2IMD15 Data Engineering
Milestone 2

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 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 and time interval we want for subsequent analysis and for this milestone we settled on 1 hour update frequency and used data from February and March. We also decided to only pick the opening price of the stocks. As a result, we ended up having 666 vectors with 369 dimensions each. Since we want to compute correlations between any p vectors 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 and aggregation 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. In order to use this measure for more than two vectors it is necessary to introduce linear transformation that map multiple vectors into one. The one applied is averaging function defined as:

$$avg(x_1, x_2, \ldots, x_n) = \frac{x_1 + x_2 + \ldots + x_n}{n},$$

where $x_1, x_2, \ldots, x_n$ is a set of vectors.

B. Total correlation

For a given set of n discrete random variables $\{X_1, X_2, \ldots, X_n\}$, the total correlation is defined by:

$$C(X_1, \ldots, X_n) =$$
$$= \sum_{x_1 \in X_1} \cdots \sum_{x_n \in X_n} p(x_1, \ldots, x_n) \log \frac{p(x_1, \cdots, x_n)}{p(x_1) \cdots p(x_n)},$$

where $p(X_1, \ldots, X_n)$ is joint distribution and $p(X_1)p(X_2) \cdots p(X_n)$ is the independent distribution. Because this measure is defined for multiple vectors there is no need to transform them. Thus, identity function was applied as aggregation.

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 and aggregation functions that we want to apply and a p value, to know the number of stocks we want to correlate together. We also pass 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 size p of groups of data and calls function $ComputeCorrForGroups$ on each group to compute correlations, making sure not to compare the same stock.

Algorithm 1: ComputeCorrelations($originalRdd$, $noGroups$, $correlationFunc$, $aggregationFunc$, p)

1: $rdd \leftarrow rdd.zipWithIndex()$
{adds unique indices to each stock value type}
2: $rdd \leftarrow rdd.groupBy(...).mapValues(tuple)$
{GroupID = zipIndex%noGroups}
3: $rddCart \leftarrow rdd.cartesian(rdd)$
{creates combinations of all possible groups}
4: $rddCart \leftarrow rddCart.filter(...)$
{removes all redundant group combinations}
5: **if** p==3 **then**
6: $rddCart = rddCart.cartesian(rdd)$
{creates combinations of three of all possible groups}
7: **else if** p==4 **then**
8: $rddCart = rddCart.cartesian(rddCart)$
{creates combinations of four of all possible groups}
9: $rddCart = rddCart.filter(...)$
{removes all redundant group combinations}
10: $correlations \leftarrow rddCart.flatMap(ComputeCorrForGroups)$

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 10. 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 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 or more 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 Lines 4 and 9 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 different value types of the same stock.

We also save on computations, when calculate the cartesian product for $p = 4$ on Line 8 of Algorithm 1. Since by the time we need to compute the cartesian product for $p = 4$ we already have a cartesian product for $p = 2$, we compute a cartesian product between a cartesian product for $p = 2$ and itself. This requires much less computations as it has to do the cartesian product only twice, when compared to making a cartesian product 3 times between itself and the original list of stocks.

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 total correlation (TC) we binned each vector in 100 bins. For 100 bins the trade-off between the top 10 correlations found and time was the optimal.

V. Theoretical complexity and experimental performance

We derived the algorithmic complexity for calculating the correlation measures. We do this separately for Pearson correlation and Total Correlation. The time complexity for generating cartesian products of p stock vectors is $O(size^p)$, where size is the number of stock vectors in our experiments. For Pearson correlation, we need to calculate all the combinations for those p vectors, which takes $O(2^p)$ time. The complexity of Pearson correlation is $O(dimension)$ where dimension is the length of each stock vector, plus the time for aggregation, it takes $O(dimension * p)$ in total for Pearson correlation. The complexity of Total Correlation is $O(dimension \cdot p)$. Hence, the complexity for calculating correlations for Pearson is $O(size^p \cdot 2^p \cdot dimension \cdot p)$ and $O(size^p \cdot dimension \cdot p)$ for Total Correlation.

The experiments were run on a Microsoft Azure server with 6 executors with a total of 15 cores and 32GB RAM.

We first determined the max complexity of the problem such that Pearson correlation and Total correlation run in 30 minutes. Pearson correlation runs in about 30 minutes for 666 vectors each of dimension 369 (February and

March) for $p = 3$. Total correlation runs in about 30 minutes for 666 vectors each of dimension 198 (only March) for $p = 3$.

TABLE I: Running time (sec) for increasing no. of vectors for $p = 3$.

Number of vectors	222	444	666
Pearson Correlation (369 dim.)	95	590	1,928
Pearson Correlation (198 dim.)	79	371	1,230
Total Correlation (198 dim.)	69	547	1,844

We ran experiments for $p = 3$ on the number of vectors used to see how the running time would behave, the results of which can be seen in Table I. Furthermore, for $p = 4$ using 111 vectors with 198 dimensions Pearson took 1,653 seconds and using 185 vectors with 198 dimensions TC took 1,778 seconds.

After running experiments, it could be concluded that for the same data complexity Pearson is faster than TC, which is inline with our expectation. Although TC uses the identity function and has therefore three times less combinations than Pearson, the implementation for Pearson uses Numpy and TC does not.

We conducted further experiments on a more powerful cluster on a Microsoft Azure server with 11 executors with a total of 40 cores and 28GB RAM. The reason for these extra experiments was to see how our application would scale and if it still efficiently used all cores and we are happy to report that it does. Tables II and III show the results of these experiments.

TABLE II: Running time (sec) for increasing no. of vectors on the large cluster for $p = 3$.

Number of vectors	222	444	666
Pearson Correlation (369 dim.)	81	375	1,163
Total Correlation (369 dim.)	57	421	1,481

TABLE III: Running time (sec) for increasing no. of vectors on the large cluster for $p = 4$.

Number of vectors	37	74	111
Pearson Correlation (369 dim.)	360	850	1,440
Total Correlation (369 dim.)	139	255	457

VI. Insights

All the results of the top ten Pearson correlations are above 0.999. The top one refers to currency exchange markets. Next 3 positions seem to look quite interesting as they show strong correlation between value of a company located in Geneva and different exchange rates of Swiss franc. Also nine of then top values include STM, either Paris or Mailand, and we suspect that it is a one and the same company present on two stock exchange markets.

Total correlation provides very different and way more interesting results shown in Table V. Even though Amsterdam MT and Madrid MTS seem to base on the same company its high correlation with Xetra 750000 is an interesting finding. The third and fourth rows show stocks that behaved very similar during this period.

Based on its definition, TC is the difference between sum of information entropy of all the stocks and the joint entropy of these stocks. Based on our results, we can use 4 bits to encode this difference. This is a proof that these pairs of stocks in our results do share some information, but still very little. Since we have 100 bins for all the stock values, we need $log(100^{666})$ bits to encode all the original stock values. Also, we see the highest values always appear among stocks with city prefix like "Amsterdam", "London", "Paris" and "Madrid", which may imply some historical events happened and influenced the whole European stock market during that time.

TABLE IV: Ten highest Pearson correlations for March+February opening price

Pair		Value
Forex_AUD	(CME_6J, CME_6A)	0.99997
Mailand_STM	(Paris_STM, Forex_ZARCHF)	0.99997
Mailand_STM	(Paris_STM, Forex_TRYCHF)	0.99997
Mailand_STM	(Paris_STM, Forex_MXNCHF)	0.99997
Paris_STM	(CME_6J, Mailand_STM)	0.99997
Mailand_STM	(Paris_STM, CME_6M)	0.99997
Mailand_STM	(Paris_STM, CME_6J)	0.99997
Paris_STM	(CME_6M, Mailand_STM)	0.99997
Paris_STM	(Forex_MXNCHF, Mailand_STM)	0.99997
Mailand_STM	(Paris_STM, Forex_HUFCHF)	0.99997

TABLE V: Ten highest Total Correlation correlations for March opening price

Pair			Value
Amsterdam_MT	Xetra_750000	Madrid_MTS	4.0257
Paris_CNP	Madrid_ACS	London_CCL	4.0158
London_ICP	Madrid_ACS	London_CCL	4.0066
Amsterdam_MT	Paris_CNP	Madrid_MTS	4.0039
Paris_CNP	London_ICP	London_CCL	4.0023
Amsterdam_MT	Madrid_ACS	Madrid_MTS	3.9748
Paris_CNP	London_ICP	Madrid_ACS	3.9733
Amsterdam_MT	Viena-Exchange_ AT0000743059	Madrid_MTS	3.9523
Paris_STM	London_SMIN	Mailand_STM	3.9442
Amsterdam_MT	Paris_CS	Madrid_MTS	3.9441