Investing in the past: A time travel project using Python

An assignment presented as part of the MSc:
Data Science and Machine Learning

Course: Programming Tools and Technologies for Data Science

School of Electrical and Computer Engineering

Author: Apostolos Moustakis

Student number:

Email:

Instructors:

Table of Contents

List of Figures

1 Introduction

The purpose of this assignment is to produce a sequence of transactions that generate high profit assuming that we have the ability to travel in the past and perform trading on New York's stock exchange. The trading begins from 1/1/1960 with an account that possesses only 1$. With the use of this amount we buy and sell stocks with the aim of generating very high profits.

We are given a dataset that contains 7195 stock files, which is a relatively high number of stocks. Each file is named s.us.txt where s is the code of the respective stock. For instance the code for the Apple's stock is aapl. For each stock the file contains information about the following attributes:

Attribute	Description
Date	Date of the stock's transaction
Open	The opening price of the stock
High	The highest price of the stock during the day
Low	The lowest price of the stock during the day
Close	The closing price of the stock
Volume	Number of stock exchanges

For example the first five rows of the Apple's stock (i.e. aapl) are displayed below. The starting day of the stock's transactions is 07/09/1984 and its opening price is approximately 0.42$, which satisfies the 1$ initial limitation for picking the first stock to start trading. The seventh attribute of each stock, named OpenInt, is not used in the analysis.

```
# A starting point: Apple's stock AAPL
import pandas as pd
df = pd.read_csv('Stocks/aapl.us.txt')
df = df.drop(['OpenInt'], axis=1) # Remove the column OpenInt
df.head() #the first five rows
```

Python Code Snippet 1: A starting point: Apple's stock

	Date	Open	High	Low	Close	Volume
0	1984-09-07	0.42388	0.42902	0.41874	0.42388	23220030
1	1984-09-10	0.42388	0.42516	0.41366	0.42134	18022532
2	1984-09-11	0.42516	0.43668	0.42516	0.42902	42498199
3	1984-09-12	0.42902	0.43157	0.41618	0.41618	37125801
4	1984-09-13	0.43927	0.44052	0.43927	0.43927	57822062

Figure 1: First five rows of AAPL stock

During the trading the two basic moves that we are allowed to make is buy-low and sell high. These moves guarantee that profit can be achieved in a later stage since we are buying a stock at a lower price and trying to sell it at a higher price. During a single day, we do not know if the highest or the lowest price of a stock occurred first and thus only one of these two moves are allowed. Additional moves that can be performed are buy-open, sell-open, buy-close and sell-close. These moves denote that an exchange of a stock can also happen at the opening or the closing price of a stock besides during the day. Therefore, stock exchanges during a day can occur with the following chronological order:

$$\{ \text{buy-open} , \text{sell-open} \} > \{ \text{buy-low} , \text{sell-high} \} > \{ \text{buy-close} , \text{sell-close} \}$$

Of course, in order to buy stocks at any time of a day we need to have the available balance in our account, while in order to sell stocks we need to buy them first. An additional constraint is that during every day the available balance for stock purchasing must not be grater than 1.000.000$. The goal of this assignment is to produce a sequence of movements that generate very high profit. The transaction movements are in the form of (d,m,s,x) where d is the date of the transaction, m is the type of the transaction (e.g. buy-low), s is the stock's code and x is the number of stocks for exchange. In this assignment two such sequences are produced, one small sequence with less than 1.000 transactions and one large sequence with less than 1.000.000 transactions. The programming language used in the whole assignment is Python, where the code is written in a Jupyter notebook.

2 Data Preprocessing

The first part of the process is importing the stock files and selecting stocks to use in our analysis that will eventually generate high profits. As we described earlier, the stock files are 7195, which is a very large amount of stocks. Before finding metrics of selecting such stocks some cleaning to the data must be performed. More specifically for each stock I delete rows that contain negative or zero values in one or more of the columns: Open, High, Low, Close and Volume. Negative values are false data entries for all the columns, while zero values can be valid only for the column Volume (i.e. zero stock transactions during that day) but these rows are not needed. Furthermore, I delete rows where the low value is 100% smaller than the Open and Close Value, as such data entries are probably false due to the unlikeliness of occurrence. Respectively, I also delete rows where the high value is 100% greater than the Open and Close Value. Last, but not least some stock files are empty and thus I decide to ignore them. It is important to notice that I create a list that is called valuable_stocks that stores the name of every stock and its dataframe after the cleaning process.

In order to initially evaluate all the the stocks I iterate through the stock files and create a new dataframe that is called performance_df. In this dataframe I store the name of the stock, the start date of the stock's trading, the end date of the stock's trading, the number of trading days, the initial price of the stock and the performance of the stock in three metrics that I created. The first metric is the average of the difference between the high and the low price multiplied by the volume of the stock. I multiply with the volume as it denotes how competitive a stock is. If during a day I could buy low and sell high and I knew that the high price occurred after the low price then the higher this metric the better the performance of the stock. Note that, as I already described, the moves buy low and sell high cannot be performed during the same day as we do not know which one occurred first. Nevertheless, this metric is valuable as a starting point. A second metric that is created refers to the average of the difference between the high and the open price multiplied by the volume. Respectively, it refers to the moves buy open and sell high that can be performed during a single day and thus this metric is better than the first one. Last but not least, the third metric is the average of the difference between the close and the low price multiplied by the volume, which refers to the moves buy low and sell close that can also be performed during a single day.

When the performance dataframe is created I sort the values regarding Metric1 with ascending order as the higher values denote better performance. I also add an additional constraint that the number of trading days must be over 100 as generally we want the stocks that will be selected to be trading for at least some time. The first ten rows of this dataframe are presented below.

	Stock	Start date	End date	Days traded	Metric1	Metric2	Metric3	Initial Price
2160	FB	2012-05-18	2017-11-10	1381	73785975.0	34832685.9	36655544.0	42.05
11	AAPL	1984-09-07	2017-11-10	8363	55236558.1	25126137.7	28079981.0	0.42388
2642	GOOGL	2004-08-19	2017-11-10	3332	52380793.8	24061680.4	25905027.2	50.0
572	BABA	2014-09-19	2017-11-10	794	50495340.6	23544493.2	25249411.0	92.7
853	BRK-B	1996-05-09	2017-11-10	5415	42501121.8	20781839.0	23695775.1	23.6
1572	DAL	1980-01-02	2017-11-10	9548	34737877.3	17276481.9	17002151.0	1169.22
4016	MSFT	1986-03-13	2017-11-10	7982	33361837.8	16872387.2	16752639.7	0.0672
327	AMZN	1997-05-16	2017-11-10	5152	32707900.6	16723180.5	17309251.8	1.97
1425	CSCO	1990-03-26	2017-11-10	6962	32355122.6	15880218.7	16084397.0	0.06599
6100	TWTR	2013-11-07	2017-11-10	1011	32340410.7	16176637.7	15151168.0	45.1

Figure 2: Top 10 stocks based on performance metrics

The first three stocks regarding the performance are FB, APPL and GOOGL that belong to three of the largest companies in the world, which are Facebook, Apple and Google respectively. Notice that if the performance dataframe was sorted regarding Metric2 or Metric3 the results would be almost the same (no changes in the first eight rows of the outcome).

Other factors can also play a significant role for the stock selection. For instance, since we start trading with only 1$ the sooner we start the better, as higher profit can be generated later. For this reason we can also choose the stock GE that starts trading on 1962 and is 49th at the performance dataframe, which is very high with respect to the number of stocks. We will definitely select AAPL as the trading starts on 1984 at a very low opening price and it is second overall in the performance dataframe.

The Python code for the data preprocessing is presented below.

```
import glob
filenames = glob.glob('Stocks' + '/*.txt') #for reading the file stocks

#First step: Clean the data and find some stocks with good performance
performance_df = pd.DataFrame(columns = ['Stock', 'Start date' , 'End date' , 'Days
                      traded', 'Metric1','Metric2', 'Metric3', 'Initial Price'], )
index = 0
valuable_stocks = [] #create list for the stocks and dataframes

for filename in filenames:
    try:

        df = pd.read_csv(filename) #the file of the stock
        stock = filename.split('\\')[-1].split('.')[0].upper() #name of the stock in
                                                               capital letters
        df = df.drop(['OpenInt'], axis=1) #drop the column OpenInt as it is not needed

        #delete rows that have negative or zero values
        df= df[(df['Open'] > 0) & (df['High'] > 0) & (df['Low'] > 0) & (df['Close'] > 0)
               & (df['Volume'] > 0)]

        #outlier detetction: delete rows where the low value is 100% smaller than the
               Open and Close Value
        df = df[~(df['Low']<df['Open']*0.5)&~(df['Low']<df['Close']*0.5)]

        #outlier detetction: delete rows where the high value is 100% greater than the
               Open and Close Value
        df = df[~(df['High']>df['Open']*2)&~(df['High']>df['Close']*2)]

        df['Stock name'] = stock #add the column stock name to the dataframe - we need it
                                 later

        #construct the perfomrance dataframe for each stock
        performance_df.loc[index, 'Stock'] = stock
        performance_df.loc[index, 'Start date'] = df['Date'].min()
        performance_df.loc[index, 'End date'] = df['Date'].max()
        performance_df.loc[index, 'Days traded'] = len(df['Date'])
        performance_df.loc[index, 'Metric1'] = round(((df['High']-
                                    df['Low'])*df['Volume']).mean(),1)
        performance_df.loc[index, 'Metric2'] = round(((df['High']-
                                    df['Open'])*df['Volume']).mean(),1)
        performance_df.loc[index, 'Metric3'] = round(((df['Close']-
                                    df['Low'])*df['Volume']).mean(),1)
        performance_df.loc[index, 'Initial Price'] = df['Open'][0]

        valuable_stocks.append((stock,df)) #append name of stock and its dataframe

        index+=1 #increase the index for the next stock

    except:  #some stock files are completely empty
        pass

#Sort the stocks based on performance regarding Metric1
performance_df_sorted = performance_df.sort_values('Metric1', ascending = False)
 #Days traded threshold: at least 100
performance_df_sorted = performance_df_sorted[performance_df_sorted['Days traded'] > 100]
performance_df_sorted.head(10) #view the top 10 stocks
```

Python Code Snippet 2: Data Preprocessing

3 Stock Trading Algorithm: Initial Approach

The algorithm that I am going to present in this chapter refers to the idea of taking advantage of only intra-day trading without keeping any stocks at the end of the day. In order to do so I need to find a way to keep only a single stock per day, ideally the one with the highest profits. Therefore, I implement the following steps: Firstly I create a dataframe that contains the values of the first 100 stocks based on the dataframe performance_df_sorted. This dataframe is called best_performance_df and contains 481.669 rows. I create the column High - Open which refers to the profit of a stock in a single day if I buy open and sell high and sort the dataframe in ascending order regarding the date and descending order regarding the High-Open column. Then for each date I keep the first row, which is the one with the highest profit (assuming I perform buy-open and sell-high). Consequently, the final dataframe contains 14.064 rows of 14.064 distinct days with a single stock in each day.

In this algorithm the best moves that can occur during a single day are either buy open and sell high combined or buy low and sell close combined. These two combinations of moves cannot be performed during the same day as we do not know if the high price occurred before the low price. In order to choose between these two combinations I check which difference is greater in order to generate the highest profit that day (i.e. High - Open or Close - Low). More specifically, I first start by creating a list to save the transactions and setting the balance to 1$, as this is our initial balance. Then I iterate throw the rows of the best_performance_df and for each row (i.e. date) I perform the following actions. If the difference of High and Open is greater than the difference of Close and Low I perform the moves buy open and sell high. In order to buy stocks I check that the available balance is greater than the opening price. In order to determine how many stocks to buy I perform integer division between the balance and the opening price as I want to buy as many stocks I can. In this step I add the additional constraint that the maximum balance for purchasing stocks is 1.000.000$ and thus when this balance is reached I buy stocks that cost approximately 1 million dollars (integer division). Subsequently, I add the two transactions that are performed in the transactions list and modify the balance respectively. Each transaction is in the form: Date, movement (i.e. buy-low), name of stock and number of stocks bought or sold accordingly, as requested. Likewise, if the difference of Close and Low is greater I follow the same steps with the distinction that the moves performed are buy low and sell close and thus the balance and the list of transactions are updated accordingly. Lastly, if the difference of High and Open is the same with the difference of Close and Low I choose to perform the moves buy-open and sell-high. In this case I also check that the opening price is lower than the high price because if they are equal unnecessary transactions are performed.

The outcome of this algorithm is profit of approximately 274 million dollars, generated through 24.374 transactions. This algorithm is very simple yet the profit is high. It is important to notice that there is no portfolio of stocks as at the end of each day all the stocks are sold. In the next chapter I will provide an algorithm that takes into account the fact that we can buy stocks and sell them at a later date and thus create a portfolio of unsold stocks. Since this is not the submitted transactions sequence I provide the following screenshots of the solution's validation and the last transactions. Note that the file transactions.txt contains 24.375 rows and not 24.374 as the first row is the number of transactions.

Figure 3: Profit and number of transactions of the initial algorithm

```
24368 2017-11-07 buy-open GOOG 973
24369 2017-11-07 sell-high GOOG 973
24370 2017-11-08 buy-low PCLN 613
24371 2017-11-08 sell-close PCLN 613
24372 2017-11-09 buy-low PCLN 609
24373 2017-11-09 sell-close PCLN 609
24374 2017-11-10 buy-low PCLN 594
24375 2017-11-10 sell-close PCLN 594
```

Figure 4: Last transactions of the initial algorithm

The Python code for the implementation of the algorithm is presented below.

```
#Create a dataframe to store the first 100 stocks based on performance
top100 = performance_df_sorted[:100]
best_performance = [] #list to append each dataframe

for (stock,df) in valuable_stocks:
    if stock in top100.Stock.values:
        best_performance.append(df)

best_performance_df = pd.concat(best_performance) #a single dataframe

#create the column High-Open & sort values based on date and this column
best_performance_df['High-Open']=best_performance_df['High']-best_performance_df['Open']
best_performance_df = best_performance_df.sort_values(['Date','High-Open'], ascending =
                        [True,False])
#keep one row per day - the one with the highest 'High-Open'
best_performance_df = best_performance_df.drop_duplicates(subset=['Date'])
```

Python Code Snippet 3: Creation of the best_performance_df dataframe

```
transactions = [] #list for transactions
balance = 1 #initial balance

for index, row in best_performance_df.iterrows(): #iterate through the rows

    indicator1 = row['High']-row['Open'] # High - Open
    indicator2 = row['Close']-row['Low'] # Close - Low

    if (indicator1 > indicator2): #check which difference is greater

        if (balance > row['Open']):

            result = min(int(balance//row['Open']), int(1000000//row['Open']))
            #result: number of stocks to buy - aditional constraint max 1m for purchasing
            transactions.append((row['Date'],'buy-open',row['Stock name'],str(result)))
            #add the transaction
            balance = balance - result*row['Open'] #buy open - reduce balance
            transactions.append((row['Date'],'sell-high',row['Stock name'],str(result)))
            #add the transaction
            balance = balance + result*row['High'] #sell high - increase balance

    elif (indicator2 > indicator1): #likewise but buy low - sell close

        if (balance > row['Low']):

            result = min(int(balance//row['Low']), int(1000000//row['Low']))
            transactions.append((row['Date'],'buy-low',row['Stock name'],str(result)))
            balance = balance - result*row['Low'] #buy low
            transactions.append((row['Date'],'sell-close',row['Stock name'],str(result)))
            balance = balance + result*row['Close'] #sell close

    elif (indicator1 == indicator2): #likewise: we choose buy open - sell high

        if (row['Open'] < row['High']) and (balance > row['Open']):

            result = min(int(balance//row['Open']), int(1000000//row['Open']))
            transactions.append((row['Date'],'buy-open',row['Stock name'],str(result)))
            balance = balance - result*row['Open'] #buy open
            transactions.append((row['Date'],'sell-high',row['Stock name'],str(result)))
            balance = balance + result*row['High'] #sell high

print(balance) #final profits
print(len(transactions)) #number of total transactions
```

Python Code Snippet 4: Initial algorithm: Perform only intra-day trading

```
#Function to save transactions in a txt file
def save_transactions(tranasctions,filename):
    with open(filename, 'w') as f:
        f.write(str(len(transactions)) + '\n') #number of transactions
        for t in transactions: #each transaction in a row
            f.write(' '.join(t) + '\n')

save_transactions(transactions,'transactions.txt')
```

Python Code Snippet 5: Function for saving transactions in a txt file

4 Stock Trading Algorithm: Final Approach

4.1 Large sequence of 7681 transactions

The algorithm that I am going to present in this chapter is based on the idea that more profit can be generated if we buy stocks at a low price and sell them at a higher price at a later date. Therefore, the move that can be performed during a single day is either buy low or sell high. For simplicity reasons I decided to start the analysis by choosing one stock. The chosen stock is AAPL as it is the second best stock in the performance dataset, begins trading early on (i.e. 1984) and its opening price is less than 1$, which is our initial balance. This algorithm takes into advantage the continuous growth of the AAPL stock during time, which is depicted in the following diagram of the AAPL's Low and High values.

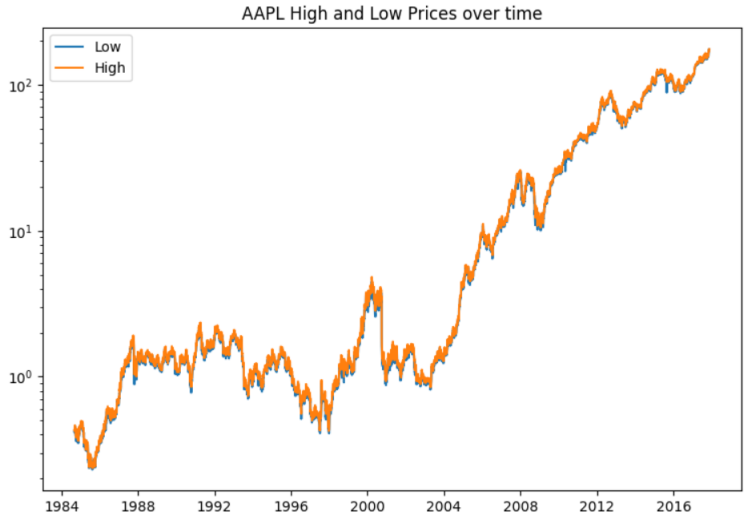

Figure 5: AAPL's Low and High prices over time

```
df = pd.read_csv('Stocks/aapl.us.txt') #Apple's stock
df = df.drop(['OpenInt'], axis=1) #remove column OpenInt
#Plot AAPL's Low and High values over time
fig, ax = plt.subplots()

df['Date']= pd.to_datetime(df['Date'])
ax.plot(df['Date'],df['Low'], label = 'Low') #low prices
ax.plot(df['Date'],df['High'], label = 'High') #high prices
ax.set_title('AAPL High and Low Prices over time') #title
ax.legend() #legend
ax.set_yscale('log') #change the scale of y
plt.show()
```

Python Code Snippet 6: Creation of the diagram: AAPL High and Low prices over time

More specifically the algorithm works as follows. Initially I create the list transactions in order to store the transactions performed, the dictionary portfolio in order to store how many AAPL stocks I have at the end of a day and the lists daily_balance, daily_portfolio and day that are going to be used for plotting the valuation diagram. Moreover, I set the balance to 1, as this is the initial balance, and a boolean variable named Flag to True, the use of which I will explain later on. In the list daily_balance I store the balance at the end of each day and in the list daily_portfolio I store the value of the stocks I possess at the end of each day. The value is defined as the product of the stocks I possess times the closing price of the stocks at the end of a single day.

I iterate through the rows of the dataframe, excluding the last one as it refers to the last day, and perform the following: If the low price of the stock during that day is less than the high price of the stock the next day, the flag is True, the balance is greater than the low price (i.e. I have enough money to buy) and smaller than 1.000.000$, which is the maximum available balance for purchasing stocks at a single day, I buy low the maximum number of stocks that I can. When I do so, I update the balance, the portfolio and the lists transactions, daily_balance, daily_portfolio and day with the correct values, as already described and I set the flag to False. The flag is set to False in order to perform the move sell high the following day. When the sell-high move is performed I update again all the respective fields and set the Flag back to True, as now all the stocks that I had are sold and the next move that I can perform

is only buy-low. In order to perform the move sell-high I also add an additional constraint that the high price the day I want to sell must be greater than the high price of the next day, which is something I will explain later. This is how the algorithm works while the balance remains less than 1 million dollars. When this balance is reached the main difference is that the flag is not set to False, which practically means that while the balance remains over 1 million dollars the only movement that can be performed is buy-low. Of course I buy-low the maximum number of stocks that I can regarding this constraint. In this way I can store a high number of stocks in my portfolio and sell them way later at a much higher price, taking advantage of the continuous growth of the AAPL stock. The additional constraint that the high price must be greater than the high price the next day is added in order to generate more profits when it is time to sell-high. When sell-high is performed all the fields are updated accordingly, with the portfolio at the end of the day set to zero as all the stocks are sold. Lastly, at the last day I sell all the stocks at the high price and update all the fields respectively.

The outcome of this algorithm is profit of approximately 11.2 billion dollars with a total of 7.681 transactions. The generated profit is extremely high regarding the fact that only one stock is traded. The validation of the solution, which constitutes the large sequence, is presented below.

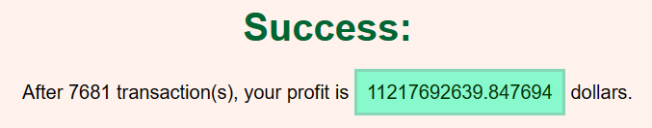

Figure 6: Profit and number of transactions of the large sequence - Proposed Algorithm

Furthermore, I present the valuation diagram for the large transactions sequence. This diagram was created with the use of a dataframe, which stores the date of every transaction, the balance, the portfolio and the total value at the end of the day. The total value is simply the sum of the balance and the portfolio at a given date.

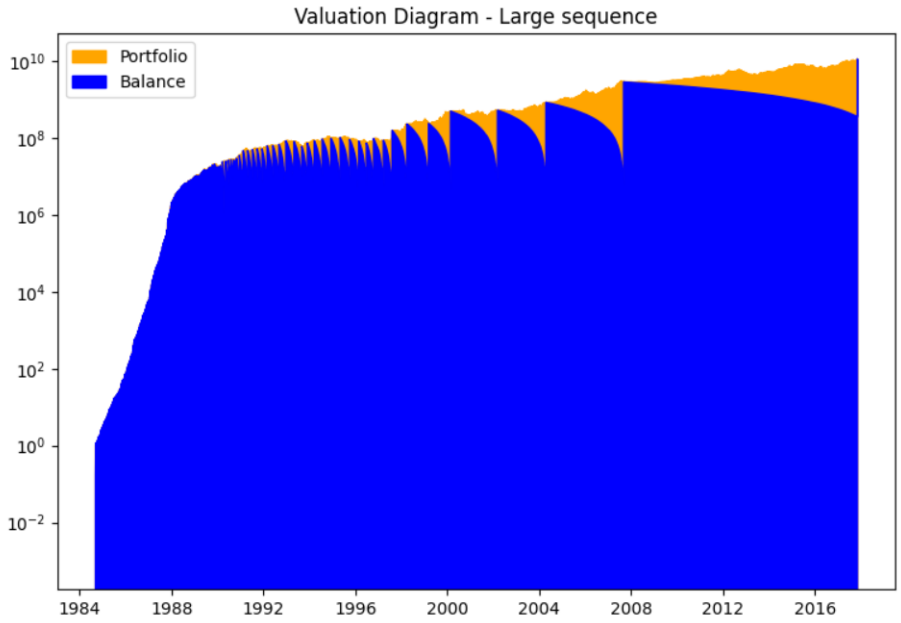

Figure 7: Valuation Diagram - Large sequence of 7.681 transactions

By looking at the diagram one can notice how the algorithm works when the balance reaches 1 million dollars and the portfolio is starting to be built in greater extent. However, this diagram is not such an accurate representation of the relationship between the balance and the portfolio as the date range is very large (i.e. 7681 days). As I described, before the balance reaches 1 million dollars the moves that are performed are buy-low and sell-high, the one after the other. Therefore, the portfolio one day increases and the next day is 0. In the diagram the whole area before the 1 million dollars balance is depicted with blue, which denotes the balance, simply because the diagram spans in so many days that it cannot accurately show the day-to-day relationship between the balance and the portfolio. If we zoom in and

inspect what happens, for instance in the first 500 days, the valuation diagram is the following, which provides a better view.

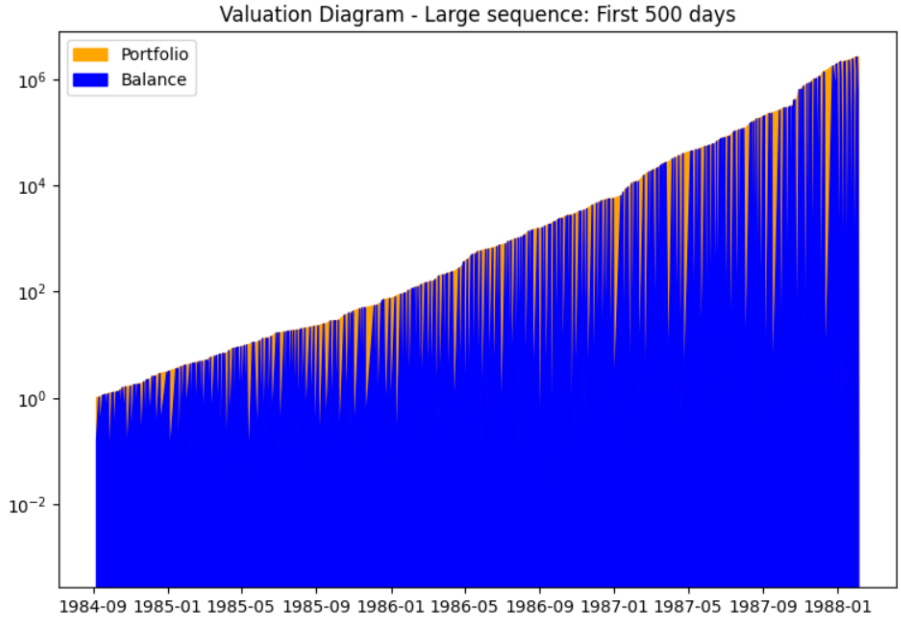

Figure 8: Valuation Diagram - Large sequence: A closer look in the first 500 days

The Python code for the implementation of the algorithm and the creation of the valuation diagrams is presented below.

```
#large sequence < 1.000.000
transactions = [] #list for transactions
portfolio = {'AAPL': 0} #dictionary to "save" the portfolio
balance = 1 #initial balance
flag = True
daily_balance = [] #keep the daily balance for plotting
daily_portfolio = [] #keep the daily portfolio/value for plotting
day = [] #keep the day for plotting

for index, row in df.iterrows(): #iterate through the rows

    if (index < len(df)-1): #all the rows expect the last one

        if (df.iloc[index]['Low'] < df.iloc[index+1]['High']) and (balance > row['Low'])
                        and (flag == True) and (balance < 1000000):

            result = int(balance//row['Low']) #number of stocks to buy
            transactions.append((row['Date'],'buy-low', row['Stock name'],str(result)))
            balance = balance - result*row['Low'] #buy low
            portfolio[row['Stock name']] += result #update the portfolio
            flag = False
            daily_balance.append(balance) #balance of the day
            daily_portfolio.append(row['Close']*portfolio[row['Stock name']]) #value of
                                unsold stocks
            day.append(row['Date']) #the day

        elif (df.iloc[index]['Low'] < df.iloc[index+1]['High']) and (balance >
                            row['Low']) and (flag == True) and (balance > 1000000):

            result = int(1000000//row['Low']) #number of stocks to buy
            transactions.append((row['Date'],'buy-low',row['Stock name'],str(result)))
            balance = balance - result*row['Low'] #buy low
            portfolio[row['Stock name']] += result #update the portfolio
            daily_balance.append(balance) #balance of the day
            daily_portfolio.append(row['Close']*portfolio[row['Stock name']]) #value of
                                unsold stocks
            day.append(row['Date']) #the day

        elif (flag == False) and (portfolio[row['Stock name']]>0) and (df.iloc[index+1]
                            ['High'] < df.iloc[index]['High']):
```

```
            result = portfolio[row['Stock name']] #number of stocks to buy
            transactions.append((row['Date'],'sell-high',row['Stock name'],str(result)))
            balance = balance + result*row['High'] #sell high
            portfolio[row['Stock name']] = 0 #set portfolio to 0 - sell all stocks
            flag = True
            daily_balance.append(balance) #balance of the day
            daily_portfolio.append(row['Close']*portfolio[row['Stock name']]) #0
            day.append(row['Date']) #the day

#sell the remaining stocks the last day at high price
last_profits = df.iloc[-1]['High'] * portfolio['AAPL'] #profits of the last day
transactions.append((row['Date'],'sell-high','AAPL',str(portfolio['AAPL'])))
balance += last_profits #update balance
portfolio['AAPL'] = 0 #set portfolio to 0 - all stocks sold
daily_balance.append(balance) #balance of the day
daily_portfolio.append(row['Close']*portfolio['AAPL']) #0
day.append(row['Date']) #last day

print(balance)
print(len(transactions))

save_transactions(transactions,'large.txt') #save the transactions
```

Python Code Snippet 7: Proposed Algorithm - Large sequence of 7681 transactions

```
#create a dataframe to plot the valuation diagram - large sequence
data = {'Date': day, 'Balance': daily_balance, 'Portfolio': daily_portfolio}
large_df = pd.DataFrame(data)
large_df['Total Value'] = large_df['Balance'] + large_df['Portfolio'] #Each day valuation
large_df['Date']= pd.to_datetime(large_df['Date'])
large_df.sort_values(['Date'], ascending = [True])

#Plot the valuation diagram for the large sequence
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (9,6)
plt.fill_between(large_df['Date'], large_df['Total Value'] ,label='Portfolio',
                 color='orange')
plt.fill_between(large_df['Date'], large_df['Balance'] ,label='Balance',color='blue')
plt.title('Valuation Diagram - Large sequence') #set title
plt.yscale('log') #change the y scale
plt.legend(loc='upper left') #add legend in the upper left corner

#A closer look in valuation diagram during the first 500 days
large_df = large_df[:500]
plt.rcParams["figure.figsize"] = (9,6)
plt.fill_between(large_df['Date'], large_df['Total Value'] ,label='Portfolio',
                  color='orange')
plt.fill_between(large_df['Date'], large_df['Balance'] ,label='Balance',color='blue')
plt.title('Valuation Diagram - Large sequence')
plt.yscale('log')
plt.legend(loc='upper left')
```

Python Code Snippet 8: Creation of the valuation diagrams for the large sequence

4.2 Small sequence of 997 transactions

The same algorithm can be used to generate a small sequence of transactions with a slight modification. The modification is that I stop performing transactions when the index reaches the number 1.430. This number was chosen with trial and error in order for the total number of transactions to be slightly less than 1.000. When the index is larger than 1430 and smaller than the length of the dataframe minus one (i.e. the last day of trading), I only update the lists daily_balance, daily_portfolio and day in order to use them for plotting the valuation diagram. During the last day of trading I choose to sell the remaining stocks (i.e. portfolio) at the high price, as I did before. The outcome of the algorithm in this case is profit of approximately 3.2 billion dollars with a total of 997 transactions. Again the generated profit is extremely high, despite the fact that the number of transactions is less than 1.000.

Furthermore, I present the valuation diagram of the small sequence. As I described previously, I take advantage of the fact that the price of the AAPL stock increases immensely over time. More specifically, the algorithm stops trading on 1990-04-24 and the remaining portfolio at that time (i.e. 18.186.104 stocks) is sold on 2017-11-10. Therefore, the stocks are sold in an extremely high price and the profits skyrocket. In this case the first 996 transactions are the same as the large sequence and thus the valuation diagram for the first 996 days is the same. Consequently, a closer look in the first 500 days will produce

the same valuation diagram as in the case of the large sequence. The validation of the solution and the valuation diagram for the small sequence of 997 moves are presented below.

Success:

After 997 transaction(s), your profit is 3189478919.69392 dollars.

Figure 9: Profit and number of transactions of the small sequence - Proposed Algorithm

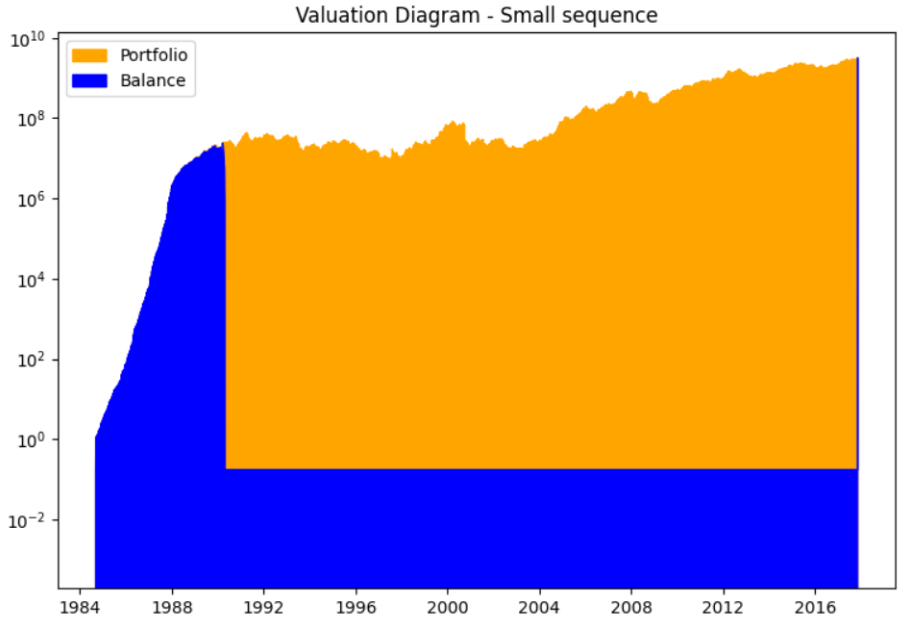

Figure 10: Valuation Diagram - Small Sequence of 997 transactions

The Python code for the implementation of the algorithm and the creation of the valuation diagram is presented below.

```
#small sequence <1.000
transactions = [] #list for transactions
portfolio = {'AAPL': 0} #dictionary to "save" the portfolio
balance = 1 #initial balance
flag = True
daily_balance = [] #keep the daily balance for plotting
daily_portfolio = [] #keep the daily portfolio for plotting
day = [] #keep the day for plotting

for index, row in df.iterrows(): #iterate through the rows

    if (index<1430): #number selected for the transactions to be <1.000

        if (df.iloc[index]['Low'] < df.iloc[index+1]['High']) and (balance > row['Low'])
                                and (flag == True) and (balance < 1000000):

            result = int(balance//row['Low']) #number of stocks to buy
            transactions.append((row['Date'],'buy-low', row['Stock name'],str(result)))
            balance = balance - result*row['Low'] #buy low
            portfolio[row['Stock name']] += result #update the portfolio
            flag = False
            daily_balance.append(balance) #balance of the day
            daily_portfolio.append(row['Close']*portfolio[row['Stock name']])
            day.append(row['Date']) #the day

        elif (df.iloc[index]['Low'] < df.iloc[index+1]['High']) and (balance >
                                row['Low']) and (flag == True) and (balance > 1000000):

            result = int(1000000//row['Low']) #number of stocks to buy
```

```
            transactions.append((row['Date'],'buy-low',row['Stock name'],str(result)))
            balance = balance - result*row['Low'] #buy low
            portfolio[row['Stock name']] += result #update the portfolio
            daily_balance.append(balance) #balance of the day
            daily_portfolio.append(row['Close']*portfolio[row['Stock name']])
            day.append(row['Date']) #the day

        elif (flag == False) and (portfolio[row['Stock name']]>0) and (df.iloc[index+1]
                                ['High'] < df.iloc[index]['High']):

            result = portfolio[row['Stock name']] #number of stocks to buy
            transactions.append((row['Date'],'sell-high',row['Stock name'],str(result)))
            balance = balance + result*row['High'] #sell high
            portfolio[row['Stock name']] = 0 #set portfolio to 0 - sell all stocks
            flag = True
            daily_balance.append(balance) #balance of the day
            daily_portfolio.append(row['Close']*portfolio[row['Stock name']]) #0
            day.append(row['Date']) #the day

    if (index>1430) and (index < len(df)-1): #added for plotting

        daily_balance.append(balance)
        daily_portfolio.append(row['Close']*portfolio[row['Stock name']])
        day.append(row['Date'])

#sell the remaining stocks the last day at high price
last_profits = df.iloc[-1]['High'] * portfolio['AAPL'] #profits of the last day
transactions.append((row['Date'],'sell-high','AAPL',str(portfolio['AAPL'])))
balance += last_profits #update balance
portfolio['AAPL'] = 0 #set portfolio to 0 - all stocks sold
daily_balance.append(balance) #balance of the day
daily_portfolio.append(row['Close']*portfolio['AAPL']) #0
day.append(row['Date']) #last day

print(balance)
print(len(transactions))

save_transactions(transactions,'small.txt')
```

Python Code Snippet 9: Proposed Algorithm - Small sequence of 997 transactions

```
#create a dataframe to plot the valuation diagram - small sequence
data = {'Date': day, 'Balance': daily_balance, 'Portfolio': daily_portfolio}
small_df = pd.DataFrame(data)
small_df['Total Value'] = small_df['Balance'] + small_df['Portfolio'] #valuation of each
    day
small_df['Date']= pd.to_datetime(small_df['Date'])
small_df.sort_values(['Date'], ascending = [True])

#For the valuation diagram we need to see how the portfolio moves during time
plt.rcParams["figure.figsize"] = (9,6)
plt.fill_between(small_df['Date'], small_df['Total Value'] ,label='Portfolio',color='
    orange')
plt.fill_between(small_df['Date'], small_df['Balance'] ,label='Balance',color='blue')
plt.title('Valuation Diagram - Small sequence')
plt.yscale('log')
plt.legend(loc='upper left')
```

Python Code Snippet 10: Creation of the valuation diagram for the small sequence

5 Conclusion and Future Work

The proposed algorithm performs well both in the small and the large sequence as the generated profit is billions of dollars in each case (approximately 3.2 billions and 11.2 billions respectively). Therefore, I managed to travel in the past and become rich with only taking advantage of a single stock. Imagine how much the generated profit can become if more stocks are taken into consideration, if the intra-day trading is used and if different strategies are tested. The strategies that can be performed for proposing an algorithm are numerous. Nevertheless, my proposed algorithm is simple and efficient while guarantees extremely high profits. If I had the chance to travel in the past and perform stock trading I would gladly share my profits with the instructors and the teaching staff.