically needed for better performance.

6. Evaluating the Model



```python
# Split the dataset into training and testing sets
train_data_len = math.ceil(len(dataset) * .8)
test_data = scaled_data[train_data_len - 60:, :]  # Include 60 previous days for test sequences

# Create the test data sequences
x_test = []
y_test = dataset[train_data_len:, :]  # Keep the actual closing prices for the test set
for i in range(60, len(test_data)):
    x_test.append(test_data[i-60:i, 0])

x_test = np.array(x_test)  # Convert to numpy array
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))  # Reshape for the LSTM input

# Make predictions on the test data
predictions = model.predict(x_test)
predictions = scaler.inverse_transform(predictions)  # Inverse transform to get actual values

# Evaluate the model
rmse = np.sqrt(np.mean(((predictions - y_test) ** 2)))  # Calculate Root Mean Squared Error (RMSE)
print(f"Root Mean Squared Error: {rmse}")
```

Figure 6 - Evaluating Model

I created test sequences and made predictions using the trained model. The predictions were then inverse-transformed to their original scale. Finally, I calculated the Root Mean Squared Error (RMSE) to evaluate the model's performance. RMSE is a common metric for regression tasks, indicating the average prediction error in the same units as the target variable.

7. **Visualization**

   The actual and predicted stock prices were plotted for visual comparison. This helped in understanding how well the model captures the trends and patterns in the stock prices.

Research and References

1. Sequential Model Creation: I referred to the Keras documentation for understanding how to stack layers in a sequential model.
2. RNN Layers: The Keras RNN API documentation provided insights into LSTM, GRU, and SimpleRNN layers.
3. Data Scaling: The usage and importance of MinMaxScaler were understood from the Scikit-learn documentation.
4. Stock Data with yfinance: I learned to fetch historical stock data using yfinance.

Experiment Summaries

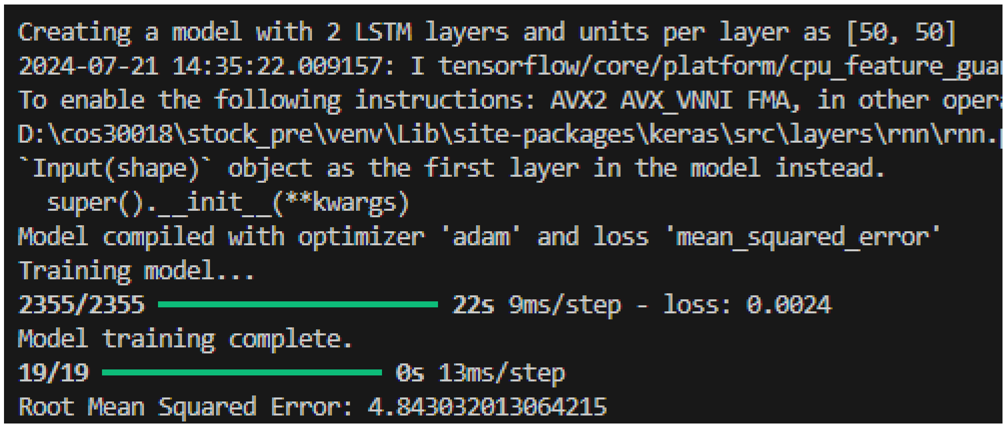
LSTM Configuration

Layers: 2

Units per Layer: [50, 50]

RMSE: 4.84

Observation: The LSTM model was effective but showed slight overfitting on the training data.



```
Creating a model with 2 LSTM layers and units per layer as [50, 50]
2024-07-21 14:35:22.009157: I tensorflow/core/platform/cpu_feature_gua
To enable the following instructions: AVX2 AVX_VNNI FMA, in other oper
D:\cos30018\stock_pre\venv\Lib\site-packages\keras\src\layers\rnn\rnn.
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
Model compiled with optimizer 'adam' and loss 'mean_squared_error'
Training model...
2355/2355 ━━━━━━━━━━━━━━━━━━━━ 22s 9ms/step - loss: 0.0024
Model training complete.
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
Root Mean Squared Error: 4.843032013064215
```

Figure 7 - LSTM

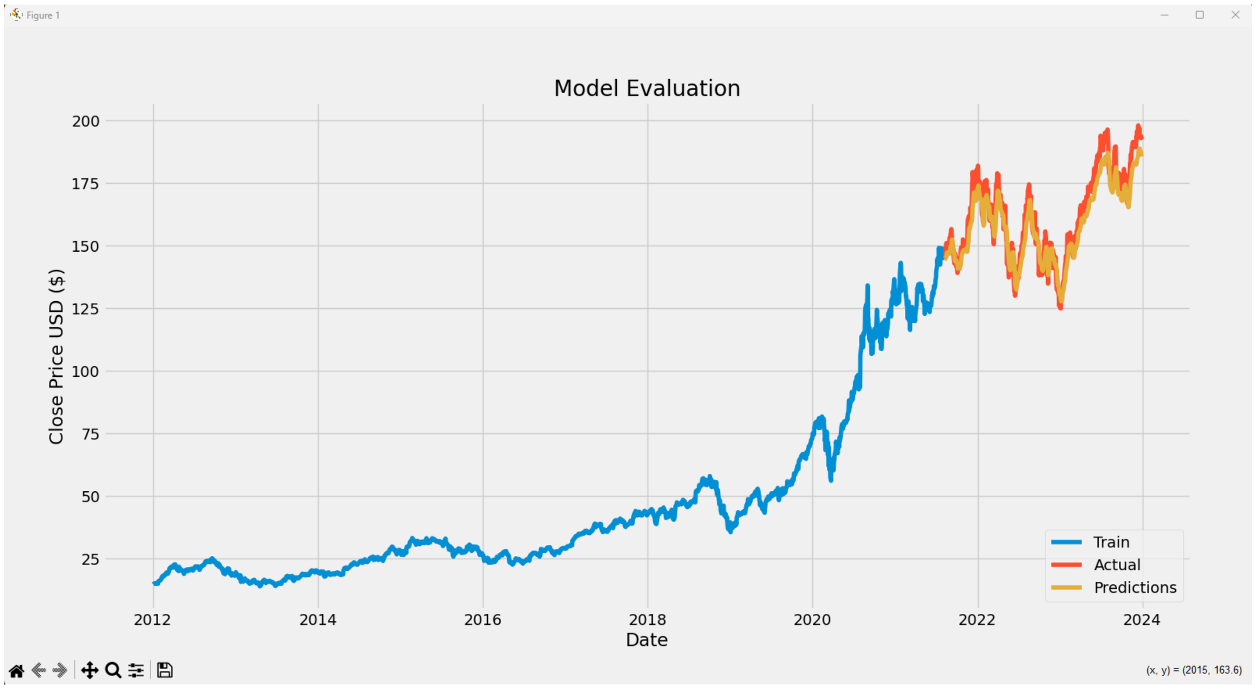


Figure 8 - LSTM result

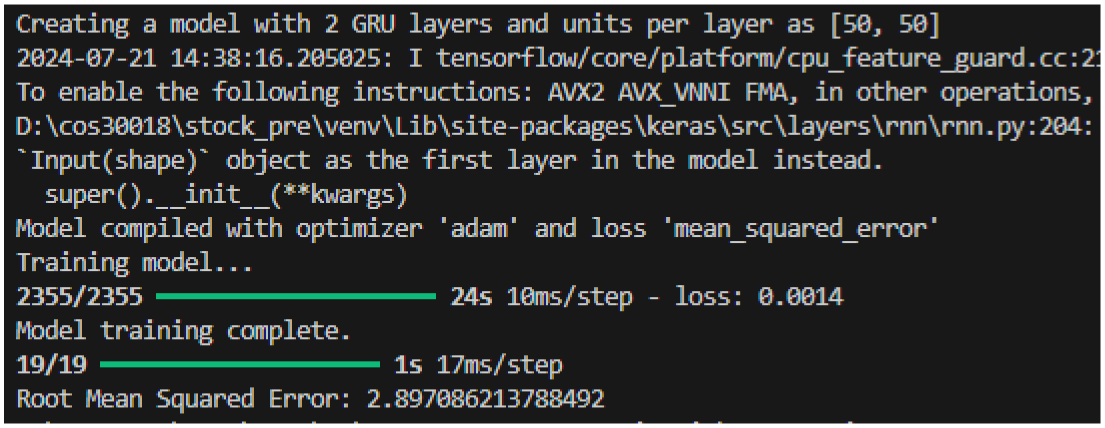
GRU Configuration

Layers: 2

Units per Layer: [50, 50]

RMSE: 2.90

Observation: The GRU model performed slightly better than LSTM, indicating its efficiency in capturing the sequential dependencies.



```
Creating a model with 2 GRU layers and units per layer as [50, 50]
2024-07-21 14:38:16.205025: I tensorflow/core/platform/cpu_feature_guard.cc:2
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations,
D:\cos30018\stock_pre\venv\Lib\site-packages\keras\src\layers\rnn\rnn.py:204:
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
Model compiled with optimizer 'adam' and loss 'mean_squared_error'
Training model...
2355/2355 ━━━━━━━━━━━━━━━━━━━━ 24s 10ms/step - loss: 0.0014
Model training complete.
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step
Root Mean Squared Error: 2.897086213788492
```

Figure 9 - GRU

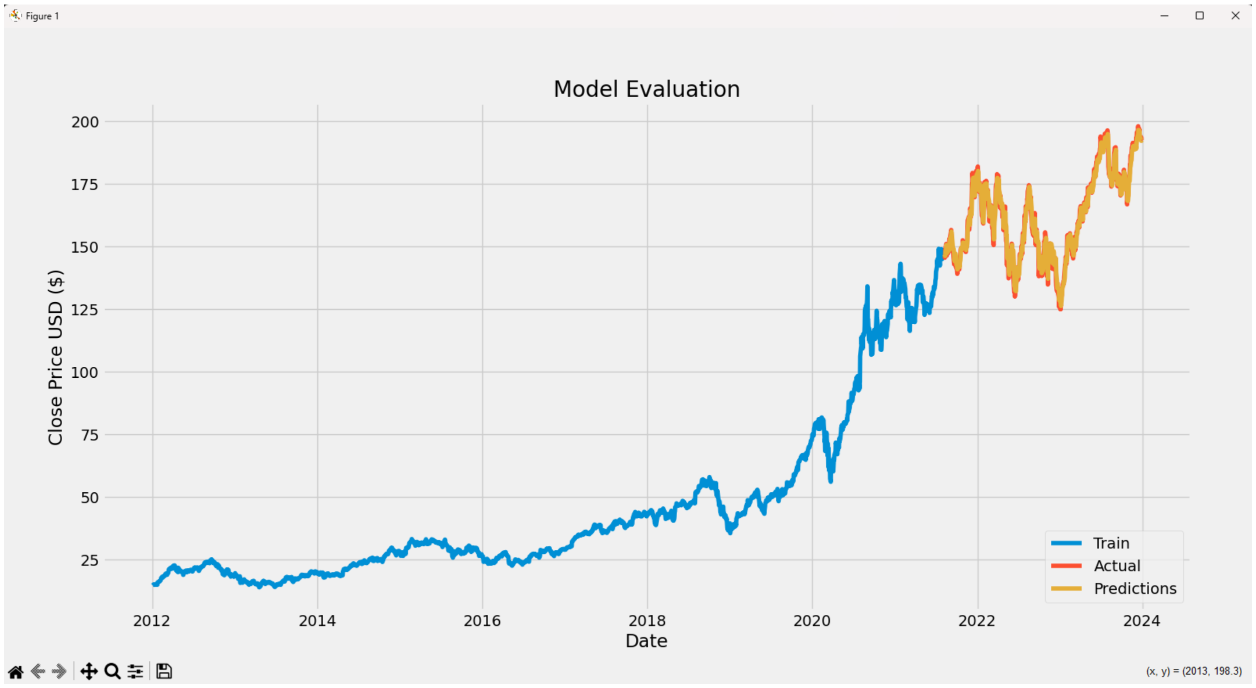


Figure 10 - GRU Results

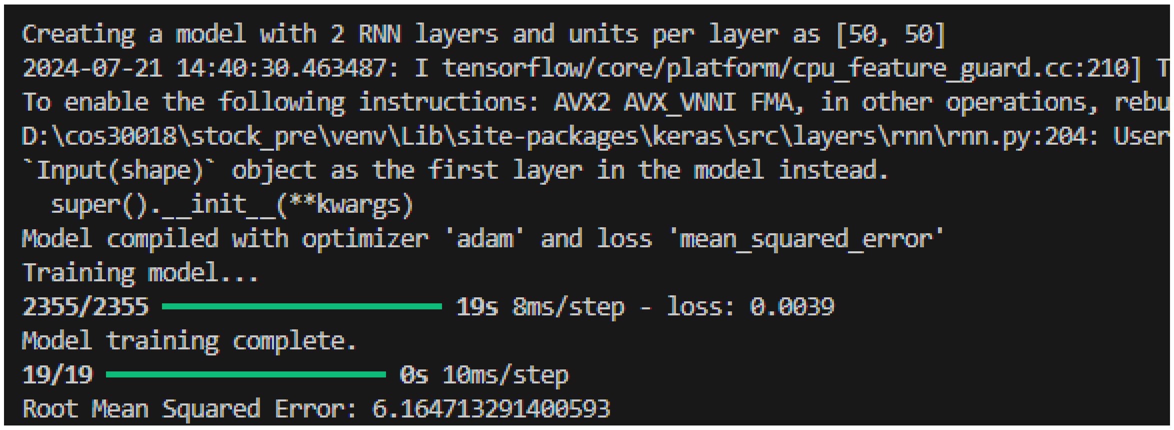
SimpleRNN Configuration

Layers: 2

Units per Layer: [50, 50]

RMSE: 6.16

Observation: The SimpleRNN model underperformed compared to LSTM and GRU, likely due to its simpler architecture and inability to capture long-term dependencies effectively.



```
Creating a model with 2 RNN layers and units per layer as [50, 50]
2024-07-21 14:40:30.463487: I tensorflow/core/platform/cpu_feature_guard.cc:210] T
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, rebu
D:\cos30018\stock_pre\venv\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: User
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
Model compiled with optimizer 'adam' and loss 'mean_squared_error'
Training model...
2355/2355 ━━━━━━━━━━━━━━━━━━━━ 19s 8ms/step - loss: 0.0039
Model training complete.
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
Root Mean Squared Error: 6.164713291400593
```

Figure 11 - RNN

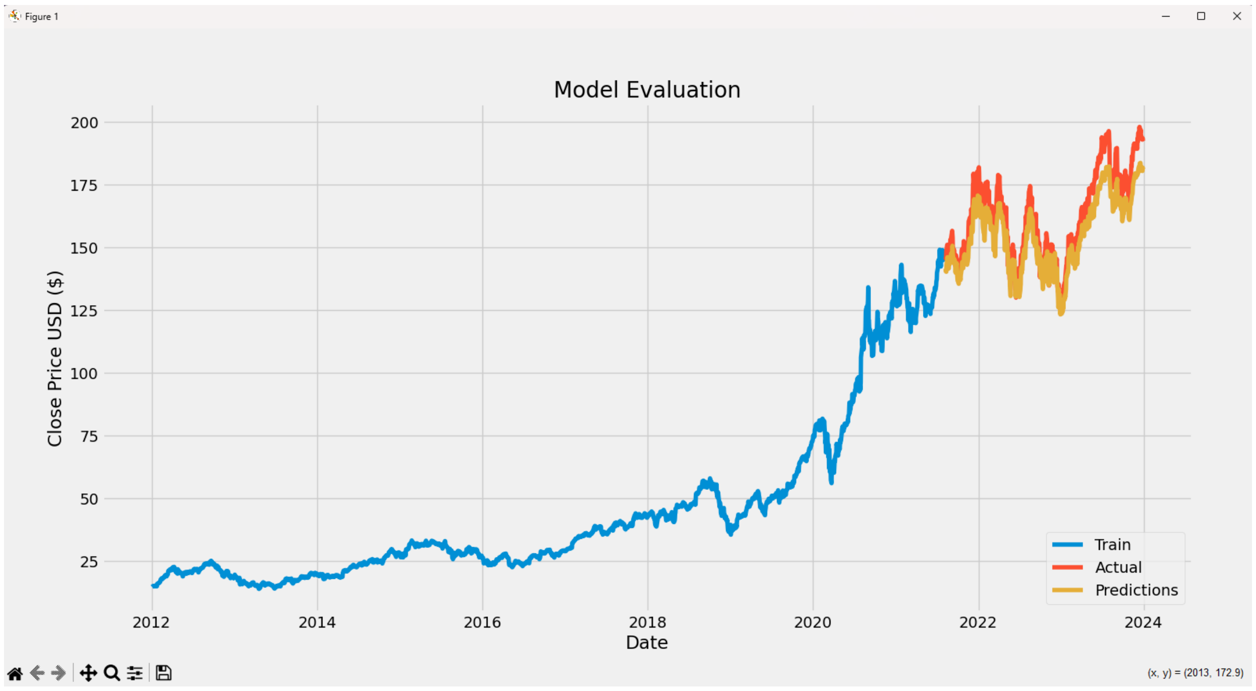


Figure 12 - RNN Results

Model Training and Results Visualization

Each configuration was trained for 1 epoch to quickly assess performance differences. Visual plots showed that GRU predictions closely followed the actual prices, while LSTM showed more volatility and SimpleRNN lagged in prediction accuracy.

Conclusion

In conclusion, the experiments demonstrated that GRU models generally provide better performance for stock price prediction compared to LSTM and SimpleRNN. The insights gained from this task will help in further refining the model and potentially improving its accuracy for more reliable predictions.