2IMD15 Data Engineering
Milestone 1

Group 1: Daniel Teixeira Militao (1486314), Manon Wientjes (1398903), Pawel Budzynski (1511734), Tong Zhao (1416790), Ugne Laima Ciziute (1495186)

I. Datasets

We chose the 2020 stock dataset[1] provided, which contains data from different stocks from January until mid April. Each file of the dataset relates to a particular stock and month as indicated by its name. Its columns are the date, time (CET), which corresponds to the frequency of updates, opening, closing, lowest, and highest price, at this timestamp, and sum of volume of all transactions, also within this update. We use the opening, highest, and lowest prices to do correlations on.

In order to do correlations on the data we needed to pre-process it. We did so with the help of Spark-SQL. We filtered out stocks that did not have entries in every month and because we wanted to have data from markets that were active concurrently we filtered out stock that were not updated between 9h00 and 17h00 (CET) on working days.

The frequency of updates for some stocks was too low so we filtered out stocks that did not have at least 12,000 entries. This is approximately the number of entries a stock should have if it has a new entry every 5 minutes for 8 hours per day for 76 working days (around the number of working days we have from January to April). However, we still had missing values in our data so we estimated them through interpolation at a frequency of 1 minute. At this point we were left with data from 666 different stock. Finally, to handle final missing values at the start and end of certain days, we extrapolated stock data by picking the nearest value.

At this stage we had opening, highest and lowest prices from January through mid April of 666 different stocks with an update frequency of 1 minute. This allows us to easily select any update frequency we want for subsequent analysis and for this milestone we settled on 1 hour update frequency. As a result, we ended up having 1998 vectors with 639 dimensions each. Since we want to compute correlations between all three variables over the whole period. We also pre-calculate values needed to compute the correlations, as described in Section IV.

[1]https://canvas.tue.nl/courses/10287/files/2383551/download?download_frd=1

II. Correlation measures

A. Pearson correlation

Given paired data $\{(x_1, y_1), ..., (x_n, y_n)\}$ consisting of n pairs, the Pearson correlation is defined as:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}},$$

where x_i and y_i are individual sample points with index i and $\bar{x}$ and $\bar{y}$ are the sample means.

B. Mutual Information

Let (X, Y) be a pair of discrete random variables with values over the space $\mathcal{X} \times \mathcal{Y}$. Then, their Mutual Information (MI) is defined by:

$$I(X;Y) = \sum_{y \in \mathcal{Y}} \sum_{x \in \mathcal{X}} p_{(X,Y)}(x, y) \log \frac{p_{(X,Y)}(x, y)}{p_X(x) p_Y(y)},$$

where $p_{(X,Y)}$ is the joint distribution and p_X and p_Y are the marginal distributions of X and Y respectively.

III. System architecture

We read the pre-processed data into an RDD so that it has the following format:

$[(\textit{stock\_name–price\_type} : String,\ \textit{data\_vector} : Array)]$.

Then we pass the RDD to Algorithm 1 together with the correlation function that we want to apply and a number of groups we want to split the data into, in order to be able to parallelize the workflow.

Algorithm 1 creates combinations of groups of data and calls function $ComputeCorrelationsForGroups$ on each pairing to compute correlations, making sure not to compare the same stock.

IV. Distribution of computation

The $groupBy$ on Line 2 of Algorithm 1 is used to divide the data into equally sized groups. This allows us to distribute the workload between multiple workers when we apply the $flatMap$ transformation in Algorithm 1 on Line 5. After all, each worker will get an equal amount of lines and hence an equal workload.

We pass the $noGroups$ to Algorithm 1, which is equal to the number of threads we initiated the SparkContext

Algorithm 1: ComputeCorrelations($rdd, noGroups, correlationFunc$)

1: $rdd \leftarrow rdd.zipWithIndex()$
{adds unique indices to each stock value type}
2: $rdd \leftarrow rdd.groupBy(...).mapValues(tuple)$
{GroupID = zipIndex%noGroups}
3: $rdd \leftarrow rdd.cartesian(rdd)$
{creates combinations of all possible groups}
4: $rdd \leftarrow rdd.filter(...)$
{removes all redundant group combinations}
5: $correlations \leftarrow$
$rdd.flatMap(ComputeCorrelationsForGroups)$

with, but not more than the hardware allows, times two. Setting too many groups will create too much overhead, since more workers are needed to perform $flatMap$. However, having too few groups will mean that the workload per worker will become too high, not utilizing Spark to its full potential as we would not be properly parallelizing tasks and could lead to memory issues. Through experimentation we concluded that the best group size for performance was related to the amount of threads we have.

Some of the workers will have less workload, specifically the ones that will be calculating correlations between the same group. This is because we make sure that we do not calculate correlations between same pairs of stocks, the specific worker that got two of the same group will have a smaller workload. This is also a bottleneck in our code, since we do not ensure that every worker gets a similar number of same group combinations, meaning that the workload is not 100% evenly distributed. Another inevitable bottleneck is that the last worker might get less workload. In order to make sure that no worker is idle for too long, we create twice as many jobs than we have threads.

To avoid redundant computations, on Line 4 of Algorithm 1 we filter out all combinations of groups that are below the leading diagonal of the Cartesian product. This allows us to get rid of most redundant computations. Furthermore, we made sure to not do redundant computations between stock value types, between pairs of stock value types within the same group or between different groups, by using a nested for loop and testing some logical predicates.

To avoid unnecessary repeated calculations during correlations we also pre-processed the data specifically for each correlation type. For Pearson correlation we pre-calculated the values of $x_i - \bar{x}$ for each x_i of vector x. For MI we converted each vector to account for the increase or decrease of the prices compared to the previous update, making the values of your time series to -1/0/+1.

V. Theoretical complexity and experimental performance

We derived the algorithmic complexity for calculating the correlation measures. We do this separately for Pearson correlation and MI.

For computing the correlations, $flatMap$'s complexity is linear in the number of lines, where each line is a combination of groups. The comparison of two groups take $O(size^2 \cdot correlation\_func\_complexity)$, where size is the size of each group. The complexity of Pearson correlation is $O(dimension)$, where the dimension is the length of each vector and the constant is 6. The complexity of MI is $O(dimension)$, since the values are mapped to -1/0/1. Here the constant is 4.

Hence, the complexity for calculating correlations is $O(\#combinations \cdot size^2 \cdot dimension)$.

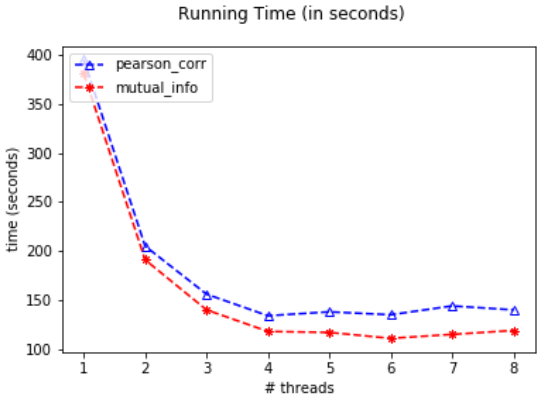

Fig. 1: Running time per number of threads

We ran experiments on the number of threads used to see how the running time would behave, the results of which can be seen in Figure 1. The experiments were run on a MacBook Pro (Retina, 15-inch, Mid 2015) with 4 cores and 16GB RAM.

After running experiments, it could be verified that the running time for MI and Pearson correlation is similar, however, Pearson correlation takes slightly longer. This is in line with our expectation according to the derived complexity. We also concluded that increasing the number of threads reduces running time but, it plateaus at around $(\#threads/2)$. That is probably also why it is recommended in Spark documentation to set the number of threads to the number of cores[2]. Of note that if we set that number to something above the actual number of threads we get the same performance as using the maximum number of threads.

[2]https://spark.apache.org/docs/latest/submitting-applications.html#master-urls

Increasing the number of threads increases the amount of RAM needed. This might cause data to overflow to disk so we will spend more time doing reading and writing operations. Therefore, it might be the case that the processing time improvement from using more threads is mitigated by the extra time spent doing I/O operations, meaning the overall running time stays more or less the same.

VI. Insights

As we had anticipated, regardless of the correlation function, the top correlations were, for the most part, between the same value type, e.g. opening with opening.

Nine of the top ten Pearson correlations are positive and between the same three stocks and value types, as displayed in Table I. The tenth correlation is an oddity however, as it is a negative correlation between the lowest price of *Forex_CHFEUR*, the exchange rate between the Swiss Franc and Euro, and the highest price of *Forex_EURCHF*, the exchange rate between the Euro and the Swiss Franc. This relation makes perfect sense, if the exchange rate from one currency to another increases, obviously the opposite rate decreases. If we look at the fifty highest Pearson correlations we verify that they are all above 0.999 and for the most part between the same stocks and value types. In the future, we could perhaps filter correlations pairs with values above 0.95 in order to derive more interesting insights.

MI provides very different results and the ten highest are shown in Table II. The scores for MI stay relatively low, raising the question whether the applied mapping was the best choice or if the MI score is meant to be low for all stock pairings. Based on its definition, the MI is the reduction in uncertainty in bits about some random variable Y after another random variable Y is observed. Each vector has a dimension of 639. Hence, we can have 3^{639} different possibilities for Y (for each position we can have -1, 0, 1). The number of bits needed to encode this is $\log_2(3^{639})$. Looking at our results, the highest values are around 0.45, meaning observing one specific vector will reduce this entropy by 0.45 bits. This is a proof that these pairs of stocks in our results do share some information. It might be a good choice to further investigate the topic and experiment with different ways of discretization of stock prices.

TABLE I: Ten highest Pearson correlations

Pair		Value
Amsterdam_MT-lowest	Madrid_MTS-lowest	0.9999
Amsterdam_MT-highest	Madrid_MTS-highest	0.9999
Amsterdam_MT-opening	Madrid_MTS-opening	0.9999
CME_6A-lowest	Forex_AUD-lowest	0.9999
CME_6A-opening	Forex_AUD-opening	0.9999
CME_6A-highest	Forex_AUD-highest	0.9999
Forex_GBP-lowest	CME_6B-lowest	0.9998
CME_6B-opening	Forex_GBP-opening	0.9998
Forex_GBP-highest	CME_6B-highest	0.9997
Forex_CHFEUR-lowest	Forex_EURCHF-highest	-0.9997

TABLE II: Ten highest Mutual Information correlations

Pair		Value
CME-eMini_NQ-opening	CBOT-mini_YM-opening	0.4579
London_TUI-highest	Xetra_TUAG00-highest	0.4488
London_TUI-opening	Xetra_TUAG00-opening	0.4349
London_TUI-lowest	Xetra_TUAG00-lowest	0.4279
CME-eMini_NQ-lowest	CBOT-mini_YM-lowest	0.4208
CME-eMini_NQ-highest	CBOT-mini_YM-highest	0.3845
vwd-Indications_ XPDUSD-lowest	NYMEX_PA-lowest	0.3827
vwd-Indications_ XPDUSD-highest	NYMEX_PA-highest	0.3489
US-Indices_DJGT-highest	US-Indices_ W1DOW-highest	0.3199
vwd-Indications_ XPDUSD-opening	NYMEX_PA-opening	0.3198