Task Report Cos30018 Option B

B.6: Machine Processing 3

Name: Le Bao Nguyen

Student Id: 104169837

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

```
[1] 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 statsmodels.tsa.arima.model import ARIMA
    from sklearn.metrics import mean_squared_error
    from IPython.display import display, HTML
    import warnings
    from google.colab import drive

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

    warnings.filterwarnings("ignore")  # Suppress warnings

Mounted at /content/drive
```

Figure 1: Importing libraries to run the code.

- The script imports previous libraries and new libraries:
- "statsmodels.tsa.arima.model.ARIMA": Provides the implementation of the ARIMA (AutoRegressive Integrated Moving Average) model for time series forecasting.
- "sklearn.metrics.mean_squared_error": Used to calculate the mean squared error between the predicted and actual values in order to assess the effectiveness of regression models.
- "warnings": A module to manage the warnings that are produced when the code runs.

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

```python
# 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).

```python
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=10, batch_size=16):
    results = []
    time_steps = 60
    input_shape = (time_steps, 1)
    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:
        print(f"Training model with config: {config}")
        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)
        print(f"Model training completed for config: {config}")
        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))
        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')
        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)
```

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

```python
    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:
        print(f"Training model with config: {config}")
        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)
        print(f"Model training completed for config: {config}")
        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))
        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')
        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({
            "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.

VI. Arima Predictions:

```
[6] def fit_arima_model(train_data, test_data, scaler, order=(5,1,0)):
        series = pd.Series(train_data.flatten())
        X = series.values
        history = [x for x in X]
        predictions = list()
        for t in range(len(test_data)):
            model = ARIMA(history, order=order)
            model_fit = model.fit()
            output = model_fit.forecast()
            yhat = output[0]
            predictions.append(yhat)
            obs = test_data[t, 0]
            history.append(obs)
        predictions = np.array(predictions).reshape(-1, 1)
        return scaler.inverse_transform(predictions)
```

Figure 8: Code to fit arima model.

- The ARIMA (AutoRegressive Integrated Moving Average) model is a popular time series forecasting technique that makes predictions about future points in a series using historical data.
- Order parameters:
  - P: The number of lag observations included in the model (autoregressive part).
  - D: The number of difference analyses (integrated part) performed on the raw observations.
  - Q: The size of the moving average window (moving average part).
- Implementation:
  - Convert the training data into a pandas series.
  - Flatten the series values to make it a 1D array.
  - Initialize the values from the training data into a list called "history".
  - Make an empty list called "predictions" that contains the expected values.
  - Fit the "history" data to an ARIMA model.

- Applying the fitted model, predict the next value.
- Add the predicted value (yhat) to the list of predictions.
- For the following iteration, add the real observation (obs) from the test data to "history".
- Restructure and convert the prediction list to a numpy array.
- To inversely transform the predictions back to the original scale, use the scaler.

VII. Ensemble Predictions:

```
def ensemble_predictions(dl_predictions, arima_predictions, weights=[0.5, 0.5]):
    min_len = min(len(dl_predictions), len(arima_predictions))
    ensemble_preds = (weights[0] * dl_predictions[:min_len]) + (weights[1] * arima_predictions[:min_len])
    return ensemble_preds
```

Figure 9: Code to create ensemble predictions.

- With ensemble methods, the final prediction is generated by combining the predictions of several models. The goal is to minimize each model's problems while improving its strengths. We integrated the ARIMA model with predictions from deep learning algorithms for our work.
- Why we use ensemble predictions:
  - Deep learning models: These models are effective at identifying complex patterns and trends in data, but at specific volatile times, they may overfit or perform worse.
  - ARIMA model: This model may not be as good at modeling non-linear patterns as it is at capturing linear dependencies and trends, which makes it difficult to overfitting.
  - We hope to get more accurate and dependable forecasts by combining the two models in a way that balances each of their strengths and limitations.
- Implementation:

- Define the function using weights for averaging, ARIMA predictions, and deep learning prediction parameters.
- Determine the minimum length of predictions to ensure both arrays align correctly.
- Take a weighted average of the ARIMA and deep learning predictions to get the ensemble predictions. The weights parameter gives you more freedom in deciding how much weight to give each model's predictions.
- Return the combined predictions.

VIII. Main script run:

```
# Example usage
if __name__ == "__main__":
    ticker = 'TSLA'
    start_date = '2016-01-01'
    end_date = '2024-03-20'
    local_file = 'tsla_data.csv'
    columns_to_scale = ["Close", "Volume"]
    train_data, test_data, scalers, original_data, scaled_data = load_and_process_data(
        ticker, start_date, end_date, local_file, columns_to_scale=columns_to_scale)

    display_custom_table(scaled_data, num_rows=5)

    lstm_config = {
        "layers": [
            {"type": "LSTM", "units": 50, "return_sequences": True},
            {"type": "LSTM", "units": 50, "return_sequences": False},
            {"type": "Dense", "units": 1}
        ]
    }

    gru_config = {
        "layers": [
            {"type": "GRU", "units": 50, "return_sequences": True},
            {"type": "GRU", "units": 50, "return_sequences": False},
            {"type": "Dense", "units": 1}
        ]
    }

    rnn_config = {
        "layers": [
            {"type": "RNN", "units": 50, "return_sequences": True},
            {"type": "RNN", "units": 50, "return_sequences": False},
            {"type": "Dense", "units": 1}
        ]
    }
```

Figure 10: The script to run the code and the prediction (1).

```python
}

rnn_config = {
    "layers": [
        {"type": "RNN", "units": 50, "return_sequences": True},
        {"type": "RNN", "units": 50, "return_sequences": False},
        {"type": "Dense", "units": 1}
    ]
}

layers_configs = [lstm_config, gru_config, rnn_config]
dl_results = experiment_with_models(train_data, test_data, scalers['Close'], layers_configs, epochs=50)

print("Fitting ARIMA model...")
best_dl_model = dl_results[0]  # Assuming LSTM is the best performing for simplicity
dl_predictions = best_dl_model["predicted_stock_price"]
real_stock_price = best_dl_model["real_stock_price"]

arima_predictions = fit_arima_model(train_data, test_data, scalers['Close'], order=(5, 1, 0))
arima_predictions = arima_predictions[:len(real_stock_price)]  # Trim ARIMA predictions to match the length

ensemble_preds = ensemble_predictions(dl_predictions, arima_predictions, weights=[0.7, 0.3])

plt.figure(figsize=(14, 5))
plt.plot(real_stock_price, color='red', label='Real Stock Price')
plt.plot(dl_predictions, color='blue', label='LSTM Predictions')
plt.plot(arima_predictions, color='green', label='ARIMA Predictions')
plt.plot(ensemble_preds, color='purple', label='Ensemble Predictions')
plt.title('Stock Price Prediction: LSTM vs ARIMA vs Ensemble')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

Figure 11: The script to run the code and the prediction (2).

```
Fitting ARIMA model...
```

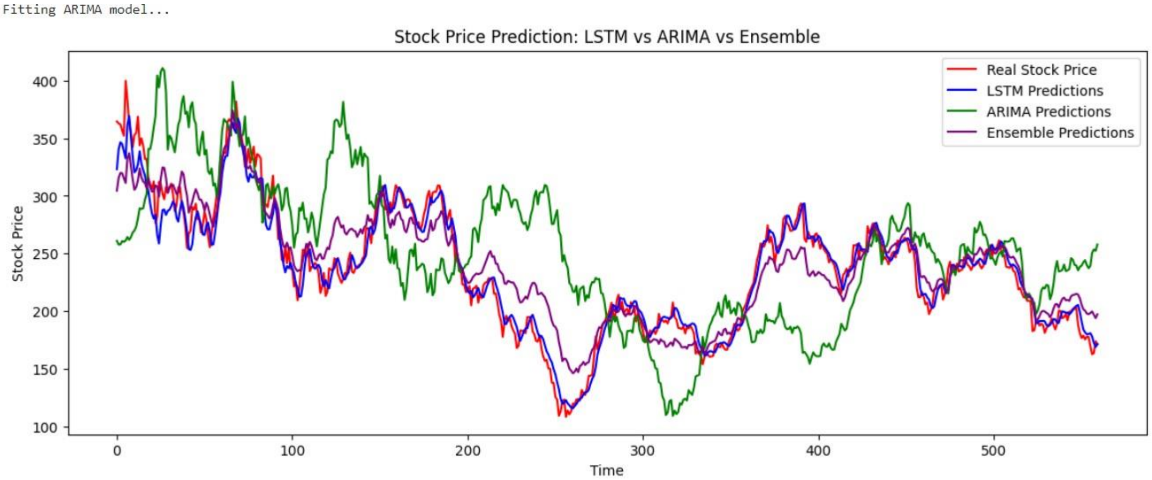

Figure 12: The arima and ensemble predictions (with LSTM).

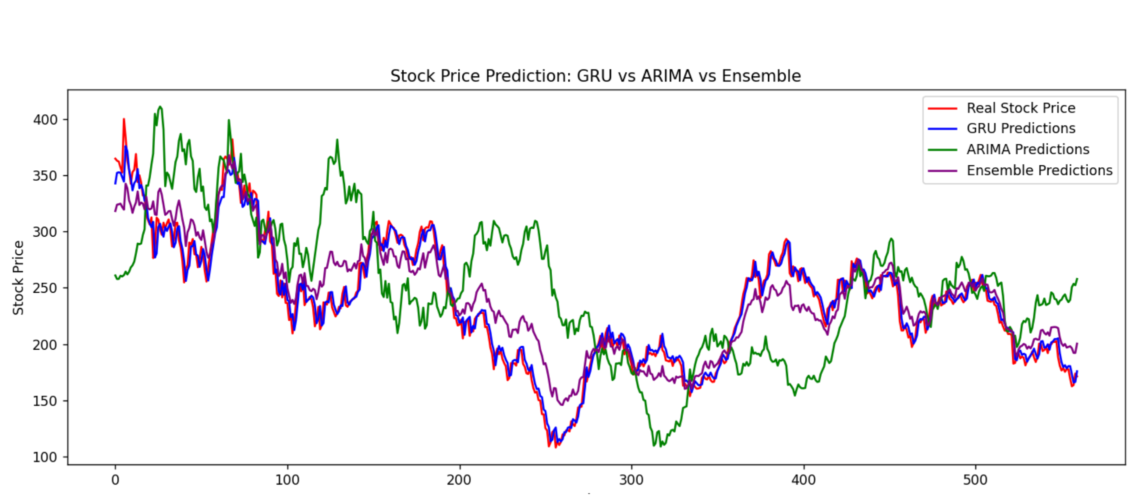

Figure 13: The arima and ensemble predictions (with GRU).

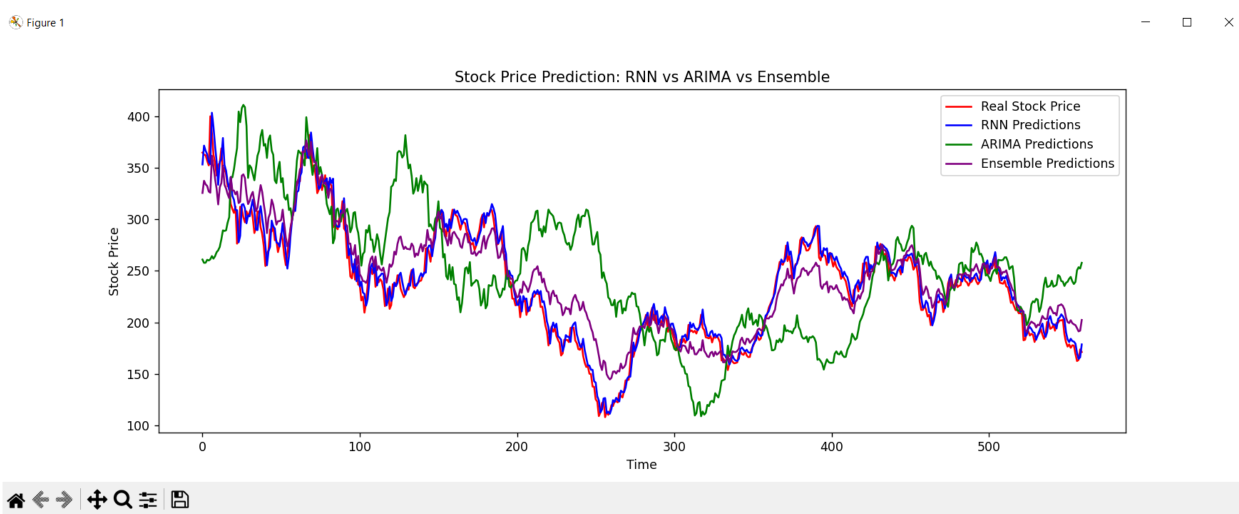

Figure 14: The arima and ensemble predictions (with RNN).

IX. References:

RAZA, M. (2020). 1CAC 40 Stock Price Forecast with ARIMA & LSTM. https://www.kaggle.com/code/razamh/1cac-40-stock-price-forecast-with-arima-lstm?fbclid=IwY2xjawEQoChleHRuA2FlbQIxMAABHURYRNO29-P9jXI1fUxeQb70JPGjKfPS9VpmFBDA3Rbw8tJqAUDe1sTiog_aem_deYZIQC-B71qaA0j4gVFyQ