Task Report Cos30018 Option B

B.5: Machine Processing 2

Name: Le Bao Nguyen

Student Id: 104169837

I. Importing libraries (From the task B.2):

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, GRU, SimpleRNN, Dropout
from tensorflow.keras.optimizers import Adam
from IPython.display import display, HTML
from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Figure 1: Importing libraries to run the code.

- We still use the same libraries like before.

II. Data loading and processing (From the task B.2):

```
[ ]  # Download data from Yahoo Finance
     data = yf.download(ticker, start=start_date, end=end_date)

     # Ensure the index is a DateTimeIndex
     data.index = pd.to_datetime(data.index)

     # Fill NaN values with previous values
     data.fillna(method='ffill', inplace=True)

     # Sanity check: Ensure high is not less than low
     if (data['High'] < data['Low']).any():
         raise ValueError("Inconsistent data: High value is less than Low value for some periods.")

     # Default to scaling the 'Close' column if no columns are specified
     if columns_to_scale is None or not columns_to_scale:
         columns_to_scale = ['Close']

     # Create a DataFrame for scaled data
     scaled_data = data.copy()
     scalers = {}

     # Scale specified columns
     for column in columns_to_scale:
         scaler = MinMaxScaler(feature_range=(0, 1))
         scaled_column = scaler.fit_transform(data[column].values.reshape(-1, 1))
         scaled_data[f'Scaled_{column}'] = scaled_column
         scalers[column] = scaler

     # Extract close prices and scale them
     close_prices = data['Close'].values.reshape(-1, 1)
     scaler = MinMaxScaler(feature_range=(0, 1))
     scaled_close_prices = scaler.fit_transform(close_prices)

     # Determine split date based on split_ratio or split_by_date
     if split_by_date:
         split_date = pd.Timestamp(split_ratio)
     else:
         split_date = pd.to_datetime(start_date) + (pd.to_datetime(end_date) - pd.to_datetime(start_date)) * split_ratio
```

Figure 2: Loading and processing data (1).

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_close_prices = scaler.fit_transform(close_prices)

# Determine split date based on split_ratio or split_by_date
if split_by_date:
    split_date = pd.Timestamp(split_ratio)
else:
    split_date = pd.to_datetime(start_date) + (pd.to_datetime(end_date) - pd.to_datetime(start_date)) * split_ratio

# Split data into train and test sets
if split_by_date:
    train_data = scaled_close_prices[data.index < split_date]
    test_data = scaled_close_prices[data.index >= split_date]
else:
    train_data = scaled_close_prices[:int(len(scaled_close_prices) * split_ratio)]
    test_data = scaled_close_prices[int(len(scaled_close_prices) * split_ratio):]

# Save data to a local file, replacing any existing file
if local_file:
    file_path = f"/content/drive/My Drive/Cos30018/{local_file}"  # Change to your desired path in Google Drive
    data.to_csv(file_path)

return train_data, test_data, scalers, data, scaled_data
```

Figure 3: Loading and processing data (2).

- We still use the same data loading and processing just like B.2.

III. Displaying data in a custom table (From the task B.2):

```
[ ] def display_custom_table(df, num_rows=5):
        """
        Display the first few and last few rows of the DataFrame with ellipses in between.

        Parameters:
        - df: DataFrame to display.
        - num_rows: Number of rows to display from the start and end of the DataFrame.
        """
        if len(df) <= 2 * num_rows:
            # Display the entire DataFrame if it's small enough
            display(df)
        else:
            # Display the first few and last few rows with ellipses in between
            head = df.head(num_rows)
            tail = df.tail(num_rows)
            ellipsis_row = pd.DataFrame([['...'] * len(df.columns)], columns=df.columns, index=['...'])
            df_display = pd.concat([head, ellipsis_row, tail])
            display(HTML(df_display.to_html(index=True)))
```

Figure 4: Displaying the data from csv file.

- We still use the same displaying data function just like B.2.

IV. Model Creation (From the task B.4):

```
def create_dl_model(input_shape, layers_config):
    """
    Create a deep learning model based on the provided configuration.

    Parameters:
    - input_shape: Shape of the input data.
    - layers_config: List of dictionaries where each dictionary specifies the type and parameters of a layer.

    Returns:
    - model: Compiled Keras model.
    """
    model = Sequential()
    for i, layer in enumerate(layers_config):
        layer_type = layer.get("type")
        units = layer.get("units", 50)
        activation = layer.get("activation", "relu")
        return_sequences = layer.get("return_sequences", False)
        dropout_rate = layer.get("dropout_rate", 0.0)

        if layer_type == "LSTM":
            if i == 0:
                model.add(LSTM(units, activation=activation, return_sequences=return_sequences, input_shape=input_shape))
            else:
                model.add(LSTM(units, activation=activation, return_sequences=return_sequences))
        elif layer_type == "GRU":
            if i == 0:
                model.add(GRU(units, activation=activation, return_sequences=return_sequences, input_shape=input_shape))
            else:
                model.add(GRU(units, activation=activation, return_sequences=return_sequences))
        elif layer_type == "RNN":
            if i == 0:
                model.add(SimpleRNN(units, activation=activation, return_sequences=return_sequences, input_shape=input_shape))
            else:
                model.add(SimpleRNN(units, activation=activation, return_sequences=return_sequences))

        if dropout_rate > 0:
            model.add(Dropout(dropout_rate))

    model.add(Dense(1))  # Final layer for output
    model.compile(optimizer=Adam(), loss='mean_squared_error')
    return model
```

Figure 5: Code to create the model.

- We still use the same displaying data function just like B.4.

V. Experimentation with Different Configurations (From the task B.4):

```python
def experiment_with_models(train_data, test_data, scaler, layers_configs, epochs=50, batch_size=32):
    """
    Experiment with different DL networks and configurations.

    Parameters:
    - train_data: Scaled training data.
    - test_data: Scaled testing data.
    - scaler: Scaler used for normalization.
    - layers_configs: List of different configurations to test.
    - epochs: Number of epochs for training.
    - batch_size: Batch size for training.
    """
    results = []
    time_steps = 60
    input_shape = (time_steps, 1)  # Input shape for the model

    # Reshape data for the model
    X_train, y_train = [], []
    X_test, y_test = [], []

    for i in range(time_steps, len(train_data)):
        X_train.append(train_data[i-time_steps:i, 0])
        y_train.append(train_data[i, 0])

    for i in range(time_steps, len(test_data)):
        X_test.append(test_data[i-time_steps:i, 0])
        y_test.append(test_data[i, 0])

    X_train, y_train = np.array(X_train), np.array(y_train)
    X_test, y_test = np.array(X_test), np.array(y_test)

    X_train = np.reshape(X_train, (X_train.shape[0], time_steps, 1))
    X_test = np.reshape(X_test, (X_test.shape[0], time_steps, 1))

    for config in layers_configs:
        model = create_dl_model(input_shape, config['layers'])
        history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(X_test, y_test), verbose=2)
```

Figure 6: Code to experiment with model (1).

```python
    X_train = np.reshape(X_train, (X_train.shape[0], time_steps, 1))
    X_test = np.reshape(X_test, (X_test.shape[0], time_steps, 1))

    for config in layers_configs:
        model = create_dl_model(input_shape, config['layers'])
        history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(X_test, y_test), verbose=2)

        # Predict and inverse transform
        predicted_stock_price = model.predict(X_test)
        predicted_stock_price = scaler.inverse_transform(predicted_stock_price)
        real_stock_price = scaler.inverse_transform(y_test.reshape(-1, 1))

        # Plot the results
        plt.figure(figsize=(14, 5))
        plt.plot(real_stock_price, color='red', label='Real Stock Price')
        plt.plot(predicted_stock_price, color='blue', label='Predicted Stock Price')

        # Create title from config without duplicates
        unique_layers = []
        for layer in config['layers']:
            layer_desc = f"{layer['type']} (units={layer.get('units', 50)})"
            if layer_desc not in unique_layers:
                unique_layers.append(layer_desc)
        model_description = ' - '.join(unique_layers)
        plt.title(f'Stock Price Prediction: {model_description}')
        plt.xlabel('Time')
        plt.ylabel('Stock Price')
        plt.legend()
        plt.show()
        results.append({
            "model": model,  # Include the model in the results dictionary
            "config": config,
            "history": history,
            "predicted_stock_price": predicted_stock_price,
            "real_stock_price": real_stock_price
        })
    return results
```

Figure 7: Code to experiment with model (2).

- We still use the same displaying data function just like B.4 but we add the model into the results in the "experiment_with_models" function.

VI. Multistep and Multivariate Predictions:

```python
def multistep_prediction(model, data, scaler, k=5):
    time_steps = 60
    input_data = data[-time_steps:]  # Select the last 'time_steps' data points
    predictions = []
    for _ in range(k):
        input_data_reshaped = input_data.reshape((1, time_steps, 1))
        next_prediction = model.predict(input_data_reshaped)
        predictions.append(next_prediction[0, 0])
        input_data = np.append(input_data, next_prediction, axis=0)
        input_data = input_data[1:]  # Slide the window forward by one step
    predictions = scaler.inverse_transform(np.array(predictions).reshape(-1, 1))
    return predictions
```

Figure 8: Code to create multistep prediction.

```python
def multivariate_prediction(model, data, scalers, feature_scalers=None, future_day=1):
    # Define the number of time steps to consider in the input data for the prediction
    time_steps = 60

    # Select the last 'time_steps' number of rows from the data
    # This forms the initial input window for the prediction
    input_data = data[-time_steps:]

    # Initialize a list to store the predictions for each future day
    predictions = []

    # Loop to make predictions for the specified number of future days
    for _ in range(future_day):
        # Reshape the input data to match the expected input shape for the model
        # Shape (1, time_steps, number_of_features)
        input_data_reshaped = input_data.reshape((1, time_steps, input_data.shape[1]))

        # Make a prediction using the model
        prediction = model.predict(input_data_reshaped)

        # If feature scalers are provided, use them to inverse transform the prediction
        if feature_scalers:
            prediction = feature_scalers["Close"].inverse_transform(prediction)
        else:
            # Otherwise, use the provided scalers to inverse transform the prediction
            prediction = scalers["Close"].inverse_transform(prediction)

        # Append the prediction (for the 'Close' value) to the list of predictions
        predictions.append(prediction[0])

        # Create the next row to append to the input data
        # Initialize an array of zeros with the same number of features as input_data
        next_row = np.zeros(input_data.shape[1])

        # Set the first element (assumed to be 'Close' value) to the predicted value
        next_row[0] = prediction[0, 0]

        # Shift the input data window by removing the first row and appending the new prediction
        input_data = np.append(input_data[1:], next_row.reshape(1, -1), axis=0)

    # Return the array of predictions
    return np.array(predictions)
```

Figure 9: Code to create multivariate prediction.

```python
def multistep_multivariate_prediction(model, data, scalers, feature_scalers=None, k=5):
    time_steps = 60
    input_data = data[-time_steps:]
    predictions = []

    for _ in range(k):
        input_data_reshaped = input_data.reshape((1, time_steps, input_data.shape[1]))
        next_prediction = model.predict(input_data_reshaped)

        if feature_scalers:
            next_prediction = feature_scalers["Close"].inverse_transform(next_prediction)
        else:
            next_prediction = scalers["Close"].inverse_transform(next_prediction)

        predictions.append(next_prediction[0])

        # Create the next row with the predicted Close value
        next_row = np.zeros(input_data.shape[1])
        next_row[0] = next_prediction[0, 0]

        # Shift the input data window
        input_data = np.append(input_data[1:], next_row.reshape(1, -1), axis=0)

    return np.array(predictions)
```

Figure 10: Code to combine multistep and multivariate prediction.

- Multistep prediction:

+ The "multistep_prediction" function forecasts several future time steps via recursive forecasting. With this approach, each future prediction is produced one step at a time, with the prior prediction serving as the basis for the subsequent prediction.

- In a loop that runs k times (where k is the number of future steps to predict), the function reshapes the input data to match the model's expected input shape.
- To keep the same window size, the oldest value in the input data is deleted and the projected value is attached to the input data.
- Using the reshaped input data, the model predicts the value of the next step.

- Multivariate prediction:

+ Multiple feature prediction at once is handled by the "multivariate_prediction" function. This method provides an improved prediction model by taking into account the interdependencies between multiple features.

- The input data is reshaped to match the model's expected input shape, which includes the number of time steps and the number of features.
- The last "time_steps" points are selected from the dataset by the function to prepare the input data. There are several characteristics in this input data.
- The model predicts the future value(s) based on the reshaped input data.

- Multistep Multivariate Prediction:

+ Combining the ideas of multistep and multivariate predictions, the "multistep_multivariate_prediction" function makes predictions for several future time steps while taking into account different features.

- In a loop that runs k times (where k is the number of future steps to predict), the function reshapes the input data to match the model's expected input shape.
- The model predicts the next time step's value(s) based on the reshaped input data.
- To keep a constant window size, the projected values are added to the input data and the oldest values are removed.
- When there are several features, the function makes sure that the interdependencies between the features are maintained and the input data is updated properly.

VII. Main script run:

```python
# Example usage
if __name__ == "__main__":
    ticker = 'AMZN'
    start_date = '2016-01-01'
    end_date = '2024-03-20'
    local_file = 'amzn_data.csv'
    columns_to_scale = ["Volume","Close"]
    train_data, test_data, scalers, original_data, scaled_data = load_and_process_data(
        ticker=ticker,
        start_date=start_date,
        end_date=end_date,
        local_file=local_file,
        split_ratio=0.7,
        columns_to_scale=columns_to_scale
    )
    display_custom_table(scaled_data, num_rows=5)
    layers_configs = [
        {"layers": [{"type": "LSTM", "units": 50, "return_sequences": True}, {"type": "LSTM", "units": 50}]}
    ]
    results = experiment_with_models(train_data, test_data, scalers["Close"], layers_configs, epochs=50, batch_size=32)

    key_value = 5 # You can change this to any number of future steps you want to predict

    # Multistep prediction example
    multistep_preds = multistep_prediction(results[0]['model'], test_data, scalers["Close"], k=key_value)
    print("Multistep Predictions for next 5 days:", multistep_preds)

    # Multivarate prediction example
    multivariate_preds = multivariate_prediction(results[0]['model'], test_data, scalers, feature_scalers=scalers, future_day=1)
    print("Multivariate Prediction for the next day:", multivariate_preds)

    # Combine multistep and multivarate prediction example
    multistep_multivariate_preds = multistep_multivariate_prediction(results[0]['model'], test_data, scalers, feature_scalers=scalers, k=key_value)
    print("Multistep Multivariate Predictions for next 5 days:", multistep_multivariate_preds)
```

Figure 11: The script to run the code and the prediction.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step
Multistep Predictions for next 5 days: [[175.4778 ]
 [175.45757]
 [175.44368]
 [175.43896]
 [175.44455]]
```

Figure 12: The multistep prediction results for the next 5 days.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step
Multivariate Prediction for the next day: [[175.4778]]
```

Figure 13: The multivariate prediction results for the next day.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
Multistep Multivariate Predictions for next 5 days: [[1.7547780e+02]
 [2.2509561e+03]
 [2.5619588e+04]
 [3.3234028e+05]
 [4.2339560e+06]]
```

Figure 14: The multistep multivariate prediction results for the next 5 days.

VIII. References:

HURSON, T. (2021). Stock Price Prediction with LSTM/Multi-Step LSTM. https://www.kaggle.com/code/thibauthurson/stock-price-prediction-with-lstm-multi-step-lstm