Blockchains and decentralized applications

Korelace BTC transakcí vůči poskytnutému datasetu

Matej Berezný
xberez03@stud.fit.vutbr.cz

1 Methodology

The correlation pipeline consists of extracting data from blockchain, extracting data from database, feeding them into the correlation algorithms and finally evaluating the results (both numerically and visually). Since the execution time on single thread proved to be fairly slow, I have additionally added support for multi-threading, thus dividing the database data into chunks and passing them through correlation pipelines at once.

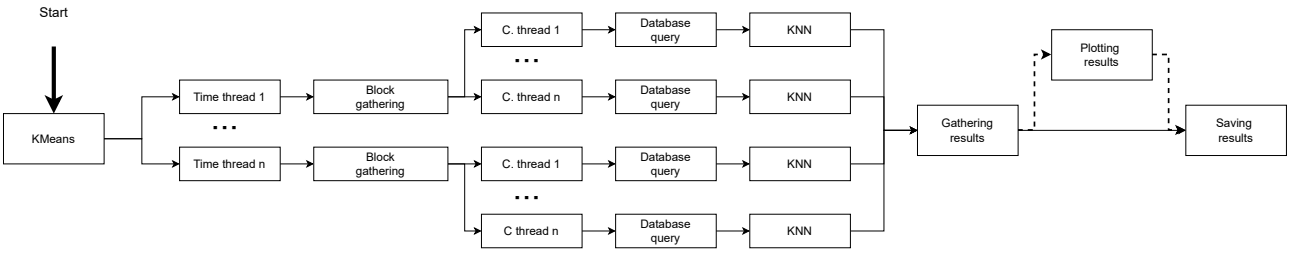

Figure 1: Program's pipeline

Even though multi-threading proved to be efficient in time-saving, unfortunately even threads are still constrained by memory limits, input/output operations and network speed.

1.1 Outliers

In attempt to decrease the load on the API server, I have devised simple method to separate time outlier purchases from the database and process them separately, if needed.

Methods consists of separating all queried timestamps into clusters via K-Means and labeling them (`k` is equal to the number of threads), finding cluster centers and computing euclidean distance from each data point to their respective cluster center. If the calculated distance is higher than quantile of 0.75, data point is branded as outlier, thus left unprocessed if not explicitly requested (via command line parameter). Each cluster gets then processed by separate thread, also called `time_threads`, limiting the range of blocks needed for correlation purposes.

1.2 Extracting data from database

Black market purchases are stored in `PostgoreSQL` database provided by the project's supervisor. For this project's needs, database was set up locally - to remove any additional overhead generated by internet connection bottlenecks.

In each `time_thread`, to further speed the process, additional correlation threads, called are generated. They share the same block pool, and `N` purchases for each thread are selected from table ordered by `time`, so each thread will receive same amount of records. Usually, all data are fetched at once and stored in dataframe, except for the case when `-sequential` option is chosen, then database cursor is set up and fetching is done one-by-one.

1.3 Extracting data from blockchain

To extract data from bitcoin blockchain, I have used API provided by project's supervisor, hosted on university servers. For more detailed description of the API used in this project, refer to the Trezor Blockbook here - [BMK18].

1.3.1 Query

To query specific block from bitcoin blockchain, either block hash or position of the block in blockchain - block height had to be provided. Since approximate block height in time could be inferred rather simply, I have chosen to use block height as query parameter. Approximate height of block in this project for specific time was calculated accordingly:

$$h_s = h_l - \frac{t_c - t_s}{block\_time} \tag{1}$$

Where h_l represents height of the latest block in blockchain, t_c latest timestamp (at the time of script's execution), t_s is the specific timestamp for which the block has to be found and `block_time` is constant representing the average time it takes to mine a block. Since the calculated block height is only approximate, constant `block_offset` is added/subtracted to the block height to further boost chances of specific purchase to be found in block's transactions.

In terms of amount of block queries sent at once, I went with two different approaches - sequentially and all at once.

1.3.2 At once

When the correlation of purchases with transaction in larger time frames / quantities are needed, trying to get block interval for each purchase separately deemed to be highly time inefficient, so I have added option to query all blocks required for certain time interval at once. It is done by extracting timestamps from first and last record located in the database query (ordered by time), calculating their respective block heights, modifying them by constant `block_offset` and gathering all transactions from blocks in such interval into one large dataframe.

1.3.3 Sequentially

Since sometimes it is only needed find blocks for one purchase, or provided hardware is insufficient for querying all blocks at once, I have added the option to query only blocks needed for single purchase at the time, thus limiting the memory requirements at the cost of fairly significant increase of time difficulty. The process is similar to querying all blocks at once, only the interval is calculated from single record representing wanted purchase.

1.3.4 Extracting data from block

Each requested block comes in `json` format further specified in API's github [BMK18]. To extract only relevant transactions from each block, I have added few additional constraints:

- There have to be money incoming to transaction
- There have to be money outgoing from transaction
- Outgoing address has to be different from incoming address
- Outgoing value has to be larger than 0

If transaction satisfies these conditions, its ID, address, value (in satoshis), block time and block height for each outgoing address are stored internally.

1.4 Nearest neighbors

To find the most similar transactions to the purchases, distance between queried purchase and block transactions was computed. Total of six various distance metrics were used to find the most accurate results:

- Euclidean distance

$$dist = \sqrt{\sum_{1}^{n}(x-y)^2} \tag{2}$$

- Manhattan distance

$$dist = \sum_{1}^{n}|x-y| \tag{3}$$

- Chebyshev distance

$$dist = max_n(|x-y|) \tag{4}$$

- Hamming distance

$$dist = \frac{\sum_{1}^{n} unequal(x,y)}{n_{total}} \tag{5}$$

- Canberra distance

$$dist = \sum_{1}^{n}\frac{|x-y|}{|x|+|y|} \tag{6}$$

- BrayCurtis distance

$$dist = \frac{\sum_{1}^{n}|x-y|}{\sum_{1}^{n}|x|+\sum_{1}^{n}|y|} \tag{7}$$

Where `n` equals to number of dimensions present in data. For purpose of finding correlations between bitcoin transactions data points and data points from black market purchases I've decided to use both 2D and 1D space.

1.4.1 1D

In case of one-dimensional nearest neighbor search, the time of the transaction was omitted in the computation and distance was calculated solely on the price values in satoshis.

1.4.2 2D

In case of two-dimensional nearest neighbor search, individual transactions/purchases were represented as a data points with timestamp as a `X-coordinate` and price/value (in satoshis) as a `Y-coordinate`.

1.5 Accuracy metric

To measure the accuracy of the correlations of bitcoin transactions against data from black market calculated in this project, I've devised simple metric, that takes into account both time when transaction was conducted and actual amount of money (in Satoshis) transferred in transaction. Additionally, I've added 2 modifiable parameters that affect the metric's strictness. The whole equation used in metric calculation would then be:

$$acc = 1 - \frac{\left|\frac{p_r - p_{tr}}{b_p}\right| + \left|\frac{t_r - t_{tr}}{b_t}\right|}{2} \tag{8}$$

Where p_{tr} represents satoshis transferred in transaction, p_r represents satoshis used in purchase, present in the provided dataset, t_{tr} is the block time when transaction was processed and t_r is time of the purchase - both are in timestamp format.

Additionally, b_p and b_t are aforementioned modifiable parameters, and they represents boundaries, or cutoffs for both price and time. Thus when the price/time difference is too high, the equation yields numbers lower than 0, so to keep the accuracy in range between 1 and 0, all negative numbers are rounded to 0.

2 Implementation

The language used in this project was Python 3, with multiple libraries (complete list of libraries can be found in `requirements.txt` file). Implementation is divided into 5 separate files, with `load.py` being the main launching script.

2.1 Config file

`config.py` is simple text file gathering modifiable constants for their easy modifications, such as `BLOCK_OFFSET`, `PRICE_BOUNDARY` etc. It also contains both `API_STRING` used for querying bitcoin blocks and credentials used for database authentication.

2.2 Setup script

Main, launching script is called `load.py`. The script is used for procession of the command line arguments, setting up the database connection, separation of the outliers, and `RunnerThread` generation. Additionally, after threads finish, saves the correlations and plots (if desired) the nearest neighbors along with the queried data points onto the graphs and saves them to the `plots` folder.

2.3 RunnerThread

`RunnerThread` is the main body of the project, located in file `runner.py`. It handles everything from database queries through creating additional support threads used for fastening the correlation process, handling of the different KNN classifiers and more.

2.3.1 Database queries

Since every `RunnerThread` has possibility of creating additional 'correlation threads' `corr_threads`, multiple database cursors had to be set, each with different offset but same length to make additional threads even viable.

It was performed by simple `COUNT`-type query into `product__stats` stats table joined by `product__variant__items`, `product__stats` on `product_id` column and by `product__variant__prices` on `variant_id` column. By doing this, the required `length` parameter was computed.

Next, for each `correlation_thread` new cursor for same tables (but with actual columns) was set, each with different `offset` equal to $(n_{threads} - 1) * length$.

2.3.2 Correlations

Since purchase logs in the database being in bitcoin, which sometimes resulted in price value being very low floating point number and blockchain's transactions being in satoshis, all db purchases were converted with rate of `1 BTC = 100.000.000 Satoshis`. Additionally all timestamps and prices in satoshis were converted to `np.int64` format.

Next, for each chosen distance metric, `K` nearest transaction neighbors to the black market purchase were found, and stored only if their computed Accuracy metric was higher than 0.

2.4 Technicals

`Technicals` is class implementing k-nearest neighbor classifiers provided `sklearn` library, calculating the block interval and executing queries to the bitcoin blockchain. The class itself is located in `technicals.py` file.

2.4.1 Blockchain queries

Since queries to the blockchain proved to be very slow if done sequentially, so 2 optimizations were performed to speed up the querying process (one of them mentioned in Methodology). The second method was to send `GET` requests asynchronously, with the help of `asyncio` and `aiohttp` python libraries. Since not all `GET` requests return successfully, I've included few `asyncio` features that somewhat ensure the successful recovery from failure. The async function implementations are stored in `utils.py` file, and as most of the program - are executed in each `RunnerThread` separately.

3 Tests

Three different categories of tests were performed to find the best possible solution for correlating bitcoin transactions with purchases on the black market.

3.1 Speed

All tests were performed with tha same `K` parameter set to `3`, and both timestamps and bitcoin price were used in correlations. As for threading, `2` time threads and `10` correlation threads were used. 3 distinct intervals - 10, 100 and 500 records from `product__stats`, where `sales_delta` was higher than 1 were used. Time results for each metric in are measured via `time` bash utility in seconds, while IO operations related to saving correlated dataframe were omitted in results.

Speed results			
Metric	10	100	500
Canberra	26	420	–
Euclidean	15	61	232
Manhattan	14	61	221
Chebyshev	15	63	231
BrayCurtis	17	66	228
Hamming	689	–	–

Table 1: Speed results

`Hamming` distance had shown steep increase in execution time in comparison to other distance metrics even on the smallest interval, so it was deemed unnecessary and too costly to continue with higher intervals. The rest of the distance metrics had shown somewhat similar results, with small exception of `Canberra` distance's performance being slightly worse.

3.2 Numerical

All tests were performed on first 10 records from `product__stats`, where `sales_delta` was higher than 1 (1M+ matched transactions). So the `-start` parameter was set to `0`, `-end` parameter to `10`, `2` time threads and `10` correlation threads were used. As for the parameters for accuracy function, `price_boundary` was set to `2.5%` and `time_boundary` to `2` hours. Different `K` values were tested on both `2D` and `1D` correlations for every distance metric mentioned in Nearest neighbors.

Numerical results							
K	Dimensions	Canberra	Euclidean	Manhattan	Chebyshev	BrayCurtis	Hamming
1	2	76.44%	94.91%	94.98%	92.84%	94.98%	74.19%
1	1	20.09%	20.12%	20.12%	20.12%	20.09%	0.05%
3	2	30.62%	93.97%	94.17%	93.11%	94.17%	00.02%
3	1	20.04%	20.07%	20.07%	20.07%	20.04%	00.02%
5	2	27.09%	92.69%	92.93%	91.78%	92.93%	00.01%
5	1	20.11%	20.15%	20.15%	20.15%	20.15%	00.01%

Table 2: Numerical results

As depicted in table 2, using price in satoshis as only correlation parameter yielded significantly lower results than finding correlations on both price and timestamp of the transaction. It can be further deduced from the results in table 2, and observed in table 4, in case of 1D correlations, every distance metric (except for hamming distance) found the same, or very similar neighbors. In terms of 2D distance metrics, both `Manhattan` and `BrayCurtis` reached top scores, closely followed by `Euclidean` and `Chebyshev` distance.

Comparison between sequential and "parallel" approach of the `RunnerThread` was deemed necessary, since sequential approach had shown to be way too slow to be worth measuring. Thus it is suggested to avoid sequential execution if not necessary.

3.3 Visual

Visual tests were performed with the same parameters as mentioned in Numerical tests, with only difference being the `K` parameter that was set to `3`. So basically, the visual results show the nearest neighbors for single data point from third and fourth row in table 2.

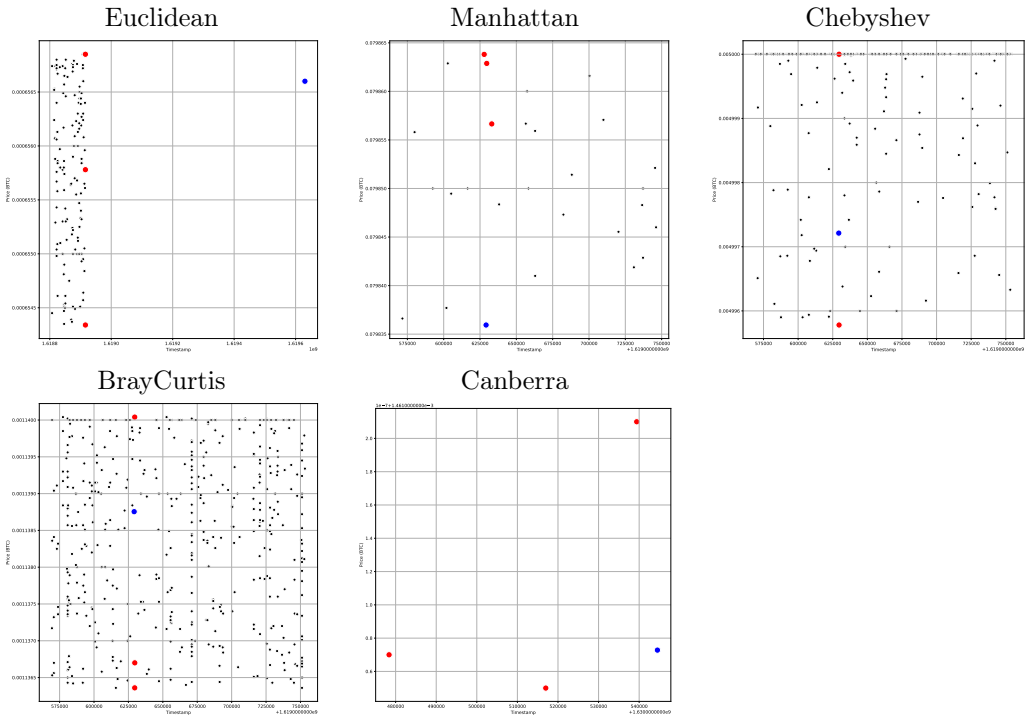

Table 3: 3-nearest 2D neighbors in 2D space for one data point from purchases logs

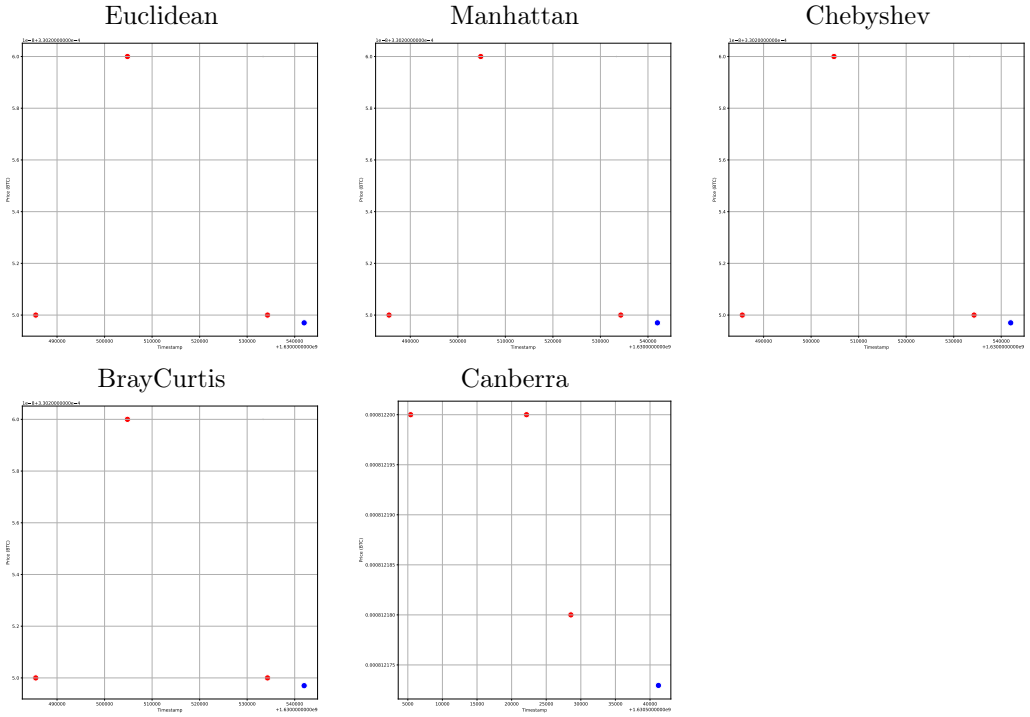

Table 4: 3-nearest 1D neighbors in 2D space for one data point from purchases logs

4 Conclusion

The outcome of this project is automatic tool for finding correlations between transactions in bitcoin blockchain and purchases from the black market. It uses K-nearest neighbors algorithm as its base, and offers plethora of distance metrics, asynchronous block fetching, sequential and parallel correlation approach and many other customizations.

Results presented in Tests section had shown, that `Hamming` distance is not the most useful distance metric for finding closest transactions to the source purchase, `Manhattan` or `BrayCurtis` distance should be used instead, since they outperform others both in terms of accuracy and speed.

References

[BMK18] Martin Boehm, Jakub Matys, and Peter Kracik. Blockbook, 2018.