PYTHON PROJECT - TIME TRAVEL

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

COMPUTER SCIENCE DEPARTMENT

Programming Tools & Technologies for Data Science

Author:
Panagiotis Lamprakis (PhD Candidate - 03003142)

Date: February 26, 2023

1 Introduction

Programming Tools & Technologies for Data Science course is offered by National Technical University of Athens in scope of the PhD program. The goal of the course is to learn the tools and the techniques to analyze data and speed up its processing.

The course is split in to parts. The first part is an introduction to R programming language and exploratory data analysis, while the second part is an introduction to Python and data processing libraries (e.g Pandas, Numpy, SciPy, etc.), and a final deliverable of an algorithmic problem solved with the aforementioned technologies.

The problem we were summoned to solve is called ”time-travel”. The data we will use come from the *Huge Stock Market Dataset* existing in Kaggle. The original dataset consists of different small .txt files, one per stock.

In section 2 we will describe the dataset and the pre-processing which has undergone, the proposed solution and we will drive the reader through so some implementation details. In addition, we will show the plots of the trades performed by our program with a low number of trades (10) and a high number of trades (5100). In section 3 we propose some modifications we could apply to make our implementation better.

The project code can be found in Github.

2 Solution

2.1 Dataset & Pre-processing

The original dataset consists of many .txt file, each one related to the stock history of one stock. Each file contains 7 columns: Date, Open, High, Low, Close, Volume, OpenInt. Open and Close columns hold the stock value of the date the stock exchange market opens and closes respectively. High and Low column hold the highest and lowest stock price for the given Date. OpenInt is always zero. Volume is the total number of the stocks traded in the given Date.

Among the files, there are some that are empty. Those files are discarded from the process. We read all the files and merge them into a singe pickle file, which is used as input oin our algorithm. The merge process takes place in the *time-travel-pre-processing* jupyter notebook, found in the deliverable of the assessment. The produced dataset contains 14,887,665 rows.

OpenInt column is always zero and it is meaningless for our problem, so we drop the column to have less data to process. Additionaly, there are some rows (24652 in particular) that have zero value in one of the following three columns: i. High, ii. Low, iii. Volume. Those rows are useless because we assume that we cannot

buy stocks for free. Also, if volume is zero then we cannot buy or sell our stocks, making that day useless to process. We drop those rows from the input dataset as well. Last but not least, we drop the default index of the Pandas DataFrame and we replace it with the dates, to achieve fast accessing of the data. After that, we sort the DataFrame in ascending order based on the index.

2.2 Description

Finding the optimal solution for the ”time-travel” problem is not an easy problem to solve, as stated in the assessment's description. Our solution focuses on trading in time intervals and buying the largest possible and allowed volume of the stock.

The number of trades is given as input to our implementation. We find the minimum (1962-01-02) and the maximum (2017-11-10) date of the dataset. Construct the intervals by dividing the difference of days between the maximum and minimum date by the number of trades and get the floor value, as described in equation 1.

$$\lfloor inteval \rfloor = \frac{(max\_date - current\_date).days}{number\_of\_intervals} \tag{1}$$

After we find the interval, we search for all stocks in that interval to find if we afford to buy some stock volume given our budget and the max trade value limitations. If we find stocks to buy in the interval then we search for the best trade that can give us the maximum profit, else we proceed to next interval. At this point, we need to mention that we have implemented an adjustment in the intervals, making them more elastic. In more detail, we find the date of the day that we need to sell our stocks and we re-calculate the next intervals starting from the next available day.

We perform the trades in pairs, meaning that in the same interval we don't buy or sell more than one stock. This is a limitation of our implementation that we will discuss later in *Limitations & Improvements* section. We have implemented two optimization strategies: i. buy-low – sell-high and ii. buy-open – sell-high. The primary optimization strategy is the *buy-low – sell-high*, because it is the one that can guarantee us the maximum profit. However, in order to accommodate the limitation of the intra-day trades, we have introduced the *buy-open – sell-high* strategy, which acts as a fallback in case the primary strategy fails.

In order to find the maximum profit, we create three more columns in the local datasets that we process. The local dataset consists of rows for a specific stock, with dates greater or equal to the minimum buy value date but within the time interval we search into. On this local dataset, we calculate the reversed cumulative high value and we produce a column named *cummax_rev*. We subtract the minimum price from this column and we create the *diff_rev* column. This column holds the maximum profit. Using the diff_rev column, we go backwards to identify the date that produces this profit.

It is profound that we have implemented a greedy algorithm. The rationale behind splitting the trades in intervals lies on the thought of breaking the greediness of the algorithm that could lead us in performing less trades than requested.

2.3 Implementation details

We applied Object-Oriented Programming techniques to structure our solution. A high level UML diagram can be seen in Figure 1. We have created, in addition, custom exceptions based on usecases. We validate that the volume we buy is not more than the permitted, see Figure 4. There is a separate class called *State* in order to keep the state of the buy and sell trades we perform. Last but not least, because of the nature of the algorithm we have implemented, we need the input number, signifying the number of trades to be performed, to be even. If the input is an odd number then we raise an error and we don't allow the program to execute (see Figure 6).

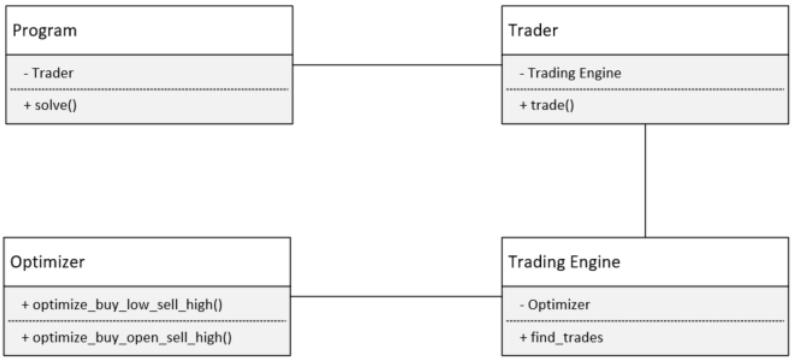

Figure 1: High level UML diagram

```
def __buy_stock__(self, best_trade: SuggestedTransaction, trade_volume: int, best_trade_df: pd.DataFrame):
    """
    Update Balance & Portfolio for all the dates after buying the stock.
    Subsequent dates will be corrected when we will sell the stock.

    Create the buy state.
    """
    self.budget = round_value(self.budget - best_trade.min_price * trade_volume)

    select_mask = best_trade_df.index >= best_trade.min_price_date
    best_trade_df.loc[select_mask, 'Balance'] = self.budget
    best_trade_df.loc[select_mask, 'Portfolio'] = best_trade_df.loc[select_mask]['High'] * trade_volume

    return State(best_trade.min_price_date,
                 best_trade.buy_transaction_code,
                 best_trade.stock_name,
                 trade_volume,
                 self.budget)
```

Figure 2: buy stock

```python
def __sell_stock__(self, best_trade: SuggestedTransaction, trade_volume: int, best_trade_df: pd.DataFrame):
    """
    The trade model is to buy as much stocks as you want and sell them at the next best price,
    since the moment of the buy. After selling the stock, the portfolio is empty, thus why we
    fill with zero value.
    """
    self.budget = round_value(self.budget + best_trade.max_price * trade_volume)

    select_mask = best_trade_df.index >= best_trade.max_price_date
    best_trade_df.loc[select_mask, 'Balance'] = self.budget
    best_trade_df.loc[select_mask, 'Portfolio'] = 0

    return State(best_trade.max_price_date,
                 best_trade.sell_transaction_code,
                 best_trade.stock_name,
                 trade_volume,
                 self.budget)
```

Figure 3: sell stock

```python
def __calculate_volume_afford_to_buy__(self, suggested_trade: SuggestedTransaction):
    """
    We can trade specific stock volumes:
    1. What we can buy based on our current budget
    2. What we can buy based on the available stock volume
    3. What we can buy in regards to the restriction of not buying stocks
       with value more than the maximum transaction threshold

    From those three categories, we select the minimum volume, because
    either our budget is not enough to buy more, or there is no more
    stocks to trade that day, or we are capped by the transaction threshold.
    """

    volume = min(suggested_trade.transaction_volume,
                 int(self.budget / suggested_trade.min_price),
                 int(self.max_transaction_value / suggested_trade.min_price))

    if volume * suggested_trade.min_price > self.max_transaction_value:
        raise ValidationError(f'Max allowed daily limit is: {self.max_transaction_value}, ' +
                              f'you bought {volume * suggested_trade.min_price} ')

    return volume
```

Figure 4: Calculate volume to buy

```python
def __decide_best_trade__(self, suggested_trade_list: list[SuggestedTransaction]) \
        -> tuple[SuggestedTransaction, int, int]:
    """
    Best trade is decided by maximizing: unit_profit * volume
    """
    best_trade = None
    max_profit = 0
    trade_volume = 0
    for suggested_trade in suggested_trade_list:
        if not self.__can_buy_unit__(suggested_trade.min_price):
            continue  # check next stock if we cannot buy not even a single unit from the current one
        stock_profit, volume = self.__calculate_profit_for_trade__(suggested_trade)
        if stock_profit > max_profit:
            max_profit = stock_profit
            best_trade = suggested_trade
            trade_volume = volume

    return best_trade, max_profit, trade_volume

def __can_buy_unit__(self, stock_price):
    return round_value(self.budget) >= round_value(stock_price)

def __calculate_profit_for_trade__(self, suggested_trade: SuggestedTransaction):
    volume_to_buy = self.__calculate_volume_afford_to_buy__(suggested_trade)
    return round_value((suggested_trade.max_price - suggested_trade.min_price) * volume_to_buy), volume_to_buy
```

Figure 5: Find best trade among others

```python
class Trader:

    def __init__(self, stock_history, initial_budget, max_transaction_value, number_of_transactions):
        self.stock_history = stock_history
        self.budget = initial_budget
        self.max_transaction_value = max_transaction_value
        if number_of_transactions < 1:
            raise ValueError('Number of transactions cannot be zero or negative.')
        elif number_of_transactions % 2 != 0:
            raise ValueError('Number of transactions must be even number.')
        self.number_of_transactions = number_of_transactions // 2
        self.engine = TradingEngine()
```

Figure 6: Input validations

2.4 Small Sequence

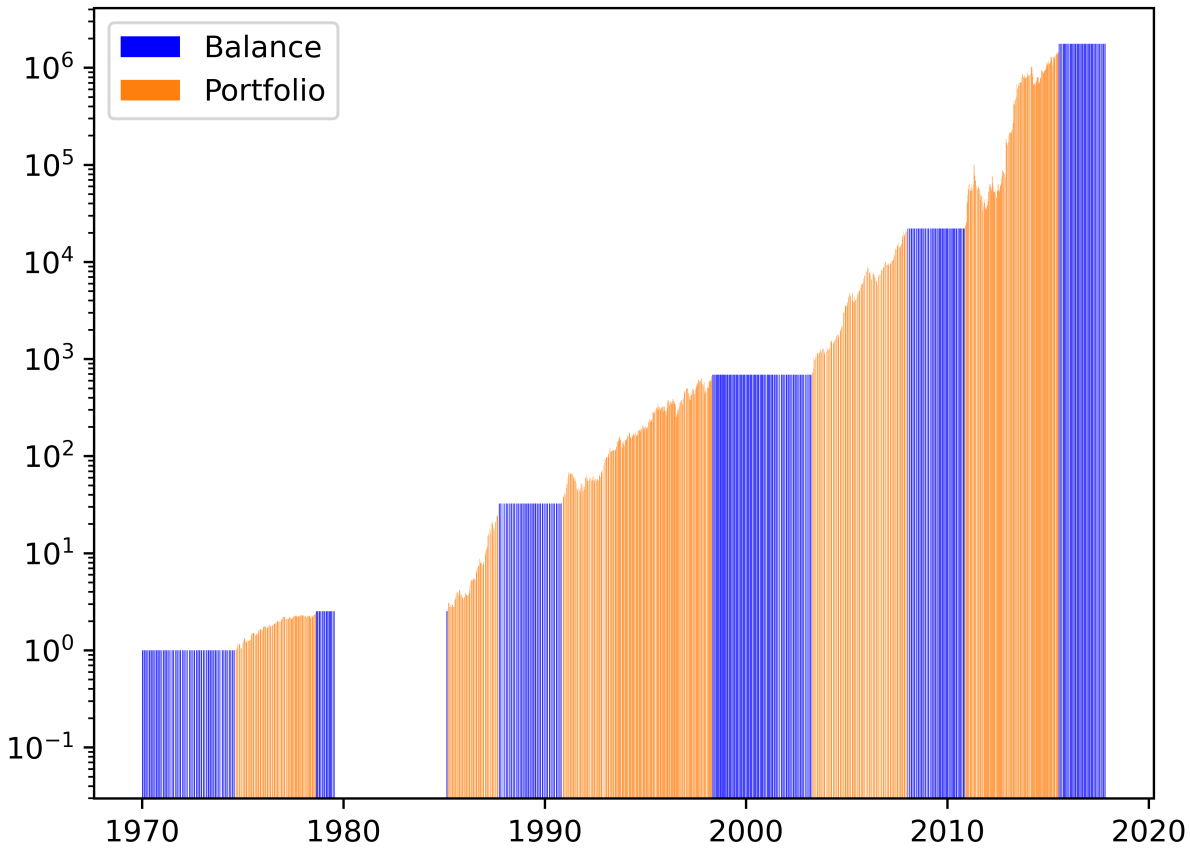

Figure 7: 10-trade graph

We believe that the large empty region appearing in Figure 8 is because of the data cleaning we have performed. Those dates miss from the date in the plot DataFrame. This is something that we acknowledge and we need to fix. Regarding the small empty regions, this is because those dates are weekend and they miss from the original dataset. Since the stock price is undefined during weekends, we shall not plot those days, thus why we have left them empty.

2.5 Large Sequence

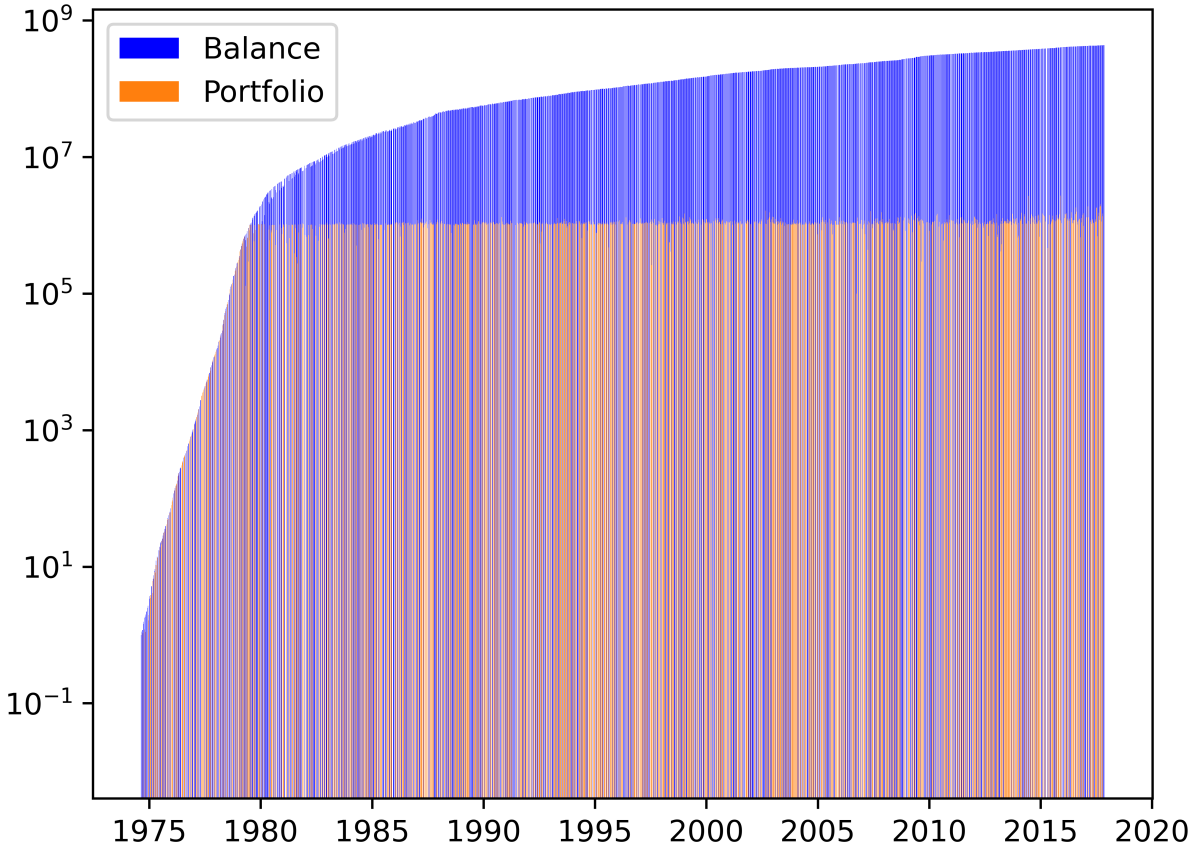

Figure 8: 5100-trade graph

In Figure 8 it seems that there are empty regions. This is a problem of scaling in TeX. The original images are attached in the project deliverable.

3 Limitations & Improvements

The solution described in the document was designed and implemented in three days, fact that didn't leave us much room for improvement of the original idea. Below, we mention some ideas that would make the solution better and more robust and that we would like to implment in the future if we had more available time.

As already mentioned, we have restricted the input to be an even number. With the current implementation, this is something that we cannot address because it is an intrinsic problem of the design.

Moreover, the algorithm does not perform well for large input (N), if:

$$N > number\_of\_days\_in\_the\_dataset \tag{2}$$

so we need to adjust the implementation accordingly. One thought would be to take the *top(K)* best trades within a trade window, where *K* is defined in eq. 3, so that in every trading window we perform more than one trade if allowed. The *K* would be elastic and it would be updated after each trading epoch ends. However, this improvement will not lift the limitation of the input to be an even number. A mitigation to this problem would be to split one random trade into two sell steps, so that we buy X volume of a stock and we sell Y and Z volume in two steps, where X = Y + Z.

$$\lceil k \rceil = \frac{N}{number\_of\_days\_in\_the\_dataset} \tag{3}$$