Task Report Cos30018 Option B

B.7: Extension

Name: Le Bao Nguyen

Student Id: 104169837

I. Install pytrends library:

```
!pip install pytrends

Requirement already satisfied: pytrends in /usr/local/lib/python3.10/dist-packages (4.9.2)
Requirement already satisfied: requests>=2.0 in /usr/local/lib/python3.10/dist-packages (from pytrends) (2.31.0)
Requirement already satisfied: pandas>=0.25 in /usr/local/lib/python3.10/dist-packages (from pytrends) (2.0.3)
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from pytrends) (4.9.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25->pytrends) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25->pytrends) (2024.1)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25->pytrends) (2024.1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25->pytrends) (1.25.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests>=2.0->pytrends) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.0->pytrends) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2.0->pytrends) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2.0->pytrends) (2024.7.4)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=0.25->pytrends) (1.16.0)
```

Figure 1: Downloading libraries to run the code.

II. Importing libraries (From the task B.2 + new libraries):

```
import numpy as np
import pandas as pd
import yfinance as yf
from pytrends.request import TrendReq
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
import matplotlib.pyplot as plt
from IPython.display import display, HTML
import warnings
import time
from google.colab import drive
import os

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

warnings.filterwarnings("ignore")  # Suppress warnings

Mounted at /content/drive
```

Figure 2: Importing libraries to run the code.

- The script imports previous libraries and new libraries:
- "pytrends:": A Google Trends API that lets you get trending search data straight from the source.
- "time": Provides various time-related functions, including delays.
- "os": A module that gives users access to operating system-dependent features like file system reading and writing.

III. Fetch google trends data:

```python
def fetch_google_trends_data(ticker, start_date, end_date, retries=5, delay=60):
    pytrends = TrendReq(hl='en-US', tz=360)
    pytrends.build_payload([ticker], cat=0, timeframe=f'{start_date} {end_date}', geo='', gprop='')
    for i in range(retries):
        try:
            trends_data = pytrends.interest_over_time()
            trends_data = trends_data.drop(columns='isPartial')
            return trends_data
        except Exception as e:
            if i < retries - 1:
                print(f"Rate limit exceeded. Retrying in {delay} seconds...")
                time.sleep(delay)
            else:
                raise e
```

Figure 3: Fetching google trends data.

- Purpose: Gets Google Trends data for a ticker within a particular time frame.
- Details:
  - TrendReq: Initializes a request object for Google Trends.
  - build_payload: Builds the API request payload using the given parameters.
  - Retries loop: Attempts to fetch the data multiple times in case of rate limit issues, with a delay between retries.
  - interest_over_time: Retrieves statistics on interest over time for the given search phrase.
  - drop(columns='isPartial'): Removes the 'isPartial' column, which indicates if the data is incomplete for the current period.

IV. Data loading and processing (From the task B.2 + add pytrends):

```python
def load_and_process_data(ticker, start_date, end_date, local_file=None, split_ratio=0.7, split_by_date=False, columns_to_scale=None):
    data = yf.download(ticker, start=start_date, end=end_date)
    data.index = pd.to_datetime(data.index)
    data.fillna(method='ffill', inplace=True)

    # Add Google Trends data
    trends_cache_file = f"/content/drive/My Drive/Cos30018/{ticker}_trends.csv"
    if local_file and os.path.exists(trends_cache_file):
        trends_data = pd.read_csv(trends_cache_file, index_col='date', parse_dates=True)
    else:
        trends_data = fetch_google_trends_data(ticker, start_date, end_date)
        trends_data.to_csv(trends_cache_file)

    data = data.join(trends_data)

    # Drop the Ticker column
    data = data.drop(columns={ticker})

    if columns_to_scale is None or not columns_to_scale:
        columns_to_scale = ['Close']

    scaled_data = data.copy()
    scalers = {}
    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

    close_prices = data['Close'].values.reshape(-1, 1)
    close_scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_close_prices = close_scaler.fit_transform(close_prices)
    scalers['Close'] = close_scaler

    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 4: Loading and processing data (1).

```python
    scaled_data = data.copy()
    scalers = {}
    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

    close_prices = data['Close'].values.reshape(-1, 1)
    close_scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_close_prices = close_scaler.fit_transform(close_prices)
    scalers['Close'] = close_scaler

    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

    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):]

    if local_file:
        file_path = f"/content/drive/My Drive/Cos30018/{local_file}"
        data.to_csv(file_path)

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

Figure 5: Loading and processing data (2).

- Purpose: Loads stock data, adds Google Trends data, scales selected columns, and splits data into training and testing sets.

- Details:
  - yf.download: Downloads historical stock data from Yahoo Finance.
  - fillna(method='ffill'): Fills any missing values using forward fill method.
  - fetch_google_trends_data: Gets data from Google Trends and saves it locally to avoid repeated downloads.
  - Join: Combines Google Trends data with stock data.
  - MinMaxScaler: Scales selected columns to a range between 0 and 1.
  - split_by_date: Splits the data into testing and training sets using a ratio or an assigned date.

V. 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 6: Displaying the data from csv file.

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

VI. Displaying trend data:

```
1 giây
[13] def display_trend_data(trends_data):
         display(trends_data)
```

Figure 7: Displaying the trend data from csv file.

- Displays the DataFrame with Google Trends data using IPython's "display" method.

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

```python
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 8: Code to create the model.

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

VIII. 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 9: 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 10: Code to experiment with model (2).

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

IX. Arima Predictions:

```python
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 11: 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.

X. Ensemble Predictions (Combine with pytrends):

```
def ensemble_predictions(dl_predictions, arima_predictions, weights=[0.5, 0.5]):
    max_len = max(len(dl_predictions), len(arima_predictions))
    ensemble_preds = np.zeros((max_len, 1))
    for i in range(max_len):
        dl_pred = dl_predictions[i] if i < len(dl_predictions) else 0
        arima_pred = arima_predictions[i] if i < len(arima_predictions) else 0
        ensemble_preds[i] = (weights[0] * dl_pred) + (weights[1] * arima_pred)
    return ensemble_preds
```

Figure 12: Code to create ensemble predictions (with pytrends).

- Return the combined predictions combining predictions from several models to enhance performance as a whole. This method makes use of the advantages of several models to provide a forecast that is more reliable and accurate.
- In order to record public interest and market attitude, which may be a good indicator of changes in stock prices, additional external data was included, such as Google Trends.

XI. Main script run:

```python
if __name__ == "__main__":
    ticker = 'TSLA'
    start_date = '2016-01-01'
    end_date = '2024-03-20'
    local_file = 'tsla_data.csv'
    columns_to_scale = ["Close"]
    train_data, test_data, scalers, data, scaled_data = load_and_process_data(ticker, start_date, end_date, local_file, split_ratio=0.7, split_by_date=False, columns_to_scale=columns_to_scale)
    display_custom_table(data)

    # Load trend data separately and display it
    trends_cache_file = f"/content/drive/My Drive/Cos30018/{ticker}_trends.csv"
    trends_data = pd.read_csv(trends_cache_file, index_col='date', parse_dates=True)
    display_trend_data(trends_data)

    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}
        ]
    }

    layers_configs = [lstm_config, gru_config, rnn_config]
```

Figure 13: The script to run the code and the prediction (1).

```python
    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}
        ]
    }

    layers_configs = [lstm_config, gru_config, rnn_config]

    results = experiment_with_models(train_data, test_data, scalers['Close'], layers_configs, epochs=50, batch_size=16)
    arima_predictions = fit_arima_model(train_data, test_data, scalers['Close'])

    ensemble_preds = ensemble_predictions(results[0]['predicted_stock_price'], arima_predictions)
    plt.figure(figsize=(14, 5))
    plt.plot(scalers['Close'].inverse_transform(test_data), color='red', label='Real Stock Price')
    plt.plot(ensemble_preds, color='green', label='Ensemble Predicted Stock Price (With Google trends)')
    plt.title(f'Ensemble Stock Price Prediction (With Google trends data)')
    plt.xlabel('Time')
    plt.ylabel('Stock Price')
    plt.legend()
    plt.show()

    mse = mean_squared_error(scalers['Close'].inverse_transform(test_data), ensemble_preds)
    print(f"Mean Squared Error of the Ensemble Model: {mse}")
```

Figure 14: The script to run the code and the prediction (2).

date	TSLA
2016-01-01	5
2016-02-01	9
2016-03-01	8
2016-04-01	13
2016-05-01	8
...	...
2023-11-01	29
2023-12-01	26
2024-01-01	34
2024-02-01	32
2024-03-01	32

99 rows × 1 columns

Figure 15: The pytrends data table.

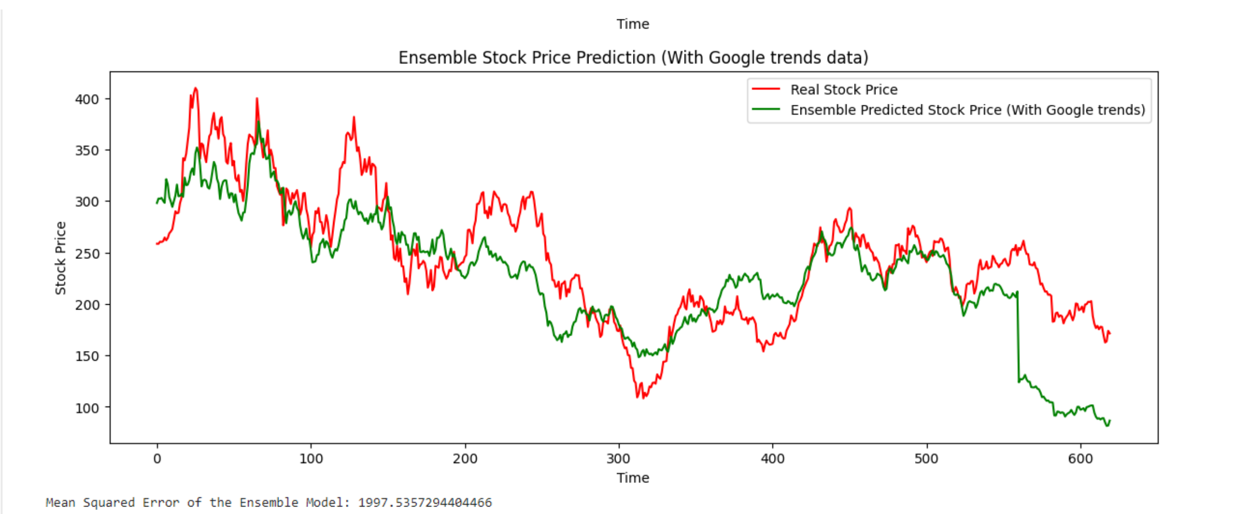

Figure 16: The ensemble predictions (with Google trends data).

XII. References:

ADEP, V. (2021). *Google Trends using Python*. https://www.kaggle.com/code/adepvenugopal/google-trends-using-python

GUTIÉRREZ, J. L. R. (2022). Get Google Trends Data using Pytrends. https://www.kaggle.com/code/luisresendiz/get-google-trends-data-using-pytrends