Data Engineering

Aniket Ninawe, 1454072, Data Science in Engineering, TU Eindhoven

1 Introduction

In this project, I worked with Spark platform for processing the big data and calculating the different aspects of the correlation measures. The data pre-processing is explained, followed by the description of system architecture and the parallel processing. Theoretical complexity and experimental performance have been analysed followed by the meaningful insights.

2 Dataset

2.1 Description

We have selected the 2018 Stock dataset provided by the professor[1] to perform our analysis on. The dataset contains trading data for 2182 unique stocks, on 40 unique stock exchanges. The monthly data is provided by stocks with each stock being associated with a specific stock exchange and is initially stored in the .txt format. Each file contains a trading history of a stock in a particular month and has the following schema.

- Date (Calendar Year 2018)
- Time (in CET timezone)
- Opening price (The first price within this minute)
- Highest price (The highest price within this minute)
- Lowest price (The lowest price within this minute)
- Closing price (the last price within this minute)
- Volume (Sum of the volume of all transactions within this minute)

The dataset can be downloaded from the website: 2018 Stock.

2.2 Data Prepossessing

The data originating from different stock exchanges or different stocks within one stock exchange may differ in the frequency of updates, the number of missing values over time and is tied to the working hours of the corresponding stock exchange. To align the data for further analysis, the following steps were taken.

1. Yearly data.

   Monthly text files by stocks were combined into yearly files. Few stock exchanges, as well as several individual stocks, for which the trading data was not available for all of the 12 months, were excluded.

2. Frequency of updates.

   Studying the frequency of updates revealed that the vast majority of the stocks have a 1-minute frequency of updates, defined as the most frequent time interval between two subsequent entries. Therefore all the less frequently updated stocks were excluded as well.

3. Clustering.

   The stock exchanges were split into 4 clusters based on their working hours in CET: Asian, European, American and Common cluster with the latter cluster including stock exchanges that cannot be attributed to a specific timezone, such as forex, or can be attributed to more than one. European cluster (14 stock exchanges) was chosen to be the focus of the analysis.

4. Data incompleteness.

   The analysis showed that although having the needed 1-minute frequency of updates, many stocks have a substantial number of missing values during the working hours, therefore such stocks, specifically those having less than 1000 entries per month (~ 50 one-minute entries per working day), were also excluded, thereby leaving 540 stocks traded on 14 European stock exchanges to proceed with.

5. Aggregating.

   The data was aggregated by resampling to 1-hour frequency, using mean as the aggregation function. Timestamp label of a specific hour was tied to the right end of the 1-hour averaging interval.

6. Finalizing the dataset.

   To produce the needed number of meaningful vectors, 3 columns (Open price, Close price, Volume) were taken for each stock from the European cluster[2]. Subsequently, the quarterly subsets were aligned to the first quarter: 3 months were subtracted from the timestamps of the Q2 subset, 6 months - from the timestamps of the Q3 subset, and 9 for Q4, correspondingly. The resulting Q1-aligned subsets were stacked by timestamps (rows).

   Finally, the dataset was rounded to 4 decimals to reduce the resulting file size. Then, it was transposed before saving for the convenience of processing in Spark. The rounding precision parameter was chosen such that the rounding effect on the output of correlations calculation is negligible.

Having undergone the above mentioned preprocessing steps, we created a dataset of 6480 vectors stored as comma-separated lines in a text file, each with 999 dimensions. The non-common missing values between vectors in the resulting dataset are dealt with during further pairwise calculation, as dropping observations if at least one of the 6480 vectors has a missing value at a certain position leads to substantial loss of information. The format of the resulting file is as follows:

```
Amsterdam_AALB_Open_Q1,42.0468,42.044,42.0652, ...
Amsterdam_AALB_Close_Q1,42.0405,42.041,42.0697, ...
```

3 Correlation Measures

Pearson's correlation and Mutual Information measures were selected for the present study.

Pearson's correlation was chosen as the most commonly employed measure of the correlation between two numerical variables. Specifically, it is a measure of linear dependence, assigning a value between -1 and 1, where 0 indicates no correlation, 1 is a perfect positive correlation and -1 is a perfect negative correlation. The formula is as follows.

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

$\bar{x}$: Average value of x.
x_i : value of x at time i.
n : The size of the sample.

[1] https://canvas.tue.nl/courses/10287/discussion_topics/81710

[2] It was taken into consideration that the correlation between opening and closing prices is trivial. Still, the number of vectors in one dataset is of major importance in itself for this particular assignment.

Mutual Information measures mutual dependencies that occur between two random variables, sampled simultaneously, on average. Mostly, it is used as a generalized correlation measure. The range of the mutual information is $[0,+\infty]$. The 0 value shows no mutual information which means the two variables are independent. The higher the MI value, the larger is the reduction of uncertainty between two variables. In the formula below, frequencies of values of a variable were taken to calculate empirical probabilities.

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p_{(X,Y)}(x,y) \log \left(\frac{p_{(X,Y)}(x,y)}{P_X(x) P_Y(y)} \right) \quad (2)$$

$p_{(X,Y)}$: Joint distribution of X, Y.

Several approaches were taken in implementation of the correlation functions in terms of the type of input they get, including Python lists, RDD, Spark dataframe, NumPy array. Among all the approaches, NumPy arrays-based implementation proved to be the fastest. Hence both correlation functions take 2 equally sized NumPy arrays as input.

4 System Architecture

Two competing architectures for calculation of pairwise correlations were developed. The first approach is based on constructing a Cartesian product of the set of all vectors with itself to get all pairs of vectors, accounting for the symmetry of the correlation measures under consideration to eliminate double computation. The second approach is inspired by the idea provided by the professor [3] and relies on an auxiliary abstract (non-materialized) table that indexes all the combinations. Both approaches are described with more details and Pyspark pseudocode below.

4.1 Approach 1

4.1.1 Parsing vectors

The data from a text file, formatted as shown above, is parsed into an RDD with elements being tuples, containing a stock name and the corresponding vector of floating-point values, stored as NumPy array structure for Python:

```
("Amsterdam_AALB_Open_Q1", <42.0468,42.044, ...>)
```

4.1.2 Cartesian product

```
combinations =
    vectors.cartesian(vectors)
           .filter(lambda combination:
                   vectorname1 < vectorname2)
```

The inequality sign above denotes alphabetical order ("A" < "B"). This way, only unique combinations of distinct vectors are left: ("A", "B"), but not ("B", "A") or ("A", "A").

It is worth noting that "vectors" object is cached before calculation of the Cartesian product (takes only ∼21 MB in memory) which helps significantly improve the overall performance.

4.1.3 Correlations calculation

```
correlations =
    combinations.map(lambda combination:
                     (vectname1, vectname2,
                      CORR_FUNC(vect1, vect2)))
```

Correlation function, given as an argument to the architecture is called inside the map transformation applied to the set of combinations. As a result, triples (vectname1, vectname2, corrValue) are produced. Finally, triples are filtered based on the threshold τ that is given as input as well.

[3] https://canvas.tue.nl/courses/10287/discussion_topics/83659

4.2 Approach 2

4.2.1 Parsing vectors into table

Initially, the text file is parsed and transformed into a Spark DataFrame, having the following format:

```
+--------------------+--------------------+---+
|                name|              values|idx|
+--------------------+--------------------+---+
|Amsterdam_AALB_Op...|42.0468,42.044..|  1|
|Amsterdam_AALB_Cl...|42.0405,42.041..|  2|
+--------------------+--------------------+---+
```

All the vectors are enumerated with a monotonically increasing index taking values from 1 to the number of vectors, 6480 in the reference dataset.

4.2.2 Auxiliary index table

The idea is to generate key-value pairs, two combinations with the same key for each unique non-trivial combination of vectors such that these pairs are subsequently passed to reducers, which essentially calculate correlation functions and are meant for exactly two objects to reduce. The keys used to generate key-value pairs can be organized in an abstract table as shown below for a small example. The table is not materialized, though its configuration is used to calculate the needed keys for each vector.

In this example, the total number of vectors is 5, and the index of vector B is 2. For instance, the correlation between vectors (D, B) is guaranteed to be calculated on the following reduceByKey step by generating exactly two key-value pairs with the same unique key: (17, B), (17, D). No other pair with key 17 will be produced.

4.2.3 Generating key-value pairs

The idea with producing key-value pairs is implemented as a generator function in Python, taking 3 arguments: *N*, number of vectors; *idx*, the index of a particular vector; *vector*, the vector itself in this case. This generator is then passed to the flatMap transformation applied to the RDD associated with the DataFrame mentioned above.

```
keyValPairs =
    vectors_df.rdd.flatMap(lambda tup:
                           pairs_generator(N, idx, vector))
```

Further, the RDD containing all generated key-value pairs is repartitioned based on the key value to ensure parallelism and then cached. It is worth noting that as *value* is the vector itself, not its name, it induces a certain memory overhead while caching (now it takes ∼616MB in memory), but this is easily managed even on a single machine, in time almost negligible as compared to the total computation time.

4.2.4 Reduce by key

Finally, reduceByKey transformation is invoked. As each key is associated with exactly 2 generated key-value pairs by design, the code below results in triples (vectname1, vectname2, corrValue), thereby being equivalent to the first approach. Similarly, at the end of the process, triples are filtered based on the threshold τ.

```
correlations =
  keyValPairs.reduceByKey(lambda a, b:
  (a.name, b.name, CORR_FUNC(a.vect, b.vect)))
```

5 Distribution of Computation

We practised in setting up a cluster on a single node with either Spark standalone deploy mode [2] or with YARN [1] and considered using several of our computers to run a distributed cluster, but for the purpose of accomplishing the Milestone 1, a decent performance was achieved running Spark locally on a single machine using logical cores as workers (16 cores in a reference machine described in the section 6). In both approaches described in the previous section, we made use of parallelism in computation as explained below.

5.1 Approach 1

The RDD associated with the initial text file is partitioned at initialization using Spark default partitioning policy for external datasets. For the reference dataset of size 6480x999, described in section 2, the number of partitions is 8. Applying Cartesian product transformation automatically increases the number of partitions to the square of the initial value: 64 in our case.

The partitions are efficiently processed by the workers in parallel. Although the distribution of the workload can be unbalanced over partitions to some extent due to filtering of the set of combinations aimed at avoiding double computation, since the workers can very efficiently switch between partitions (vectors set persists in memory) this does not lead to bottlenecks. Such organization, on the other hand, can have limitations when the code is run on a much bigger dataset stored on a distributed cluster when resources are tied to specific nodes, as opposed to cores of one machine, and the vectors set cannot fit in the memory of one node, thus the network overhead could be noticeable.

5.2 Approach 2

Approach 2 differs in the way the workload is distributed since repartitioning is explicitly invoked after producing key-value pairs: that is essential because the intention is for the pairs with the same key to occur in the same partition. A logical decision is to set the number of partitions after flatMap, thereby the number of reducers, equal to the number of workers (16 in our case). In this case, the workload is perfectly balanced.

As discussed in section 4, persisting the multiplied in quantity set of key-value pairs in memory requires by far more resources, than vectors set in Approach 1. Still, this approach has potentially better scalability, since partitions by key are totally independent for computation and the idea of the approach still will make sense even in case key-value pairs are moved to distributed storage.

6 Theoretical complexity and Experimental performance

In both approaches provided in section 4 the theoretical complexity is determined by the number of vectors in the initial dataset, their dimensionality and is approximately $O(v^2 \times \text{Dimensionality of vector})$, where v is the number of verctos. The reference dataset described in section 2 has 6480 vectors, each with dimensionality of 999. Thus, the total number of unique combinations of distinct vectors is ~ 21 million.

Performance of both approaches was compared on a single machine, which had the following configurations:

- Operating System: Microsoft Windows 10 Home
- Hard Drive: M.2 PCIe Micron 2200S NVMe 1024GB SSD
- Processor: Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz, 2400 Mhz, 8 Core(s), 16 Logical Processor(s), 9th Generation
- RAM: 32 GB DDR4-2666MHz, 2x16GB

The Spark processing was distributed between 16 logical processors used as workers, which enabled parallel processing. The following results were obtained for both approaches and both correlation measures.

- Time required to calculate Pearson's correlation using Approach 1 was 3 minutes 49 seconds whereas it takes 4 minutes 49 seconds using Approach 2. The difference can be explained by the repartitioning and caching overhead of the second approach.
- For calculating mutual information much more time was required due to the higher complexity of the code. Using Approach 1 time consumed to calculate mutual information is 33 minutes 12 seconds whereas it is 33 minutes 50 seconds using approach 2.

We observed that the speed of calculation of an individual correlation for a pair of vectors has a crucial impact on the overall performance. In particular, Pearson's correlation calculation for an individual pairs of vectors (with the same dimensionality of 999, stored in NumPy arrays) takes on average $40.3\text{x}10^{-6}s$ (40.3 microseconds), while it takes around 575 microseconds for the mutual information measure. At the same time, using Python lists as input for Pearson's correlation function yielded 46 times longer computation time: 1880 microseconds.

7 Insights

In Table 1 top 10 highly correlated stocks volume price concerning Pearson's correlation are tabulated. Correlation between volume of stocks of same company in different stock exchange are excluded as it is not interesting and does not add value. Volume of London_*DKG* is highly correlated with London_*EXPN* with Pearson's Correlation coefficient 0.9956. After analysing the data the threshold is decided as 0.9929 for Pearson's Correlation coefficient.

Table 1: Correlation between volume of different stocks in quarter 1

Stock 1	Stock 2	Correlation
London_BKG	London_EXPN	0.9956
London_DLG	London_EXPN	0.9944
London_DGE	London_EXPN	0.9943
London_DGE	London_DLG	0.9940
London_AAL	London_EXPN	0.9935
London_EXPN	London_TUI	0.9934
London_ITRK	London_TUI	0.9934
London_CRDA	London_EXPN	0.9932
London_EXPN	London_MNDI	0.9932
London_EXPN	London_FRES	0.9931

In figure **??** it is clearly visible that volume of London_*BKG* and London_*EXPN* in Q1 is highly correlated.

Moreover, the mutual information between close Price of the same stock in quarter 1 (Q1) and Quarter 2 (Q2) is analysed and the top 10 values are tabulated in table 2. Threshold determined for pairwise mutual information of same stocks in Q1 and Q2 is 8.4487. Mutual information of FTSEMIB between Q1 and Q2 is highest with 8.6294.

Table 2: Mutual Information between Close Price of same stock in quarter 1 and quarter 2

Stock Name	Mutual Information
Mailand_FTSEMIB	8.6294
Swiss-Exchange_SGSN	8.5600
Swiss-Exchange_GIVN	8.5497
Swiss-Exchange_BARN	8.5290
Swiss-Exchange_STMN	8.5290
Swiss-Exchange_FI-N	8.4980
Kopenhagen_MAERSK-B	8.4895
Stockholm_AZN	8.4810
Kopenhagen_MAERSK-A	8.4753
Paris_KER_Close	8.4507

References

[1] Running Spark on YARN - Spark 2.4.5 Documentation.
[2] Spark Standalone Mode - Spark 2.4.5 Documentation.