Task B2: Data Processing 1

Conducted by:

Group 1

Team Members	*Email*
Thanh Tam Vo	103487596@student.swin.edu.au
Phuc Khai Hoan Cao	103804739@student.swin.edu.au
Tai Minh Huy Nguyen	104220352@student.swin.edu.au

March 5, 2024

Table of Contents

1 Introduction

In the Task B2: Data Processing 1, the main goal is writing a function to load the data in a efficient way. It must satisfiy these requirements:

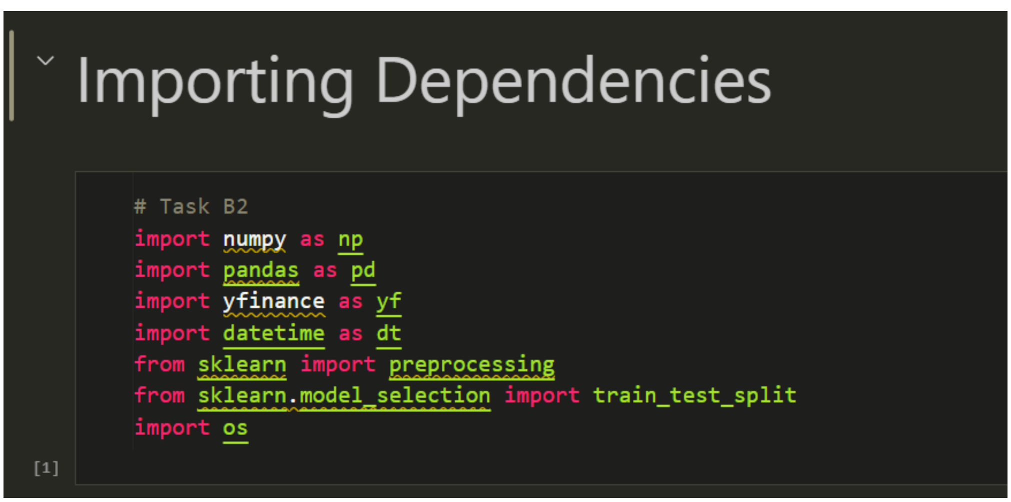
- The user can specify the start date and end date before download the stock data
- The function should handle the NaN values of the stock data
- The user can specify a ratio to split the dataset to train/test sets
- After loading the data, the local machine should store the stock data for further manipulation
- There will be an option to allow the user to scale the data before training

2 Importing Dependencies

There are several packages that are need to be imported:

- **numpy:** to manipulate the array
- **pandas:** to manipulate the dataframe
- **yfinance:** a Python library that provides access to financial data from Yahoo Finance
- **datetime:** to manipulate the datetime datatype
- **sklearn.preprocessing:** A defined function from `scikit-learn` to transform the data
- **sklearn.model_selection.train_test_split:** A defined function from `scikit-learn` to split the data

The Figure 1 below is what we did in the file **TaskB2.ipynb**


Importing Dependencies

```
# Task B2
import numpy as np
import pandas as pd
import yfinance as yf
import datetime as dt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import os
```

[1]

Figure 1: Importing Dependencies

3 Hyperparameters

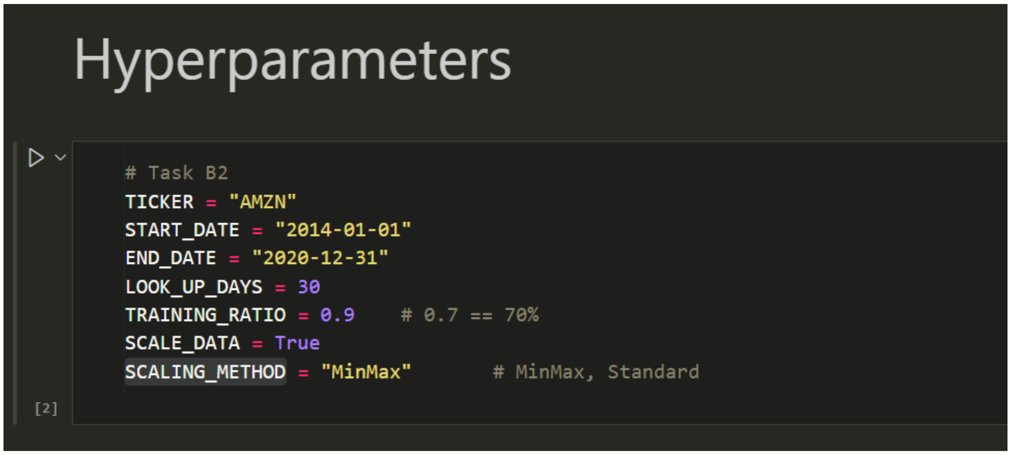


Figure 2: Defining hyperparameters

In the Figure 2 above, we defined all necessary constants as hyperparameters. If the user would like to configure anything, they only need to modify those constants. Below is the description for each constant:

- **TICKER** is the code of the target ticker (In the Figure 2, we took ticker of Amazon as an example)
- **START_DATE** is a start date string with format YYYY/MM/DD
- **END_DATE** is an end date string with format YYYY/MM/DD
- **LOOK_UP_DAYS** is an integer. The train model will look at the records of the last LOOK_UP_DAYS to perform prediction
- **TRAINING_RATIO** is a float ranging from 0 to 1, it indicates the training set ratio among the whole dataset
- **SCALE_DATA** is a boolean value indicating whether the dataset should be scaled
- **SCALING_METHOD** is a string indicating a scaling method (‘MinMax‘, ‘Standard‘)

4 Scaling the dataset

We defined the function **`DataScaler()`** (Figure 3) that takes 2 parameters:

- **`stock_data`** is a stock data **Pandas.DataFrame**
- **`scaling_method`** is the constant `SCALING_METHOD` as default

```python
def DataScaler(stock_data, scaling_method=SCALING_METHOD):

    DatasetScaler = None
    ColumnScalers = {

    }
    if scaling_method == "MinMax":
        DatasetScaler = preprocessing.MinMaxScaler()


    elif scaling_method == "Standard":
        DatasetScaler = preprocessing.StandardScaler()


    # Learn the whole dataset
    col_names = stock_data.columns
    features = stock_data[col_names]
    DatasetScaler.fit(features.values)
    features = DatasetScaler.transform(features.values)
    scaledDataFrame = pd.DataFrame(features, columns = col_names)
    scaledDataFrame.index = stock_data.index


    # Learn each column
    for column in col_names:
        column_scaler = None
        if scaling_method == "MinMax":
            column_scaler = preprocessing.MinMaxScaler()
        elif scaling_method == "Standard":
            column_scaler = preprocessing.StandardScaler()
        column_scaler.fit(stock_data[column].values.reshape(-1,1))
        ColumnScalers[column] = column_scaler

    return scaledDataFrame, DatasetScaler, ColumnScalers
```

Figure 3: `DataScaler()` function

`DataScaler()` will return a tuple `(scaledDataFrame, DatasetScaler, ColumnScalers)` where:

- `scaledDataFrame` is a **Pandas.DataFrame** storing the scaled stock data
- `DatasetScaler` is a an instance of the MinMaxScaler or StandardScaler class. This scaler learned the pattern of the whole dataset
- `ColumnScalers` is a dictionary containing the scaler of each column (of the **Pandas.DataFrame**), each column scaler learned the pattern of the correspond column

4.1 `DatasetScaler = preprocessing.MinMaxScaler()`

This line creates an instance of the MinMaxScaler class, which scales the data to be between 0 and 1. This is supposedly done to make the training process easier and faster, as the netwrok can learn from smaller and normalized values.

4.2 `DatasetScaler = preprocessing.StandardScaler()`

This line creates an instance of the StandardScaler class, which standardizes the features in a dataset. Standardization means transforming the features to have zero mean and unit variance and can improve the performance of many machine learning algorithms. The StandardScaler class can compute the mean and standard deviation of the features on a training set, and then apply the same transformation on the testing set.

4.3 Learn and scale the whole dataset

First, we get all the column names of the Stock Data and invoked the `DatasetScaler` to fit the data. Next, `features` is a numpy array, which is an output of the transforming process from the `DatasetScaler`. Finally, we converted `features` to a **Pandas.DataFrame** and adding index to it.

4.4 Learn each column

We iterated every column, which is a **Pandas.Series**, and invoked the `column_scaler` to fit the pattern. Then, we stored it in the `ColumnScalers` dictionary.

5 Loading the stock data

We defined a function called **`DataLoader()`** that takes a ticker, a start date, an end date, a scale flag, and a scaling method as inputs, and returns a result dictionary that contains the dataset, the dataset scaler, and the column scalers. They are the output of the **`DataScaler()`** function.

The function checks if the data is already downloaded in the **data folder**, and if not, it downloads the data from yfinance. It also filters the data according to the given dates, and sets the date as the index. If the scale flag is True, it calls the DataScaler function defined above to scale the data.

5.1 Preprocessing the input parameters

```python
def DataLoader(ticker=TICKER, start_date=START_DATE, end_date=END_DATE, scale=SCALE_DATA, scaling_method=SCALING_METHOD):
    '''
    ticker: is the code of the target ticker
    start_date: a start date string with format YYYY/MM/DD
    end_date: an end date string with format YYYY/MM/DD
    scale: a boolean value, True by default
    scaling_method: MinMax(by default), Standard.
    '''

    # result
    result = {
        "dataset": None,
        "datasetScaler": None,
        "columnScalers": None
    }

    # processing the input parameters
    start_date = dt.datetime.strptime(start_date, "%Y-%m-%d")
    end_date = dt.datetime.strptime(end_date, "%Y-%m-%d")
```

Figure 4: Preprocessing the date format

5.2 Creating necessary folders

Next, we created two folders:

- **results:** this folder stores the output of the training model
- **data:** this folder stores the stock data

```python
# creating necessary folder
if not os.path.isdir("results"):
    os.mkdir("results")

if not os.path.isdir("data"):
    os.mkdir("data")
```

Figure 5: Creating necessary folders

5.3 Downloading the DataFrame if necessary

```python
# checking if the data is already downloaded
## Get a list of files in the directory
files = os.listdir("data")
## Check each file in the directory
data = None
for file_name in files:
    ## if we already downloaded the ticket data
    if file_name.startswith(ticker) and file_name.endswith(".csv"):
        ### Read the file
        file_path = os.path.join("data", f"{ticker}.csv")
        data = pd.read_csv(file_path, parse_dates=['Date'])
        break

## else, we gonna download the stock data
if data is None:
    stock_data = yf.download(ticker, start_date, end_date)
    file_path = os.path.join("data", f"{ticker}.csv")
    stock_data.to_csv(file_path)
    data = pd.read_csv(file_path, parse_dates=['Date'])
```

Figure 6: Checking if the data is already downloaded

```python
# if the given time is included in the file, we just take the nessecary dataframe
if data.head(1)["Date"].values[0] <= np.datetime64(start_date) and data.tail(1)["Date"].values[0] >= np.datetime64(end_date):
    data = data[(data['Date'] >= pd.to_datetime(start_date)) & (data['Date'] <= pd.to_datetime(end_date))]
    print("Local Stock Data is enough for requirements, do not need to download")
else:
    stock_data = yf.download(ticker, start_date, end_date)
    file_path = os.path.join("data", f"{ticker}.csv")
    stock_data.to_csv(file_path)
    data = pd.read_csv(file_path, parse_dates=['Date'])
    print("Local Stock Data is not enough for requirements, continuing downloading...")

# Setting Date as Index
data.set_index('Date', inplace=True)
```

Figure 7: Downloading the data if necessary

5.4 Scaling the data if required and return result

```python
# Scale Data
if scale:
    data, scaler, column_scalers = DataScaler(data, scaling_method)
    result["dataset"] = data
    result["datasetScaler"] = scaler
    result["columnScalers"] = column_scalers
    return result

result["dataset"] = data

return result
```

Figure 8: Scaling the data if necessary

6 Splitting the Dataset

We then defined a function called **`datasetSplitter()`** that takes a dataset DataFrame, a look up days parameter, a training ratio parameter, and a list of feature columns as inputs, and returns a dictionary that contains the X and Y training and testing sets for each feature column.

The function splits the data into X and Y arrays based on the look up days, and then splits them into training and testing sets based on the training ratio. It also converts the arrays to numpy arrays.

Below is the output of the **`datasetSplitter()`** function:

```
{
        "X_training_set": {
                        'Open': <class 'numpy.ndarray'>,
                        'High': <class 'numpy.ndarray'>,
                        'Low': <class 'numpy.ndarray'>,
                        'Close': <class 'numpy.ndarray'>,
                        'Adj Close': <class 'numpy.ndarray'>,
                        'Volume': <class 'numpy.ndarray'>
                        },
        "Y_training_set": {
                        'Open': ...,
                        ...
                        },
        "X_testing_set": {
                        'Open': ...,
                        'High': ...,
                        ...
                        },
        "Y_testing_set": {
                        'Open': ...,
                        'High': ...,
                        'Low': ...,
                        ...
                        }
    }
```

Figure 9: Traning and testing sets

6.1 Preparing the dataset

The Figure 10 is already explained in the Section **3.2.3 Preparing Data** from the Report of Task B1.

6.2 Splitting the dataset

In Figure 11, we used the built-in function **`train_test_split()`** from **scikit-learn** to split the dataset into training and test sets. More importantly, since we are dealing with time

```
for column in feature_columns:
    dataset_in_column = dataset[column].values.reshape(-1, 1)      # <class 'numpy.ndarray'>
    x_data = []
    y_data = []

    for x in range(look_up_days, len(dataset_in_column)):
        x_data.append(dataset_in_column[x - look_up_days:x, 0])
        y_data.append(dataset_in_column[x, 0])
```

Figure 10: Preparing Data for Training

```
for column in feature_columns:
    dataset_in_column = dataset[column].values.reshape(-1, 1)      # <class 'numpy.ndarray'>
    x_data = []
    y_data = []

    for x in range(look_up_days, len(dataset_in_column)):
        x_data.append(dataset_in_column[x - look_up_days:x, 0])
        y_data.append(dataset_in_column[x, 0])

    splitResult["X_training_set"][column],
    splitResult["X_testing_set"][column],
    splitResult["Y_training_set"][column],
    splitResult['Y_testing_set'][column] = train_test_split(x_data, y_data,
                                             test_size=1-training_ratio, shuffle=False)
```

Figure 11: Splitting the dataset

series data, we must set the **shuffle** attribute to **False**.

6.3 Converting to Numpy Array and return the result

```
        ## Converting to numpy.array

        for column in feature_columns:
            splitResult["X_training_set"][column] = np.array(splitResult["X_training_set"][column])
            splitResult["Y_training_set"][column] = np.array(splitResult["Y_training_set"][column])
            splitResult["X_testing_set"][column] = np.array(splitResult["X_testing_set"][column])
            splitResult["Y_testing_set"][column] = np.array(splitResult["Y_testing_set"][column])

    return splitResult
```

Figure 12: Converting to Numpy Array and returning the result

7 Data_Processing_1()

```
def Data_Processing_1():
    dataLoader = DataLoader()

    scaledStockData = dataLoader["dataset"]
    datasetScaler = dataLoader["datasetScaler"]
    columnScalers = dataLoader["columnScalers"]

    dataset = datasetSplitter(dataset=scaledStockData)

    return dataset, scaledStockData, datasetScaler, columnScalers
```

Figure 13: **Data_Processing_1()** Function

```
dataset, scaledStockData, datasetScaler, ColumnScalers  = Data_Processing_1()
```

Figure 14: Calling **Data_Processing_1()** Function

Finally, We defined a function called **Data_Processing_1()** and prints some fundamental information about the result tuple, which contains the dataset, the scaledStockData, the datasetScaler, and the columnScalers.