Prediction Of Crypto Currencies Exchange Rate Using AI Algorithms

by:

1. **Nerya Hadad**
2. **Inbar Gera**

Background and Problem description:

Crypto-currencies are digital means of exchange, created and used by private individuals or groups.
Because most cryptocurrencies are not regulated by national governments, they are considered alternative currencies – mediums of financial exchange that exist outside the bounds of state monetary policy.
Nowadays, most of the use of those currencies is for trading, Successful prediction of their change direction might be translated to money.
Our purpose is to create an algorithm for investing in crypto-currencies that will yield the best return on investment.
In every hour, for every currency, the system will decide whether to go long on the coin (guess it will go up), short (guess it will go down) or to do nothing (if the prediction is uncertain).

Project Report Structure:

The project itself is compound of 4 major steps:

1) Fetching and cleaning the data
2) Transforming raw data into features for the learning algorithm
3) Developing and training the algorithm itself
4) Grading the results

The report is constructed in the following was:

1) Description of the high level of the journey we went through steps 1-3, and then reference to the code that is responsible for each part.
2) Step 4, and the experiments we conducted in a detailed manner.
3) Summary and future research.
4) Appendix - Full description of the program in relation to the explanation in this report.

Data sources and Preprocessing:

In order to train and predict using AI algorithms, data is a must.
In this project there are 2 types of information:

1) Technical information regards the value of a coin.

2) Human related information.

For the first category, historical data regarding coins exchange rates is used.
For the second, there are 2 data sources, which indicates human behavioural information:

1) Google trend - the amount of times a specific coin related term has been searched for at a given time.
2) Twitter mentions - same as above, but with tweeted term instead of search count.

Raw Data:

Searching for the smallest granularity (time between two entries in the dataset) available online, we chose intervals of one hour, the whole data is fetched from this site: www.cryptodatadownload.com.

The first task was for each coin, to get the historical exchange rate of all of the datasets, and to aggregate them to one coherent dataset.

The desired data format contains the following entries with fixed format:

1) Open, Close, High, Low - exchange rates of the coin during the corresponding hour.
2) Volume - the amount of coin bought and sold during this period.
3) UnixTimestamp - the time the hour began.

most of the markets didn't contain all the required information for most of the coins.

Thus, we had to use a lot of different markets and aggregate the data to get the most accurate values.

For start, we took the most old and recent time that is available at all the datasets, and for every dataset - padded empty entries in this time period with Null value.

The problems we encountered during the preparation of the datasets before the aggregation process are as follows:

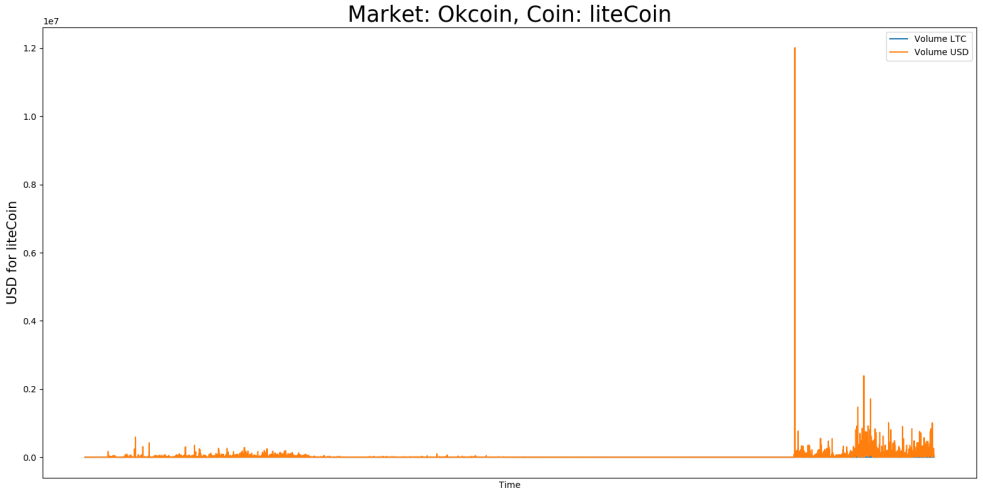
1) The data format of the different datasets was not the same:

Binance_ethereium.csv | gemini_ethereium.csv

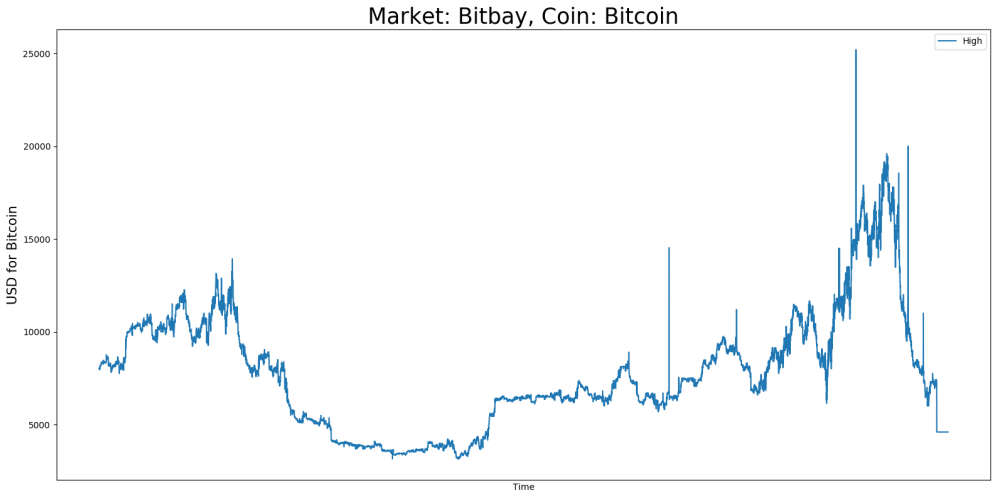
```
1  Timestamps are UTC timezone,https://www.CryptoDataDownload.com
2
3  Date,Symbol,Open,High,Low,Close,Volume ETH,Volume BTC
4
5  2019-10-17 09-AM,ETHBTC,0.0219,0.02198,0.02189,0.02193,4573.28,100.31
```

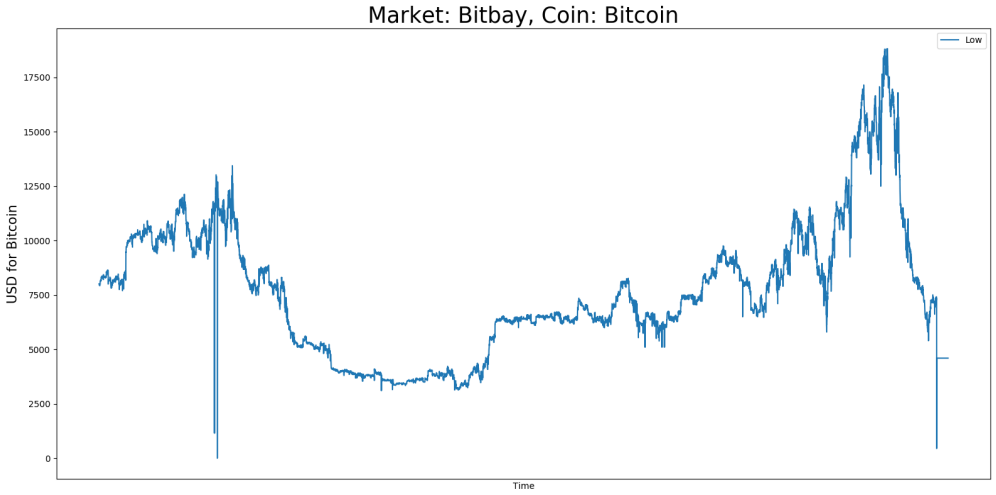
Binance_ethereium.csv | gemini_ethereium.csv

```
1  Created on 10-27-2019,https://www.CryptoDataDownload.com
2
3  Unix Timestamp,Date,Symbol,Open,High,Low,Close,Volume
4
5  1572134400000,2019-10-27 00:00:00,ETHBTC,0.01927,0.01927,0.01927,0.01927,0.0
```



2) The data sometimes contained entries with Volume=0 (last graph).
3) Data with Unrealistic high\low:





4) The currency in which the exchange rate was presented was mostly USD, but for many datasets it was Bitcoin and Etherium.

Our solutions:

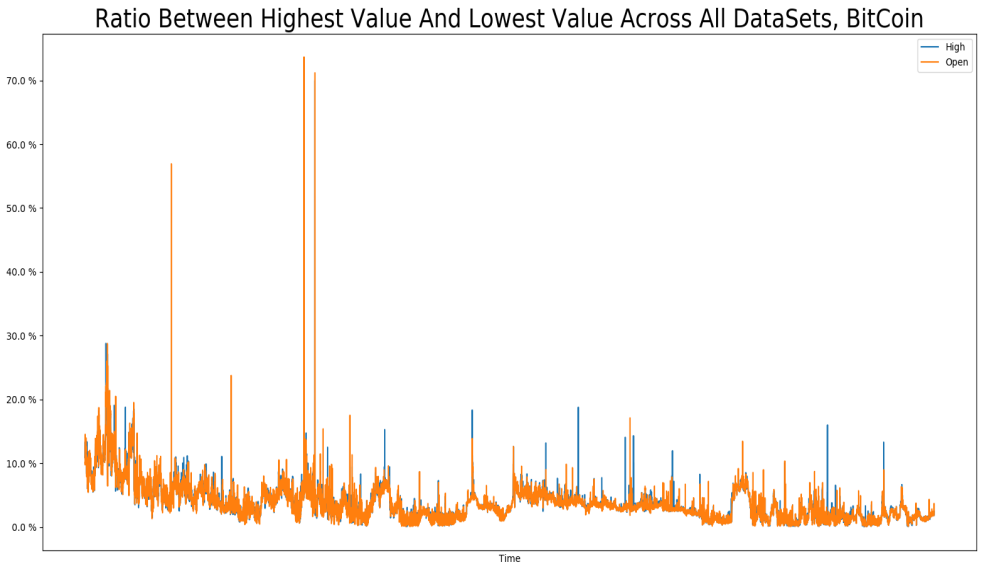
1) Standardize the time using Unix Timestamp, which was extracted from the date (string of the date). For volume we merged the sales and purchase volumes to one volume.
2) Every entry with volume=0 was marked as invald, and ignored in the next stages.
3) For each entry we used following verification checks, and if it failed (true), marked it as invalid:
    a) High value is bigger than Open and Close by more than 20%
    b) High value is lower than Open and Close by more than 20%

Additional conditions we checked are that volume is not negative, and that Open and Close are within the bounds of High and Low, those two conditions were always OK though.

4) As most databases exchange currency was USD, we used the following solution:
    a) First aggregate all datasets whose exchange rate is USD, and just ignore other datasets.
    b) Translate all currencies exchange rate to USD using the aggregated data.
    c) Now, when all the datasets exchange currency is USD, repeat the aggregation process with all of the datasets.

<u>Aggregation:</u>

Motivation for good aggregation: many cryptocurrencies exchange rates have huge arbitrages (about 5% at some time for a whole day) from one market to another.
Example: (for “High” and “Open” values)



The aggregation of each entry is conducted in the following way:

If an entry in a dataset was invalidated in previous steps - ignore it.
High is the highest High across all the datasets that had any trade volume during this hour.
Low is the same, just the minimum.
Volume is the sum of all volumes.

Open and Close are calculated using a weighting heuristic score:

For every entry the weight was the total volume of trade during the last week, and eventually the relative weight was its proportion in the overall weights of all of the datasets.

We chose this heuristic for two reasons:

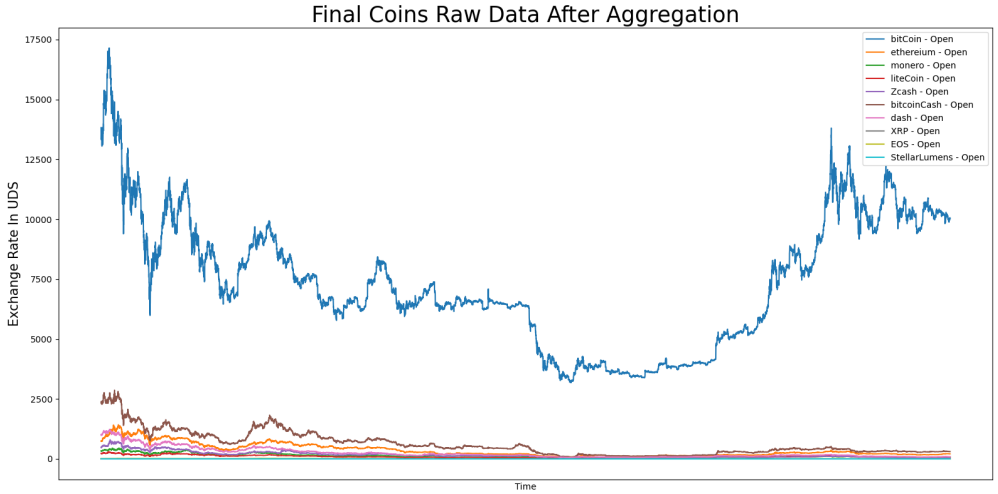
1. It represents the popularity of a market in a good way.
2. To smoothen the data: as there are large arbitrages between different markets, if at one point a big market has a small volume due to some human related reason (holiday in a country, night time, etc), the other markets suddenly will have a dominant footprint, and this will cause a spike in the graph due to the arbitrage difference and not due to the real change in value.

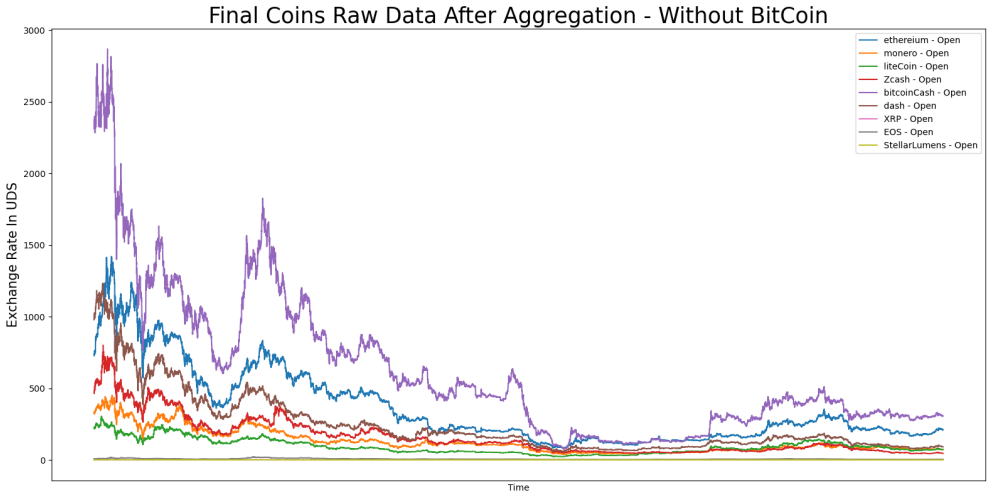
If at this point there were still bad entries (entries which were bad at all of the datasets):

If there was more than 24 hours of missing data in a row, it's considered as bad data and raises an error. Otherwise, fill those entries by the value from the previous hour (with volume 0).

This helped us choose a time frame and coins to conduct our experiments on, as some recent times and other coins had this problem and thus considered (by us) as bad data.

Those are the results of the aggregations ("Exchange rate in **USD**"):

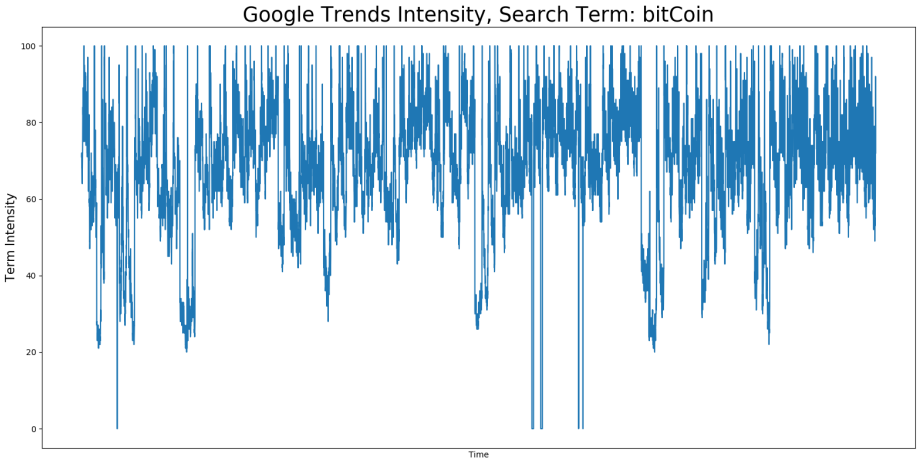




The code that downloads the datasets is located at downloadData/dataBases.
To get the aggregated data run genData/aggregateDataSets.py file, and look for the results at <communication path>/<coin name>.py (for every different coin).

Google Trends:

Another data source was Google Trends.
Google saves a lot of data regarding its users search history.
This data is freely available at a website google provides, named google trends: https://trends.google.co.il.
Google trends make it possible to get the intensity of search for a word in a specific time and variations on.
Example:



Such data contains information regarding behaviour of a big mass of people, and as such it is suspected (and later proved) to have value for our algorithm.

We used a web scraping python library (detailed in external libraries) to fetch the related data (number of searches for the target and the target price in every hour).

For every coin “coin” we got the intensity searches per hour of: “coin”, “coin price” and the ratio between those two.

The reason for the first two is to get the public interest in the coin, and in its exchange rate.

The reason for the last one is to create a feature that represents the ratio between supply and demand, as we know that the price is where the number of sellers meets the number of buyers. Since we believe that coin-price is more likely to be searched by people who already bought the coin and are looking for a good time to sell it, the ratio between the two might indicate the above.

For getting the Google Trends data run downloadData\googleTreds.py - notice that this part has some known limitations, so please refer to the instruction first.

Twitter mentions:

The last data source is Twitter: https://developer.twitter.com/ (genData/tweets.py) We wanted to derive twitter mentions of the target during the time intervals we were working with.
Tweeter provides a convenient API for registered users to fetch this data.
The problem was, that this API allows access for data of up to a few weeks ago, while our data is ~2 years. We decided that we better use long time with no twitter mentions, than using just a few months at most for train, validation and test. So in this project we won’t use twitter mentions.

Data Splitting:

As opposed to other prediction problems, exchange rate is in nature extremely chaotic. For every given period of time there might be a distribution that is able to explain the data in a convincing way, but it is just forcing the model over the data.
The common approach to this problem, and the one we went with, is to train on a period of time, and then to test the results over a period of time in the future.

We split the data into 4 parts:

1) train
2) validation 1 (referenced as “train-validation”)
3) validation 2
4) test

The splitting is done in such a way that every set is the chronological extent of the previous set, in such a way, we ensure that there is no effect of looking into the future at any step.

Generating two validation sets is done retroactively when developing the classifying algorithm itself: there are two steps there, where the second part is verifying the first part, so doing both parts with the same validation set makes no sense due to obvious overfit.

External libraries:

- **pytrends** - The purpose of this library is to get information from google trends website.
  Difficulties:
  1. Sometimes Google website blocked us as a result of overuse of the api, and sometimes we get an error since the bandwidth was not good enough.
  2. This library has a bug - when looking for hourly info of more than a week, the library returned twice the same hour (00:00) on the seven'th day, and from this hour and later - the results were wrong and undetermined.

  coping:
  1. At the rest of the code, when running a specific logic part, the logic is applied again and then overrides its previous results.
     In our case we left previous work as is, so when the program reached the request limit it gracefully noticed it to the user and exits.
     Next time the downloading part has been run, we check what work is done and continue from there instead of overriding it.
     That way we just need to run the download part a few times until all of the data will be at home.
     Finally we implemented a code that goes to sleep for 15 minutes when Google servers are blocking us, and they try again, repeating this loop until all of the data is present.
  2. Instead of requesting all of the data in one big batch, we divided the requests into chunks of 6 days, and for each chunk we sent a separate request.
     When the request has returned, we checked its validity, and merged it to the rest of the chunks, (and when all of the chunks are received, we checked that the merging of them contains each required entry exactly once).

- **ta -** The purpose of this library is to produce technical-analysis out of dataframes\files with information on prices with constant time distance between every instance (hour in our case), for every hour we need to have the following info: `open, high, low, close, volume`. We thought that it would be better to generate all features available in the library, and later filter the relevant ones. And so we did.

Feature Engineering:

After the raw data is ready, it is time to transform it into features that the learning algorithm can work with.

Since we are trying to predict the prices of cryptocurrencies, the historical prices reflect a lot of information on the prices behavior. When looking at those prices we need to emphasize the relative changes instead of the actual exchange rate (avoid the effect of the price-mean), also we need to use only casual features.

This Field is called Technical-analysis and thoroughly explored. We decided not to invest time for new exploration, but rather to use the common Technical-analysis tools - Technical-Indicators, and to explore how to produce the best out of them.

These are the steps we took in that regard:

Generating features:

As explained above, simply generates all technical-indicators that “ta” library can produce.

Another type of feature is based on the number of searches of a coin in google. We were trying to derive from this: the interest in a coin and the supply and demand ratio of it, this is detailed at “data sources and preprocessing”, under “Google trends”.

Another feature type than we thought to create are based on the correlation between different coins:

The hypothesis was that because different coins are very correlated, there might be a phenomenon that some coins are following the direction of other coins, but after a delay of some time.

In such a case, we might be able to use it in order to predict the change in the “following” coins.

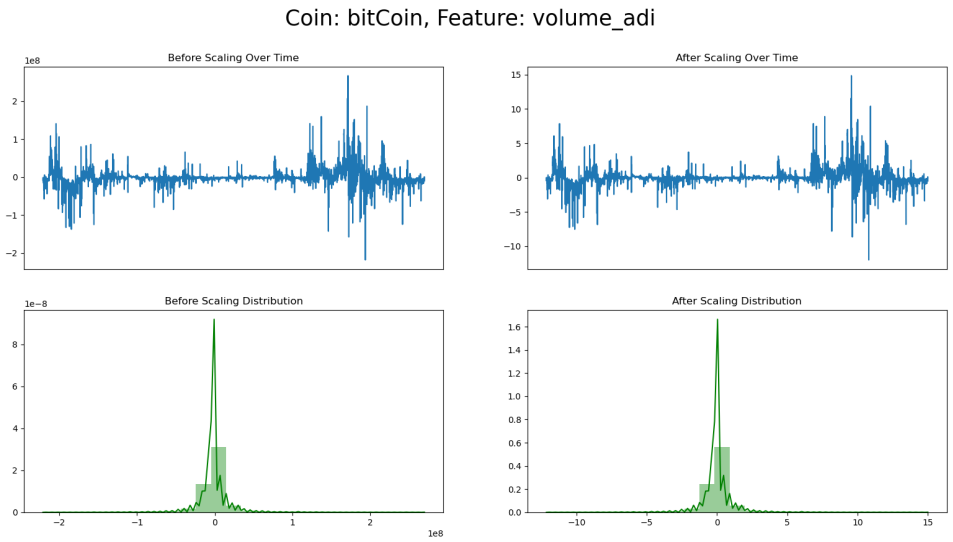
This hypothesis was thoroughly explored in experiment number 1, and the result was that such types of features will not have informative value for the classification algorithm.

Features scaling: (genData/normalizeFeatures)
Motivation: Since most of the ML algorithms use Euclidean distance between two data points in their computations, the features with high magnitudes will weigh in more than features with low magnitudes. To suppress this effect we need to scale the features.
We created a code that given a dataset and a feature, decides whether to normalize or standardize the data and runs respectively.

- Choosing transformation and values (min, max, std and var) is done using the train dataset and implemented on all of the sets.
  The method for choosing the distribution is done using probability density function analysis:
  For a given finite set we can compute its density function.
  Afterwards, we compare this density function with the density function of a known distribution (sum of squared distances at chosen points, for example), and choose the one with the lower deviation from ours function.

- According to the more suitable transformation:
  Normalization brings the value between 0 and 1: $\frac{value - min}{max - min}$
  Standardisation replaces the values by their Z-scores: $\frac{value - mean}{std}$

Examples from the most popular features is as follows:



Feature selection:

Motivation:

Due to the chaotic nature of currency exchange rate, success rate of price prediction is usually not high. Thus, we need to avoid overfit to features who can “predict” the validation set in relative high success.

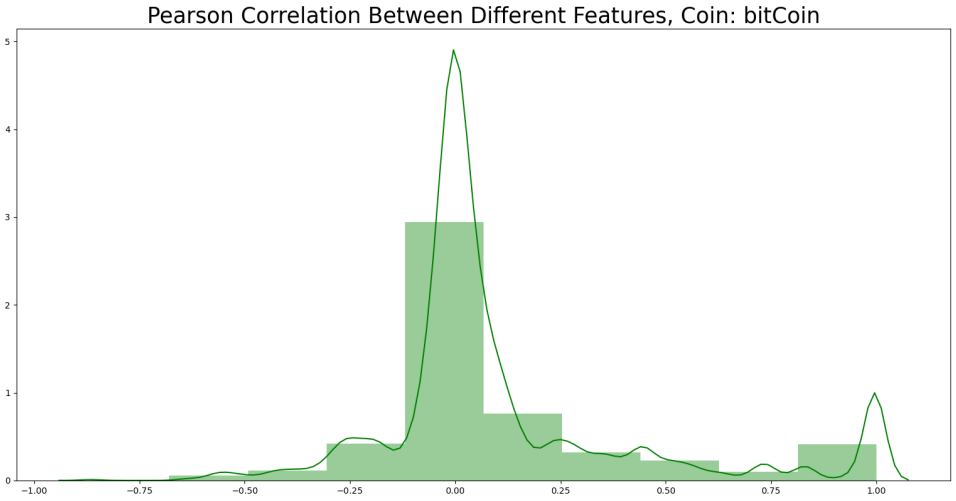
Furthermore, we are not totally familiar with all the features, so knowing in advance which features has a tendency for overfit due to its actual meaning was not a viable option for us.

Notice: In this stage, whenever we use an algorithm which requires validation data, we use the first validation set.

For every coin we did the following steps for selecting features:

1) Remove junk features (all got the same value, infinite value etc’).
2) As most of the features are statistical generated features, we suspected that a lot of them will be highly correlated, and as such will do more harm than good (confuse the learning algorithms and will not yield any additional information).

In order to do this filtering, we calculated Pearson correlation between all features, and for every set of correlated features with more than 0.9, only one was chosen and the rest was removed.

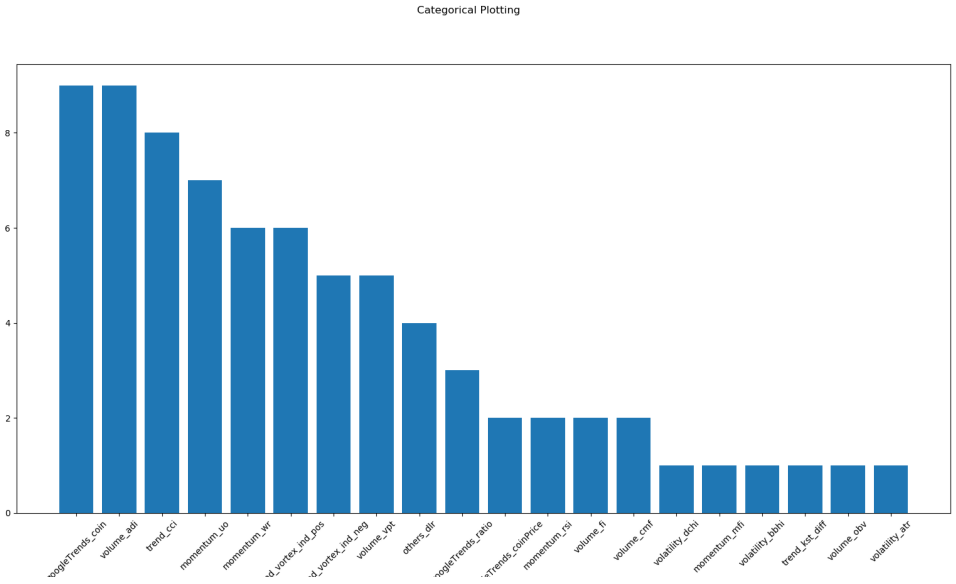


This part reduced the feature count from ~60 to ~40, this indeed supports our hypothesis, as pearson correlation of more than 0.9 indicates very close correlation.

3) We used choose-k-best method:
This step is explained in detail in experiment 2 section.
In summary - for each k from 2 to 11, we got 3 sets of k features, selected by the select k algorithm with 3 different metrics.
For each set, we evaluated the prediction of 3 different classifying algorithms (judges) that have been trained and tested on a data set containing only the k features.
Finally, passed on the set of features which got the best score according to the judges.

4) After the appropriate k features were selected, we used another final selection mechanism, which is a Greedy hill climbing over the features domain.
The greedy hill stops when the increased added value for another feature is very close to 0.
The scoring algorithm was a basic decision tree, because it trains very fast and Greedy hill climbing itself is time consuming.
The score itself is described in the scoring section.

After the above, this is the histogram of the features that remains (one point for every coin that uses this feature for his classifier):



As we can see, the 4 most significant are googleTrends_coin, volume_adi , trend_cci, and momentum_uo, this is how they are distributed across all of the 10 coins.

While the top feature is the regular Google-mentions (all the classifiers are using this feature), 3 classifiers using the ratio of google mentions, and 2 using the <coin price> google mentions. This is actually very high, since the search popularity of <coin price> is low and has a lot of zeros, and some coins even don't have this at all, means ~3 is the estimated number of good <coin price> features.
Since the ratio depends on <coin price>, the conclusion is that the regular google mentions and the ratio are relatively great in prediction.
A possible explanations of this:

- While technical indicators represent people who are already in the "game" of trading, google-mentions takes into account people that aren't in the game, hence there is no sign for them in the graph.
- Google trends data is not available to a lot of traders, and thus it is not explored and used by others.

Another observation - some technical indicators used more than others.
We believe some indicators are built in a way that our greedy selection features algorithm is inclined to choose. And the ones that are rarely chosen are already represented by the first ones and do not provide much additional information.

Scoring metering:

Before the description of the classification algorithm itself, it makes more sense to first describe the scoring mechanism we used, as they are not straight forward.

In order to evaluate the algorithm, there are two questions that came to mind:

1) Which metering to use in order to evaluate our performance?
2) How to compare and check if the results are good?

Evaluation of our results:

At first we used two simple classifiers:

1) Random classifier - we need to win random guesses in order to even be considered as a learning algorithm.
2) “monkey” algorithm: in every hour, the algorithm invests by the same result as the previous hour, this algorithm is very basic, and scoring above it is a pretty basic requirement for a successful algorithm.

After looking at the first results, we observed that the “monkey” made right guesses in less than 50% of the times:

```
Monkey:  0.468
Random:  0.501
```

The complement of the monkey (invest if the coin is reduced in value at the last hour and vice versa), is also an algorithm to compare with, as its simplicity is the same as the original monkey, but it provides better results.

Our classifications overcome the three of them, which is described at the experiment section.

Metering for Evaluation of our results:

We used 3 scoring metering methods:

1) Average hit rate: **the percentage of correct classifying**, we used this metering for every tuning and comparison through the algorithm.

2) Average increased score: this method is more related to real investing:
As the main purpose of the algorithm is to yield profit for its user, we wanted to check how much profit was done, instead of how many times the algorithm predicted correctly.
The first take on this score is the average percentage of revenue [across all the times we decided to invest (long or short)].
Since the profit depends on the time it is conducted (when there is no change in the currency, no profit can be made), we normalized this score using an oracle, which knows the future and invests in the right direction every hour.
Means, the score is the average increase in revenue across all guesses, divided by the same score of the oracle (when investing every hour).

3) The third metering method is an adaptation of the second method.
When we introduce the ability to invest only if we are sure above some threshold of certainty, the algorithm invests with relatively low frequency, but when it does, it invests mostly in times when the change is high.
In this case the average profit per investment (on validation set) exceeds the maximal average profit of the oracle, because the oracle is also investing in low change times, which lowers its average change.
Those are the **average** results across all coins in **validation set** in case of high threshold:

```
Averages:
averageHitRate scores:
Our Clf:  0.848
Monkey:  0.468
Random:  0.501
averageIncreaseScore scores:
Our Clf:  2.588
Monkey:  -0.021
Random:  -0.016
```

As the major mission of the algorithm is to earn as much money as possible, being able to predict a few correct times might be less profitable than to predict more and to have a higher failure rate.

To handle this problem, we created a slight **adaptation of the second metering method, where the average increase in score is calculated over all of the hours**, even times that the algorithm did not invest.
That way, too conservative algorithm will be punished.
Notice: The high success might be the result of overfitting: The classifier was chosen according to this validation set. Since we used a high and optimized threshold - the algorithm invests only in edge cases, which probably has high overfit.

Classifier Construction:

The algorithm construction is conducted of two parts:

1) For each coin, find the best classifier:
    a) For each type of learning algorithm, create a tuned classifier.
    b) Choose the best classifier among the tuned classifiers.
2) For each best classifier: find the best investing threshold.

In order to avoid overfit, we used different validation sets for finding the best classifier:

The tuning is done using training with the train set, and validation with the validation 1 set.
Choosing the best tuned classifier is done by training over the train+validation 1 sets, and selecting with validation 2 set.
The tuning is done using exhausting search over the parameters, it is described in details in experiment number 3.

After the best tuned classifier was found, we added another layer over it to get better results in regards to the scoring meters.
This layer is certainty - now we introduce the option of not investing in a given hour: for every threshold of certainty, if the classifier predicts that the coin value will change at a rate lower than the provided threshold, it will choose not to invest in that hour.
The process of finding the best threshold is done in experiment 4.

As a result of the market's nature, the hit rate success is not expected to be significantly higher than 50%. Thus, overfit has a high effect even though we used 2 validation sets. Thus, we suspected that the optimal threshold is mostly overfitted, so we added thresholds of 0%, 1% and 2% at the final test.

Experiments:

First experiment:

Target temporal correlations

For better understanding of the target patterns, we will examine the targets auto-correlation function in the train period - it will give us quantitative results that might be used in the learning process.

The target auto-correlation function C(d) measures the temporal correlation between the coins y[n] and z[n], where z is shifted by d hours - z[n+d]:
$C(d) \equiv corr(y[n], z[n+d])$, where $corr(a,b) \equiv \langle (a - \langle a \rangle)(b - \langle b \rangle) \rangle / \sigma a \sigma b \in [-1,1]$

Correlation of 1 between two variables means they are completely correlated, while -1 means they are completely anti-correlated, they move in opposite directions.

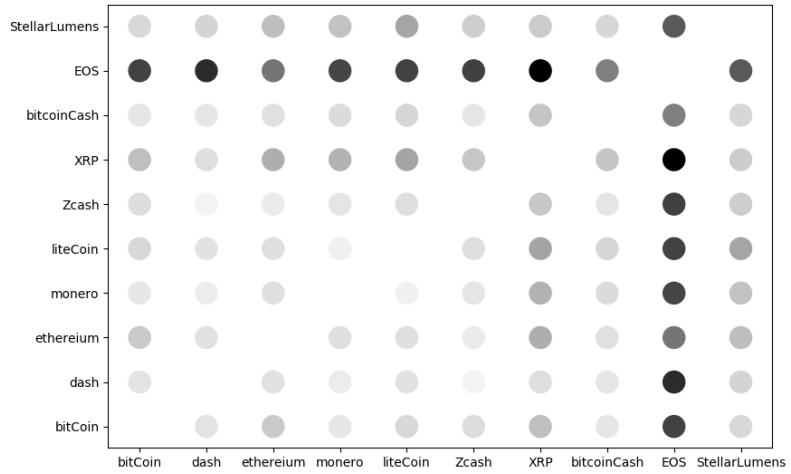
We can use the auto-correlation to find:

- **Causal connection between y and z** – a peak in C(d=delay) means one might predict the behavior of the other in delay of d hours.
- **Periodic patterns** - a peak in C(d=period) for y=z, in our case we will not use it, since the features of every coin with no use of another one are automatically generated.

Highest d can't be too high since we want to use it. Since we work with intervals of one hour, d>400 won't be helpful.

We calculated max(c(d)) for all coins's pairs:

results:
`median: 0.929, mean: 0.886`



Most of the highest correlation received with no shift, though there are 11 excepcions, the ones with highest correlation are:

bitCoin ethereium 0.90 , 17

dash ethereium 0.95 , 126

ethereium Zcash 0.96 , -17

ethereium XRP 0.85 , -151

ethereium bitcoinCash 0.94 , -19

ethereium StellarLumens 0.88 , -69

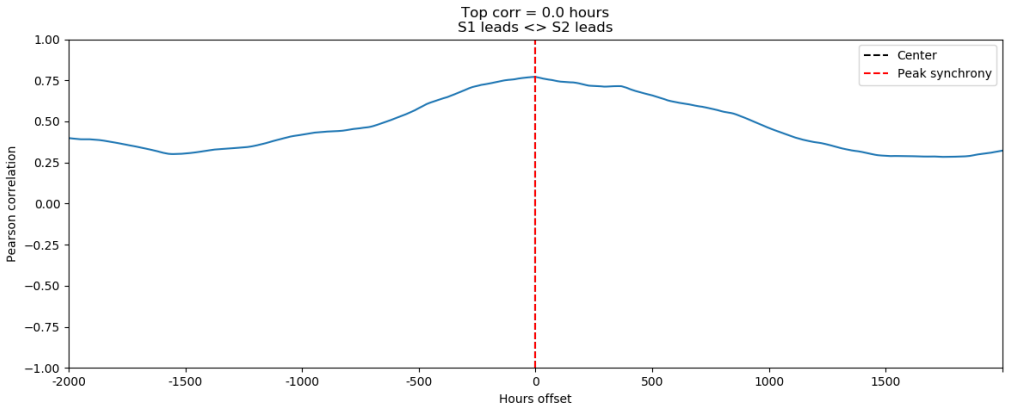
Zcash XRP 0.90 , -70

Findings:

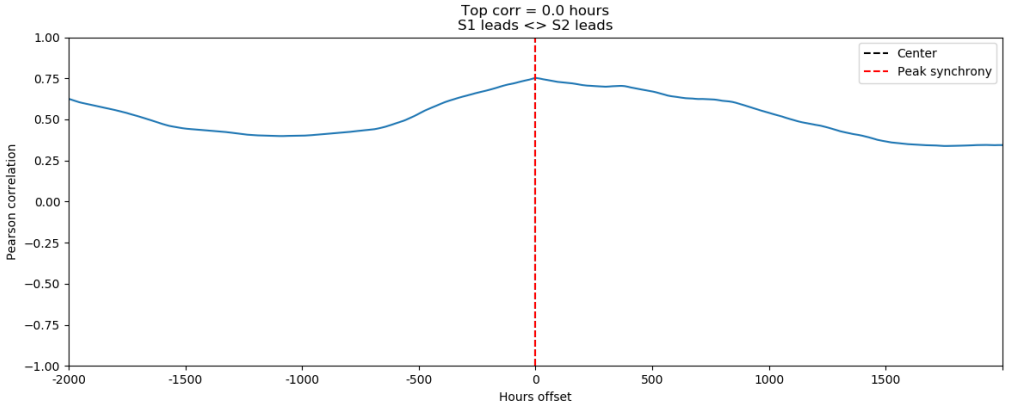
- the highest correlation usually accepted with no shift.
- High correlation between different coins.
- Highest correlation with shift is almost always with Ethereium.

Lowest correlative coin is EOS with:

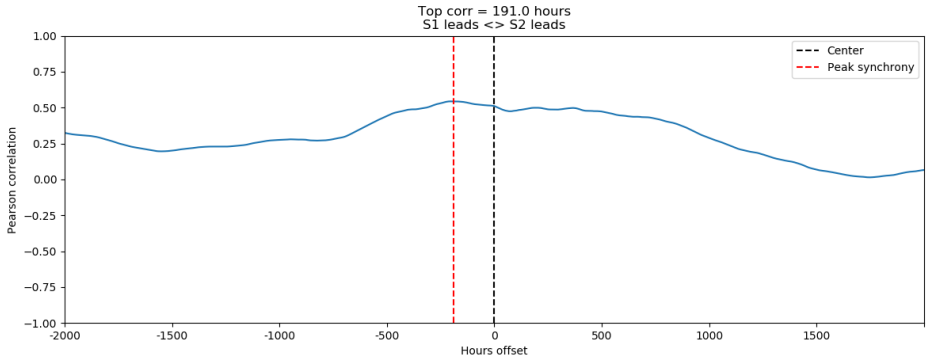
Highest: EOS , bitcoinCash || 0.771 || 0, #s1, s2 || corr || shift



EOS , ethereium || 0.752 || 0

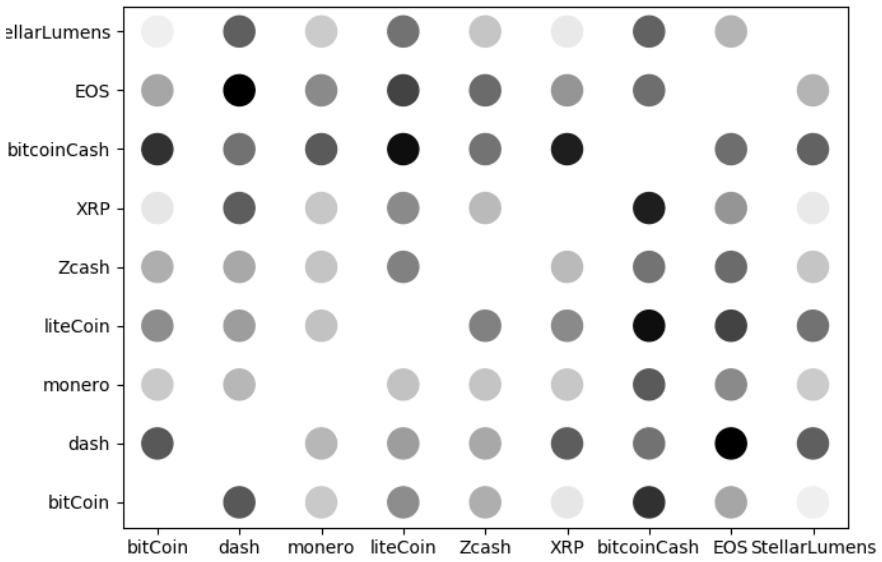


Lowest: EOS , XRP || 0.5431699647591385 || -191



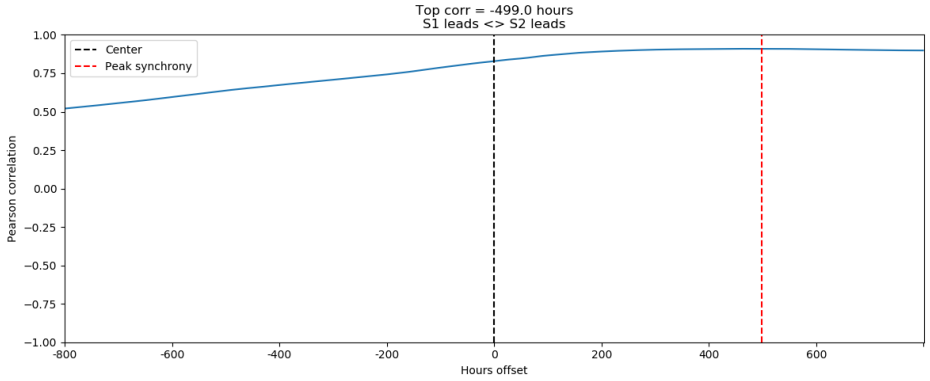
After some research, we found out that the reason Ethereum has such high correlation with all of the coins is because most of the coins consist of some Ethereum (builded on top of the Ethereum blockchain).
As a result, all the coins are correlative, since part of them (part of every coin) is exactly the same, for this part, we can invest directly in Ethereum.
To find out the real correlation between the coins (the correlation of the different parts), we exchanged their prices from USD to Ethereum.

Results:
Correlation is dramatically down:
`median: 0.649, mean: 0.628`

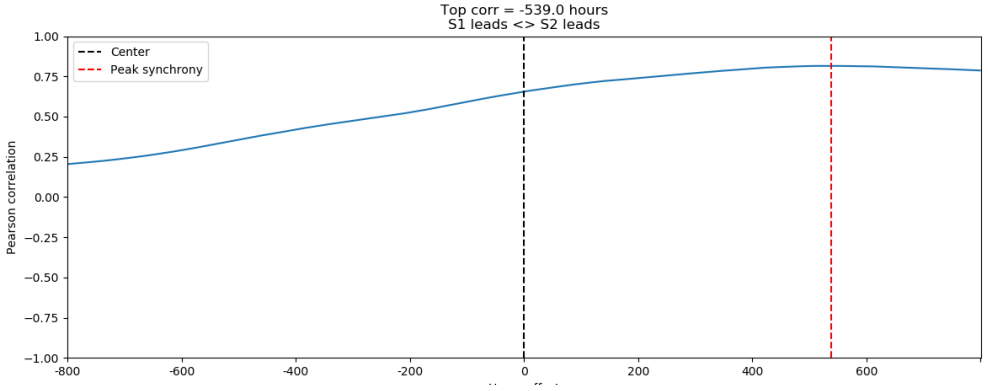


Now the highest correlation perceived is in a lot of cases with delay, which may be helpful for prediction, for example:

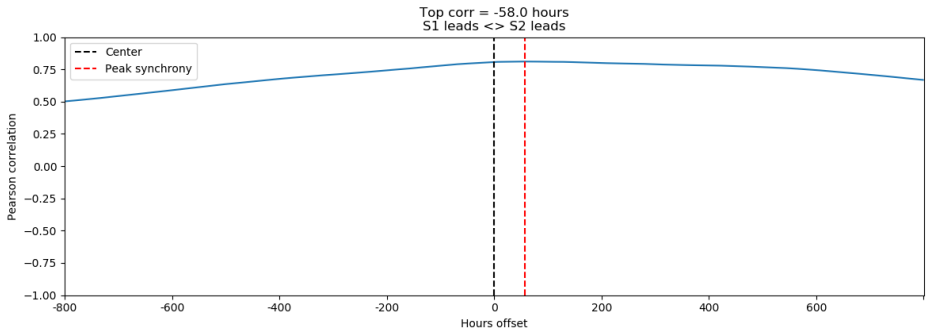
bitCoin XRP



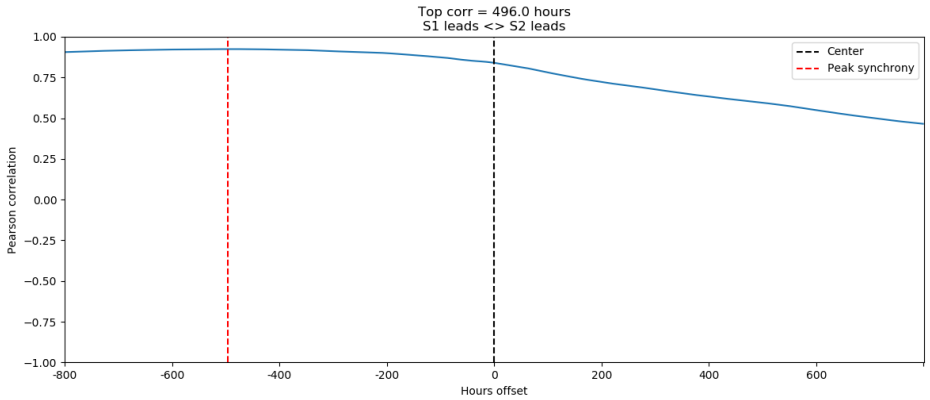
`monero XRP`



`monero StellarLumens`



XRP StellarLumens

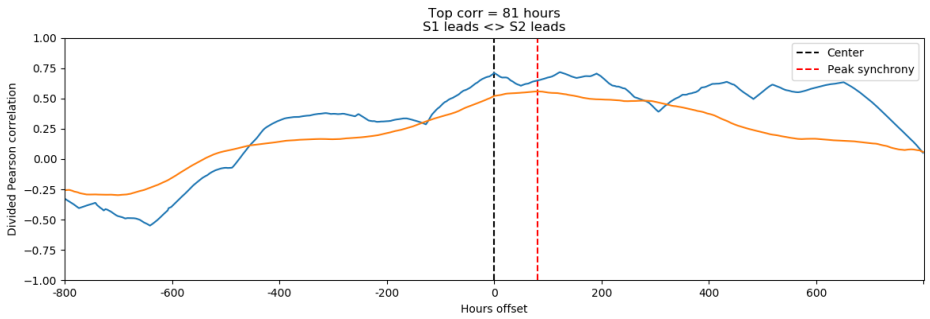


Before we deduce that they have such a great correlation and shift, let's consider other options:

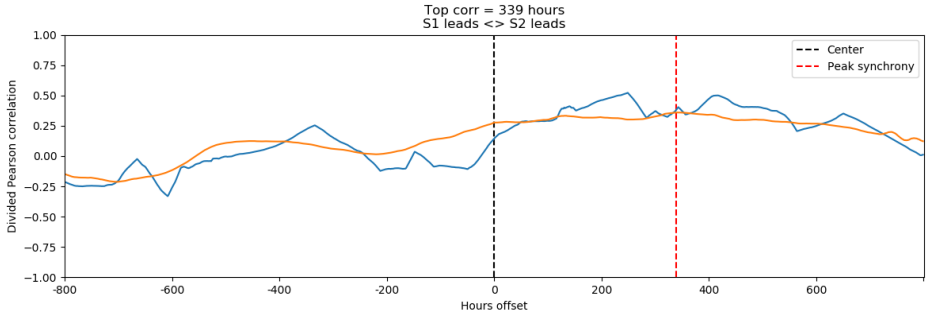
A. Since we look at price correlation in long periods the prices in some periods can be far from the mean relatively to the change in prices, so the change in price effect on this correlation is low. Thus, the change in price correlation might be low, but prices correlation is still high.
B. The high correlation is the result of the fact that the price is in Ethereum currency, so if Ethereum goes down the rest will go up, and visa versa.

A. 1. In purpose to check this hypothesis, the data of every 2 coins were divided into 9 pieces. Then for every shift we checked the correlation for every 2 suitable pieces.
Now the noise of high\low values will affect only ninth of the results.
For every 2 coins we took the median and average of the 9 pairs, as shown at the following graphs.
Let's look on the last 4 pairs which have very high correlation with shift:
Blue – median. Orange – average.

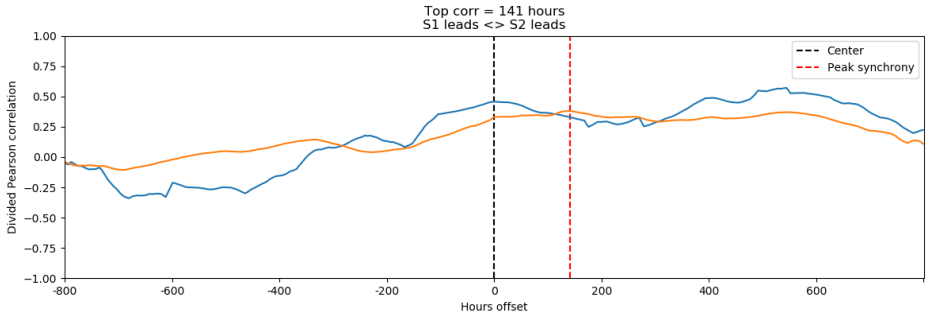
bitCoin XRP



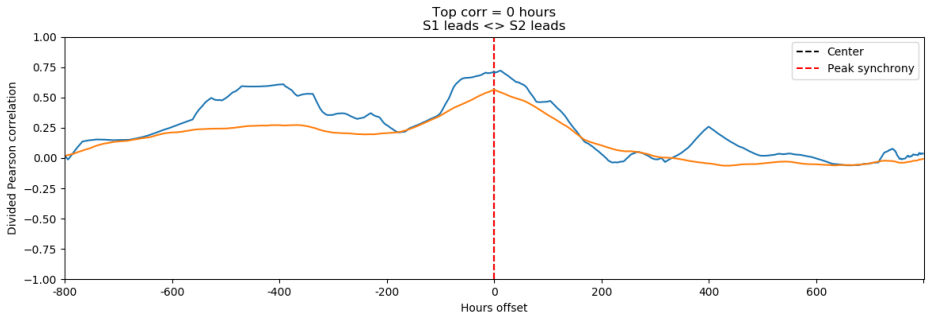
monero XRP



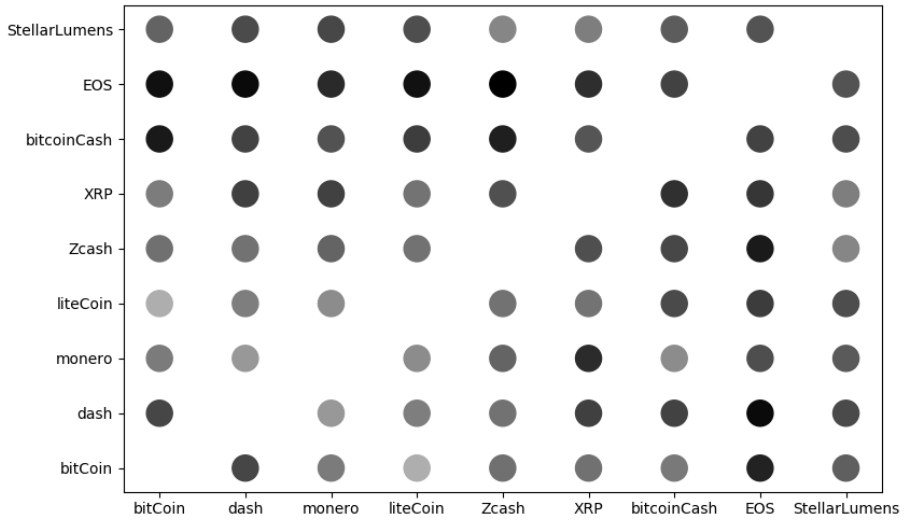
monero StellarLumens



XRP StellarLumens



As we can see the correlation is lower and the shift is different.
Conclusion: the high correlation and shift was the result of prices correlation and not of the change correlation.
Results for all the coins:
`median: 0.495, mean: 0.444`



Findings:

1. The correlation is lower than before.
2. The highest correlation received in a lot of cases with shift.
3. The shift is different (from before) and not with the same coins and hours.

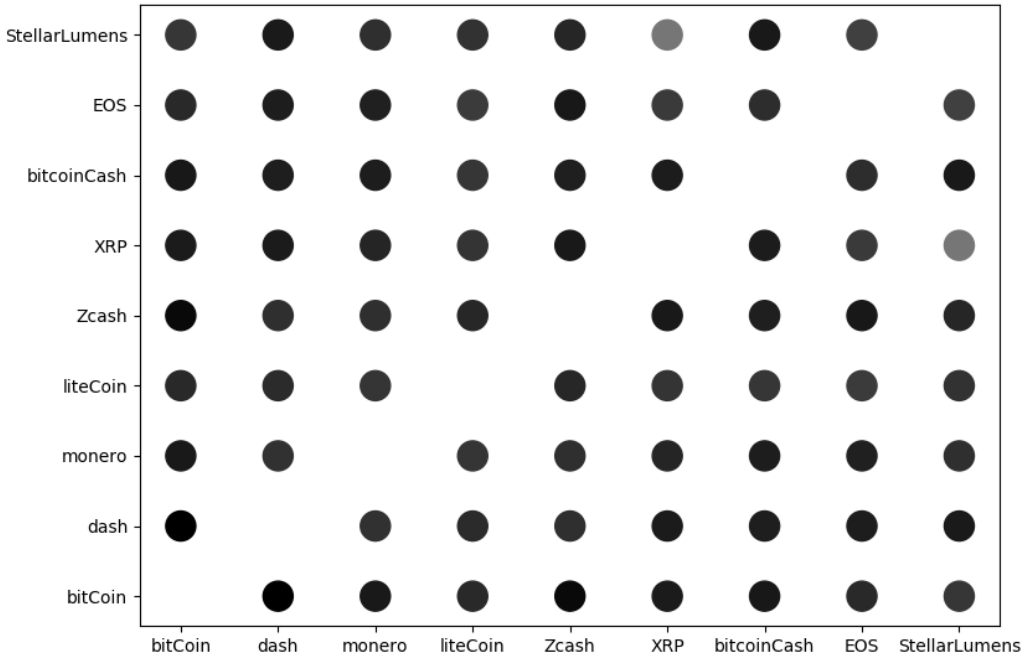
Explanations:

1. This is a result of the fact that in shorter periods the prices are close to each other and close to the mean. As a result, the effect of the change in price is higher and the correlation is closer to the correlation of the change in price. It strengthens the hypothesis of option A as the explanation for the phenomenon.
2. Result of the way we measured, since the correlation is not high, one part with high correlation can make a relatively big difference on average with the correlation of it's shift.
3. This corroborates the suspect that the shift before was the result of prices correlation and not change correlation.

In light of the previous explanations, Let's now check directly the correlation of the **relative change** in price, this will avoid any effect of the prices correlation.

A. 2. **percentage change correlation**:

`median: 0.219, mean: 0.3`

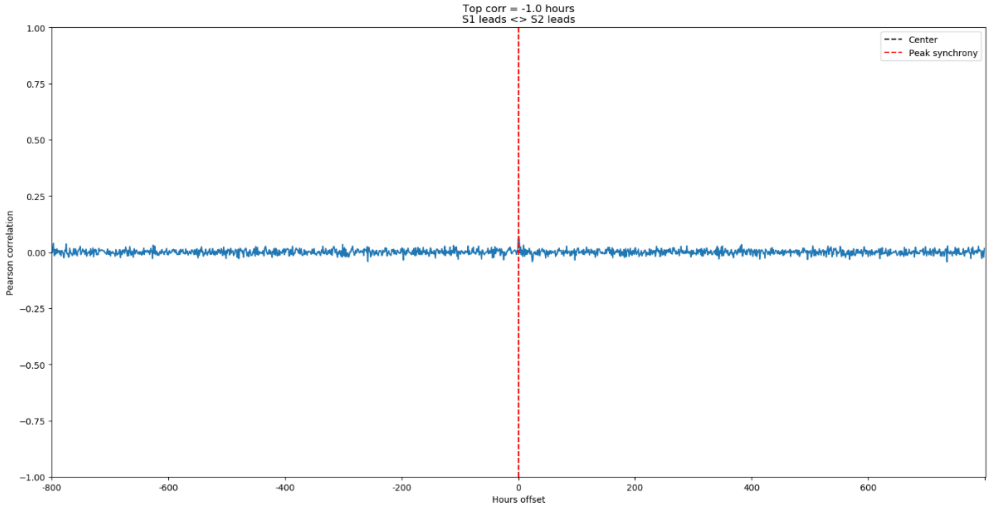


It is dramatically lower, in those numbers the correlation of coins with themselves has relatively high effect on the mean, without them:

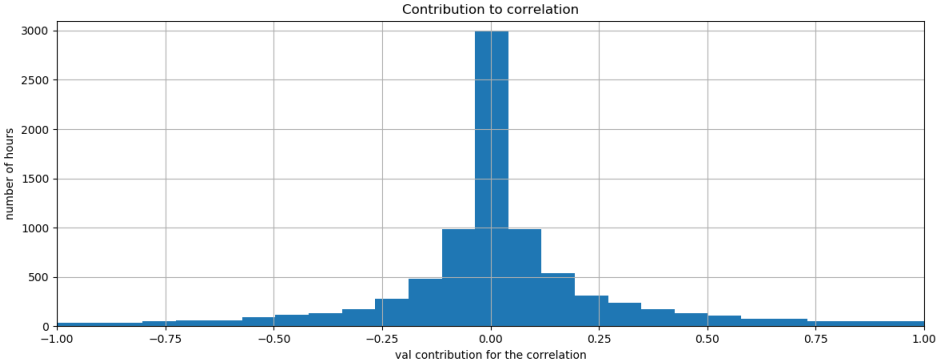
median: 0.2, mean: 0.213

The only pair with shift that is not 0 for the highest correlation has low correlation and shift of only 1 hour:

bitcoin, dash || 0.07 || 1:



While this might look promising, it might be the result of extreme non common values. Let's look if the histogram of Xi is around 0.07 or more than 0:
Xi = (bitcoin(i)-E(bitcoin))*(dash(i)- E(dash))/(std(bitcoin)*std(dash))
Which uphold mean(Xi)=0.07.



Since we have a value of 0 in the middle, and a third of the hours are right there, we deduced that the correlation is the result of high and rare values and not of real correlation, thus, there is no use of this result.

Experiment conclusion:
There is no use of correlation between two different coins in purpose to predict the change of one coin using the other previous results.

Second experiment:

Introduction: The experiment itself is basically choosing a parameter k for the k best algorithm, and using this k to choose k features.

Justification for the experiment:

We choose to conduct this experiment because of the last feature selection method: greedy hill climbing.

Greedy hill climbing is a heuristic for finding the best set of features (the problem itself is in NP), and as such a smaller set of features to begin with is very beneficial.

The experiment:

For each k between 2 and 11, we did the following procedure:

For every scoring meter (of k best algorithm):

Choose k best features by the score of this meter.

Using those features - calculate the score given by a *court of classifiers*.

Choose the k that yields the best score, and with it the k best features (using the meter score that gave the best score).

The *court of classifiers* is composed of untuned RandomForest, Knn and naive bayes.

We chose a court because we were not sure what type of classifier to use at that moment, so one judge from every type of classifier seems like a good balanced choice. Moreover, the more the judges, the better their judgement.

Hypothesis for the expected results:

Our hypothesis is that the score will increase as k increases (additional information is added), up to a specific point, and then will start to decrease.

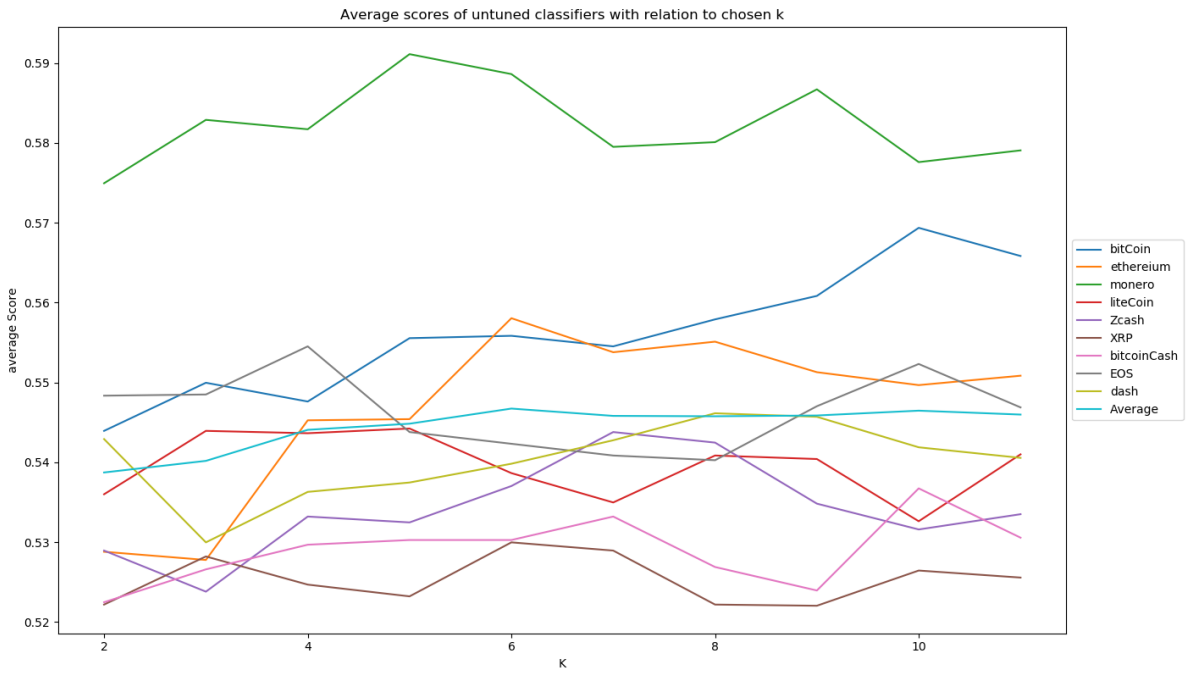
Increasing to a certain point: due to additional information at the new features.

Decreasing after a certain point: famous curse of dimensionality.

Also, we guess that the pick of the score will be the same for all of the coins, as their behaviour is pretty similar to each other.

Results:

The results of the experiment are in the following graph:



As we can see, **on average** across all coins, the score indeed increased up to a certain point, but then it remains at a plateau instead of decreasing.

When looking at all of the coins, we can see that indeed there was a raise in score at the beginning, but then for each coin the score was pretty unstable, meaning that the plateau we got on average is just a smoothing effect of the noise from all of the coins.

conclusions:

- ~6 features are enough to describe the coin state.
- Out of ~60 features, a minimum of 4 features are required to intelligently classify change rate, but at around 6-7 features, adding additional ones is not adding any value (nor hurt, at least up to 12).

Third experiment:

Introduction: This experiment is testing the effect of different tuning parameters on the performance of different classifiers in regards to our problem.

Justification for the experiment:

The main objective of the project is to optimize the profit of the final classifier, as the features are statistical features it is hard to predict which classifier type will fit.

So we did an exhaustive search over classifiers and their parameters, and hope to get some insights out of the results.

The experiment:

For every coin, repeat the following procedure:

> For every classifier from the set: RandomForeset, DecitionTree, Knn and NaiveBayes:
>
> > For every combination of parameters of the classifier:
> >
> > > Train a classifier using those parameters, and then test its score using the train-validation set.

FInally check using the validation 2 set, which tuned classifier yields the best results, and choose it.

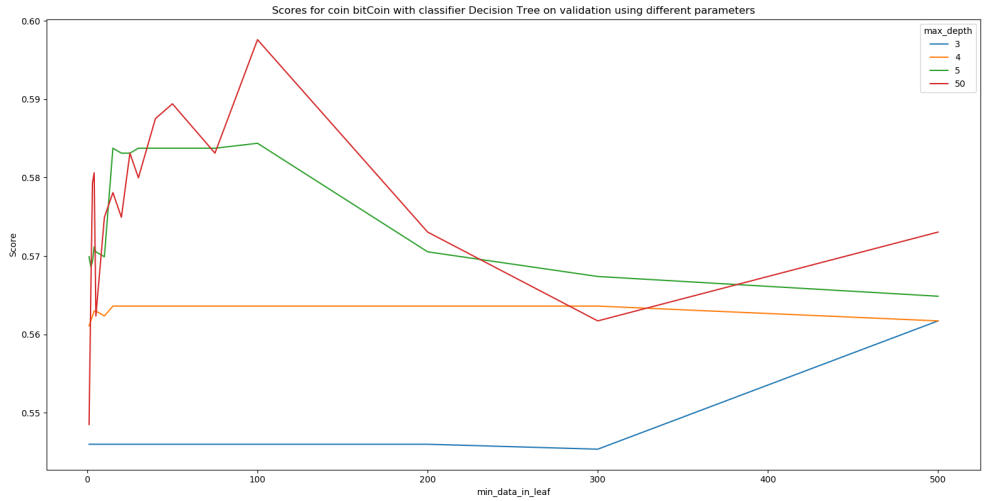
This way, we avoid choosing the classifier and parameters which have the highest overfit to the validation set, since the parameters tuning is not based on the validation set.

Results:

The following graphs exist for every coin, we added only bitcoin graphs as they represent pretty well the other coins:

Classifier: Decision Tree,

Parameters: max-depth and min-values per leaf.



Decision tree is interesting:

Notice that max_depth is also setting min_data_in_leaf (since its set max for the number of the leafs).

As we can see, when there is no limit to the depth of the tree, and the minimum amount of data in a leaf is small, it becomes very unstable. This is a strong indication of overfit: when the limit for the minimal amount of data in a node is low, the tree can fit itself to the data. In this case any small change in the limitation might make it recalculate the whole mechanism.

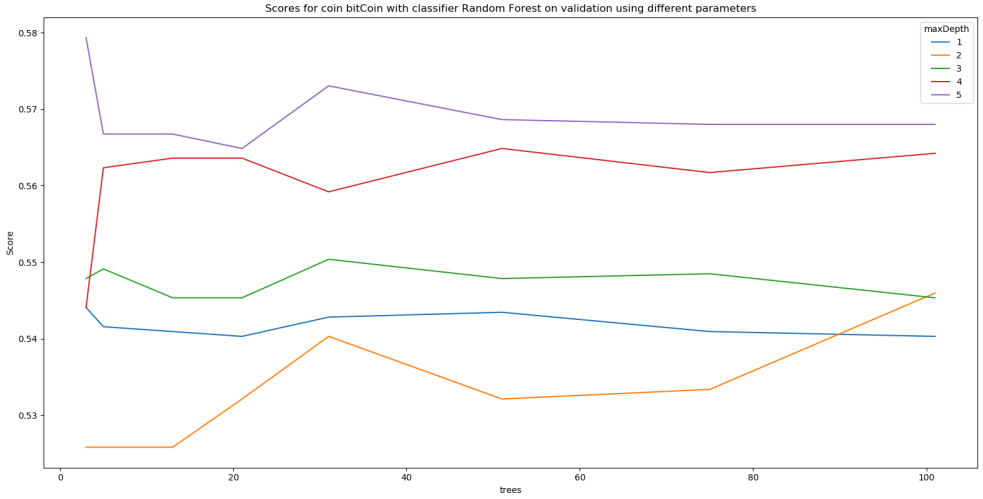
In trees with small limitations on the depth - the minimal size of a leaf is already big, thus the leaf size limitation is not relevant at small numbers. For example, for depth 3 - the minimal size of a leaf has effect only from 300 and above.

For the other depths: we can see that the deeper the tree is able to be - the better it's results, but we should look for depth between 5 and 50, since there might be maximum there.

Also, when the minimal data in a leaf exceeds 100, the 5 and 50 depth tree start to decrease in accuracy, where the 4 depth tree does not, this indicates that the 5 (and 50) depth tree using small leafs. While 4 depth might use only sizable leafs.

Classifier: Random Forest.

Parameters: max-depth and amount of trees



Random forest enhances the argument that 5 or more depth is needed, it is more stable than the regular tree due to its randomness.

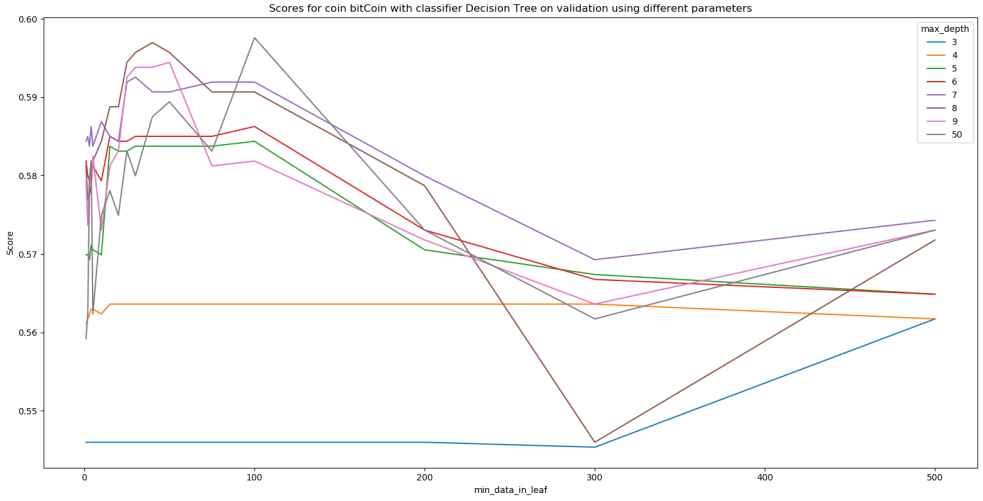
Small number of trees get unstable results, but even 20 trees are enough to stabilize the results.

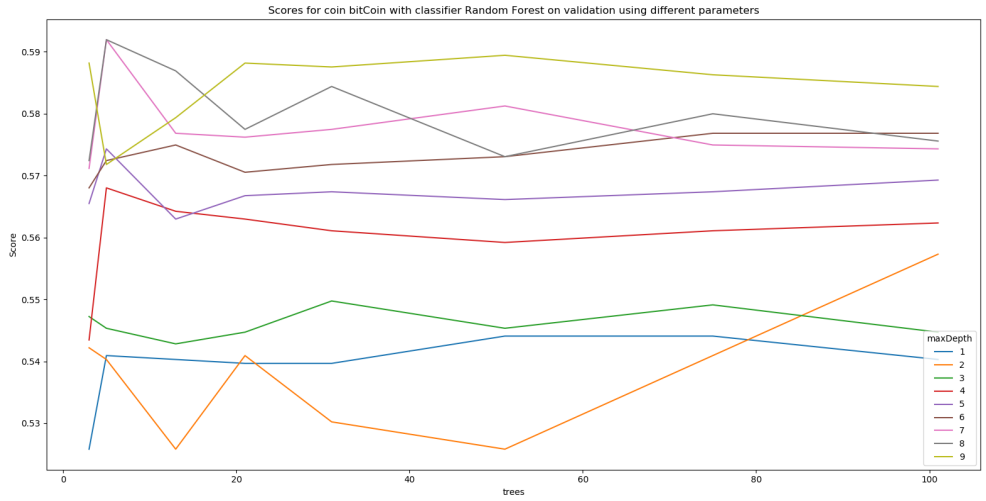
Classifier: Random Forest

Parameters: max_depth, min_data_in_leaf

From the results of the decision tree, it is likely to assume that a bigger (than 5) limit on the tree's depth could be beneficial, as it seems that the accuracy is kept raising in a marginal amount when the limit had been increased, so probably 5 as an upper (real) limit is too little.

Those are the new graphs with the bigger limitation:





findings:

decision trees:

1. Again high max-depth usually get better results, but 7 is the best.
2. High min_data_in_leaf gets stable and pretty good results, even though low min gets better results, it is probably the result of an overfit.

random forest:

1. Again high max-depth gets better results, but here in our numbers - the more the better - 9 is the best.
2. As expected, more trees make it stable.

explanations:

While most of the results were expected, the difference of max-depth need to be explained with the highest accuracy of the random forest:

We choose the features for the decision trees in a greedy way.

In that way, depth of 7 makes the leaf pretty close in classification - another partition of the leafs to different classifications will make it worse, That's why 7 get the best results in decision trees.

While this is true, there is a better way to describe the data in a leaf, but for that - we need to choose at the (relatively) beginning features that will not get the best partition according to immediate classification success, meaning features that the greedy algorithm will not choose.

Those features are chosen by some of the trees of the random forest (since his trees get only part of the feature to choose from), and that explains two things:

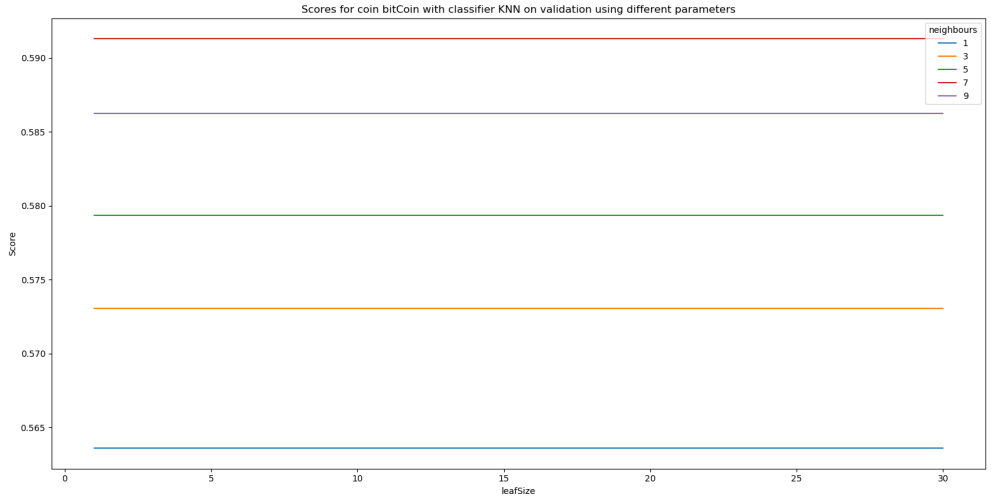
1) Why random forest gets top results in highest depth than 7 (7 in decision tree)

2) Why the results of the random forest are better (which might also be the result of the random forest's committee).

We decided not to raise the max depth limit even more, as we are a bit afraid of overfit.

Classifier: Knn

Parameters: k, leaf size



Knn: leaf size is some parameter for reducing computation time at the cost of accuracy, as we can see in this range it has no effect. Actually, it seems like leaf size is not a parameter.

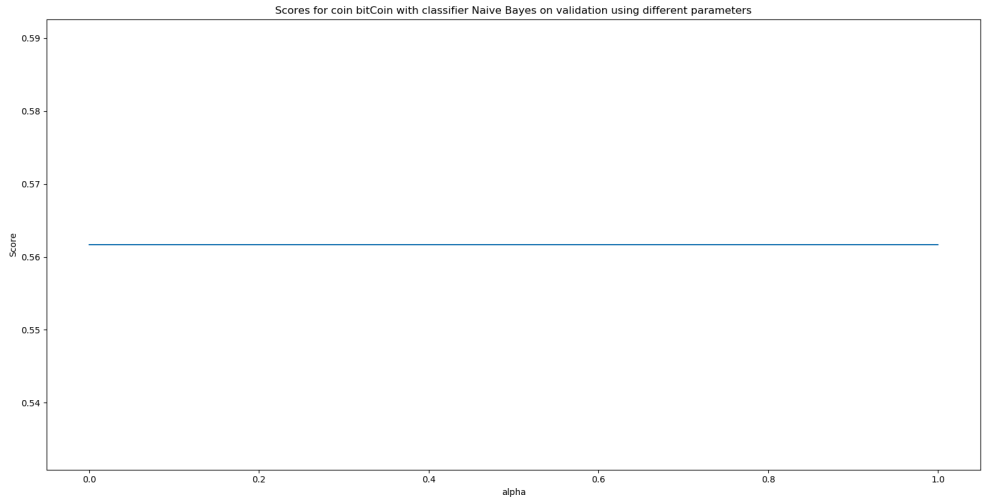
We can see that different k's produce significantly different results - first the accuracy is raised and then fall, when k=7 is the best.

This is not surprising, since the features we use can't totally predict the right classification. Looking at a low number of neighbors in Knn gives too much importance to the classification of any neighbor, while it's classification might be "wrong", i.e. according to the features we use it should have the opposite classification.

So highest k reduces the importance of any neighbor's classification and produces higher accuracy. Yet, the higher the k - the farthest the neighbors we look at, so this reduces the accuracy. This gave us the best accuracy at k=7.

Classifier: Naive Bayes

Parameter: alpha



Naive Bayes is doing an ok job at prediction, but it does not respond to alpha parameter, this does not give us any interesting information.

Fourth experiment:

Introduction: this experiment is testing the effect of different certainty limitations on the results of our prediction algorithm.

Justification for the experiment:

After we got the best classifier we could, this is the time to do the best with what we have got.

In real life there are two major considerations that we did not mentioned yet:

First, there are for every buying and selling of a coin some exchange fee and difference between buying and selling price (for the same coin at the same time), if the prediction is not profitable enough when buying and selling every hour, those fees cost can be higher than the profit, and make the whole process unprofitable.

Second issue is risk management, there could be a scenario in which we will want to have a better success rate in expense of profit.

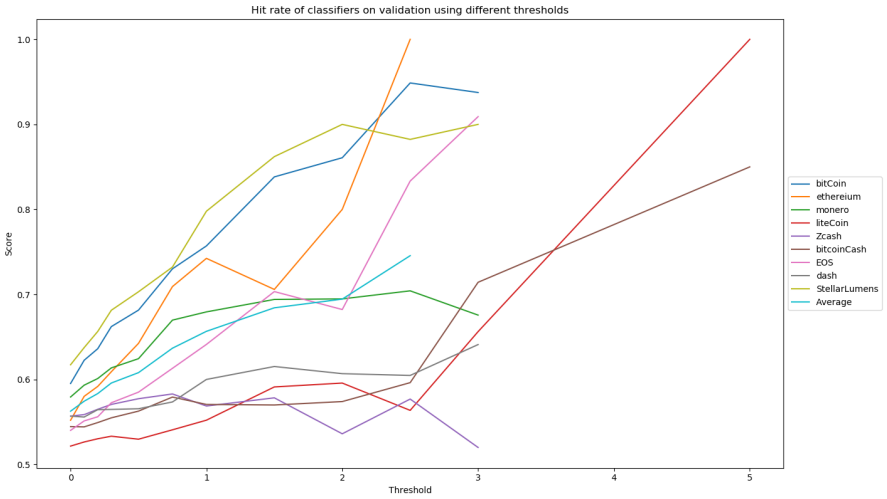
The experiment:

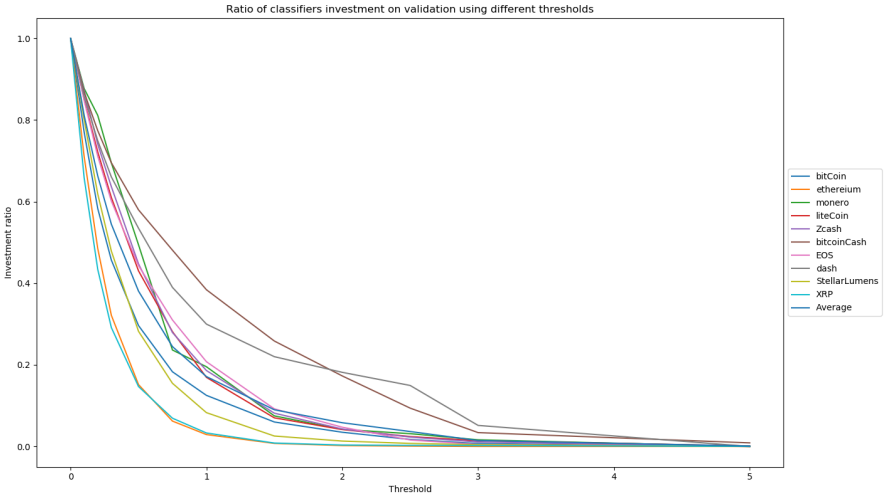
To address those issues, we introduce a variation to the classifier: invest only if you predict that the change will be bigger than a certain threshold. This might produce higher revenue per investment.

This change addresses both issues: first issue, because higher revenue might overcome the exchange fees. Second issue, because higher revenue makes the ratio of risk to profit better.

The experiment is the effect of a threshold on the profitability and success of the algorithm.

Results:

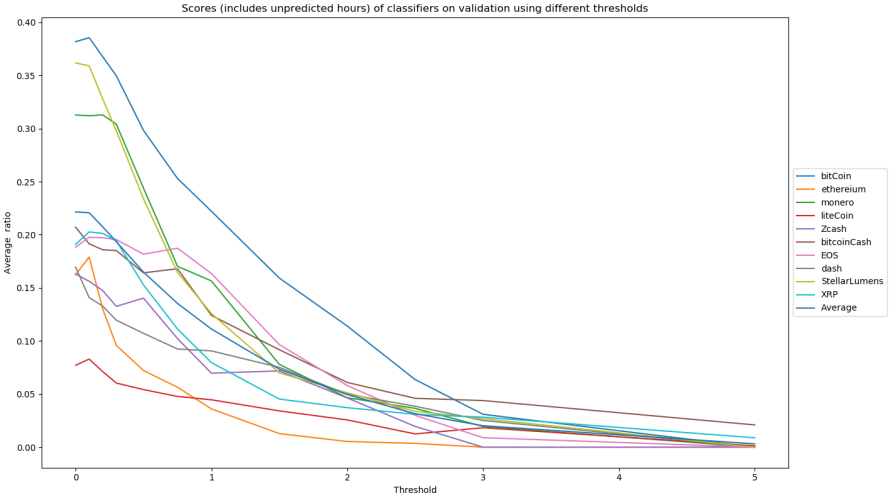




The results support the hypothesis: on average, increasing the threshold results in a significant increase in accuracy.

Also, high thresholds are more commonly predicted by the classifiers than we suspected: on *average*, trading hours in which the change is predicted to be more than 1% are about 20% of the times.

An interesting graph is the following one:

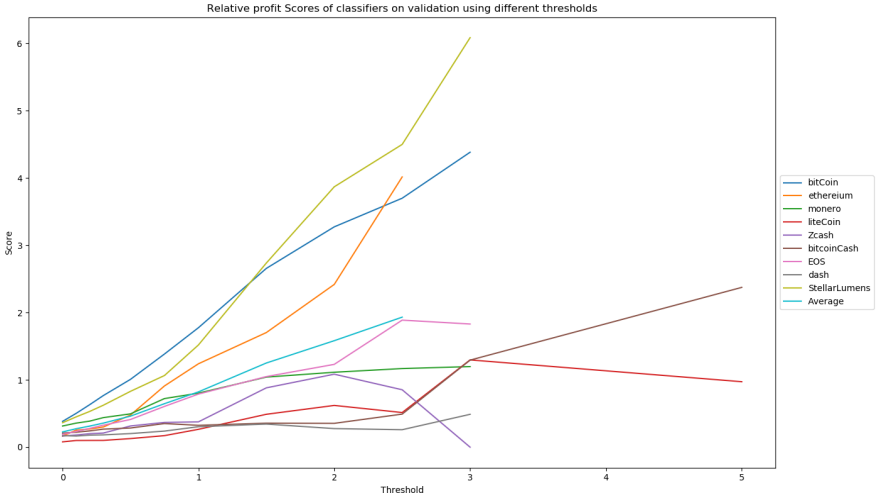


This graph shows the ratio between the profit of an investment with our algorithm with the threshold, to investment with a perfect algorithm (the one who knows the future).

We can see that the overall profit is, in several coins, a bit better when limiting the algorithm with a low threshold, than when not limiting it at all, and only then start to fall. This means that when the algorithm predicted very low change, he got almost 50% (a bit more) success rate on average.

This being said, on average it is indeed less profitable (ignoring fees) to put a threshold limit on the algorithm, and the profit is monotonously decreasing the bigger the limit threshold gets.

For investing purposes, we can see that, as suspected, higher threshold results in higher accuracy rate (the first graph in this experiment), and higher average profit per investment:



Final Results:

As a result of experiment number 4, we compare here the results of 4 different classifiers:
The results are averaged across all coins.

The results over the test set:

threshold:	2%	1%	0%	Optimized
hit rate when invested	60.2%	55.8%	53.6%	53.7%
% of the time invested	15.8%	33.3%	100%	8.1%
% of the time invested and hit	9.2%	18.7%	53.6%	4.6%
% of the time invested and missed	6.5%	14.6%	46.4%	3.5%
average profit per investment compare the average change (average change in all the hours)	0.747	0.323	0.122	1.091

The results over the validation set:

threshold:	2%	1%	0%	Optimized
hit rate when invested	74.9%	65.5%	55.8%	86.8%
% of the time invested	3.1%	12.0%	100%	3.4%
% of the time invested and hit	2.1%	7.7%	55.8%	2.0%
% of the time invested and missed	1.0%	4.3%	44.2%	1.4%

average profit per investment compare the average change (average change in all the hours)	2.185	0.905	0.211	4.417

The verification set gets higher results than the test, this is probably the result of overfit of the chosen algorithm to the verification set. Conclusion: use more than 1 or 2 verification sets.
Looking on the test, we can see that the threshold of 2% gets the best results; until 2% - the higher the threshold the better the results.
The results are going down in the optimized threshold, possible explanations:

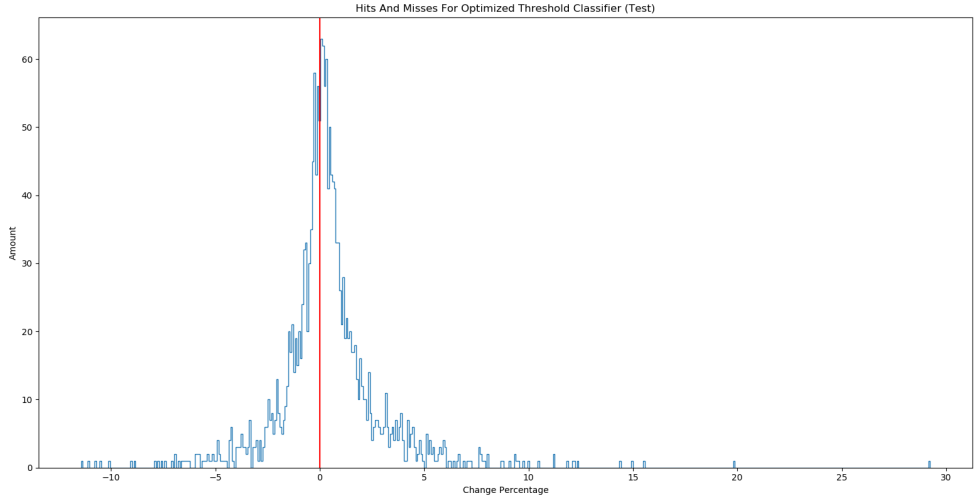
- Since this threshold is rare, at a high percentage of the time it's happening as a result of overfit - that is the main reason for predicting such a high change.
- The features values that are responsible for such a high percentage change, are correlated to hard prediction values, which means it is more possible to be wrong in these cases.

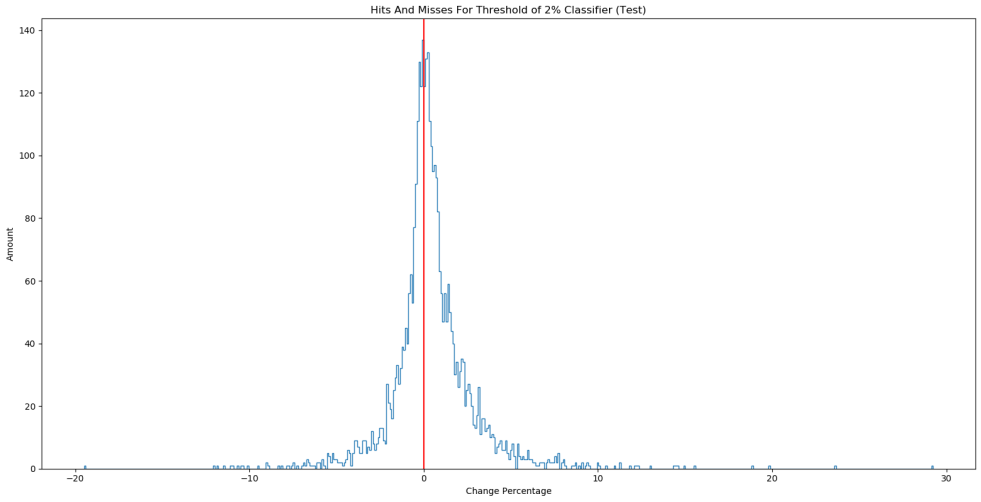
Yet, the "average profit per investment compared to the average change" is the greatest in the optimized: while it is highly over fit, it still tends to catch the really good investment.
At other thresholds, the better one in binary predictions, the better its profit per investment.
It is worth mentioning, is that 2% get 0.747 in "average profit per investment compared to the average change", and choose to invest at 15.8% of the time, which is almost 4 hours per day (24 hours). So it might be useful for daily trading.
On the test, trading with a range of [2-optimized]% will yield the higher hit rate but not the best average profit.

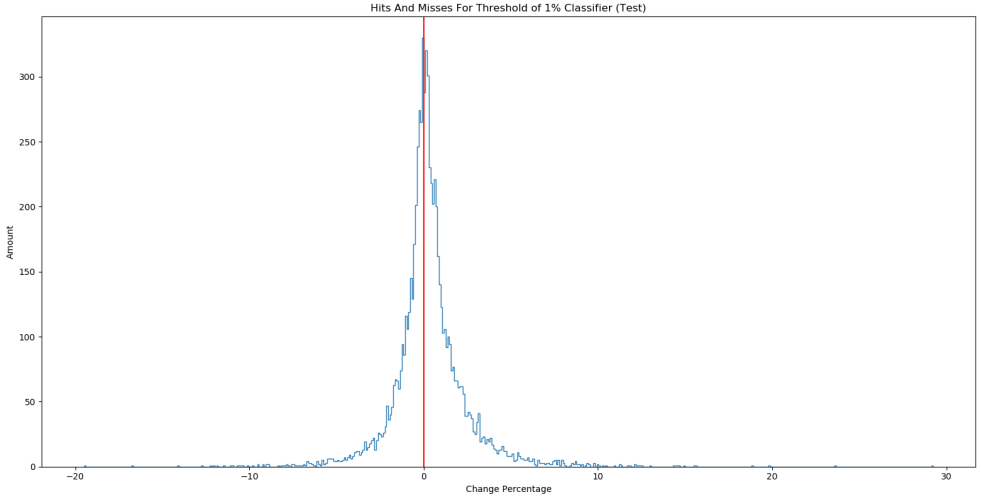
Looking on revenue histograms will yield the same results and conclusions.
The optimized has almost half of the investment in the left side.
2% threshold is higher on the right side.
1% threshold looks similar to the 2%, but higher in the middle for the obvious reason, which makes the right edge look relatively lower.

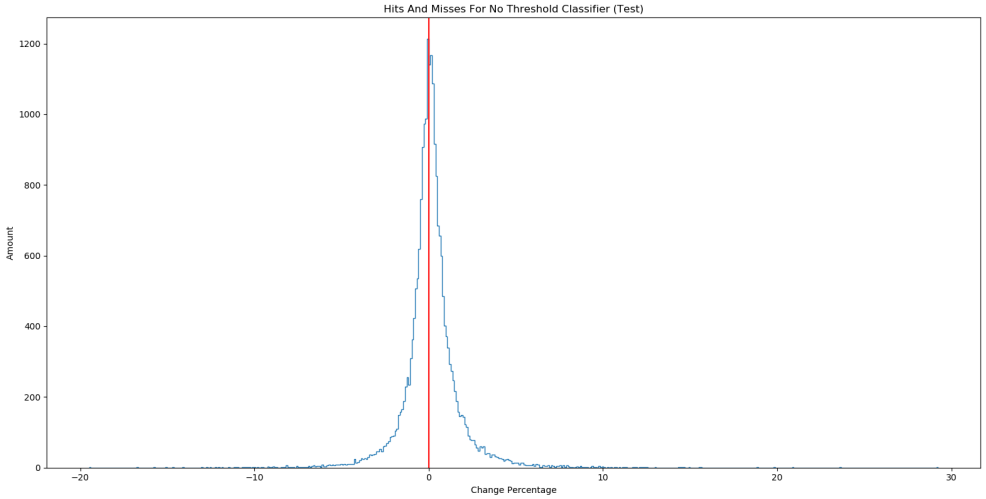
No threshold starts significantly high and falls smoothly until the edge.

Notice that all the histograms looks similar since every graph contained in the one under it:


Hits And Misses For Optimized Threshold Classifier (Test)
Amount
Change Percentage


Hits And Misses For Threshold of 2% Classifier (Test)
Amount
Change Percentage


Hits And Misses For Threshold of 1% Classifier (Test)
Amount
Change Percentage



Another important point: the market efficiency is a bit lower for google trends, as we saw that those features are usually selected. This might imply that other uses of features different from technical indicators might yield good results.

Future research:

In this project we successfully implement a learning algorithm which is able to predict cryptocurrencies exchange rate with a success rate that is better than simple guess, and in some cases might even yield profit.
Future research following this project might be:

- Use more validation to avoid overfit, as it's hard in such a market, and find the best range to trade. This range shall optimize the trader purpose, such as hit rate, average profit, commission, #investments/day etc.
- Use more classifiers, including deep learning - LSTM and more.
- Store twitter mentions for ~1 year and use it for the classifying.
- Use features based on mentions in public news in addition to Google and Twitter.
- Instead of aggregating all of the different markets data into a single dataset, select one market to conduct the investing at, and use the aggregation of the rest of the markets as indicator of the current arbitrage between the market and the "real" price.
- Fetch the fees of buying a coin and the difference between buying and selling prices for every coin, and use this information when optimizing the threshold for investing.
- Try more investment strategies, for example - Pairs trade.
- Focus on classifiers with success only when the prospect profit is 2% or more, and classifiers that can detect where the first might be wrong, or maybe a limit of less than 2%, but only when all the classifiers agree.

Appendix - program description:

Introduction for the system:

The program flow is basically one big pipeline, the first step has no input (it generates its own data from the internet), and afterwards every part is using previous part output as input.
All of the communication between the different parts is done using files.
Those files are generated throughout the program inside a configurable folder, **which must be specified in order for the program to work.**
The address of this folder is expected to be found in the “env.py” file in the main folder.

Once the program has been run once, every part can be run again by simply running the corresponding file (python <file>).

Important notice! - Running a file should be done only from the main folder with its relative path, otherwise it will not work.

System description:

The system pipeline - beginning at fetching the raw data from the web, up to printing the results of the final test.

Here is this exact pipeline including references between the code and the sections of the report, the report itself is mainly builded in the same order as the pipeline.

The (<path>) near a logical section is the path needed to be run in order to execute this part (python <path>).
Input to a logical part is the output of the previous part, the program handles it in such a way that when a part is finished, its output is implicitly ready for the next part.

The data flow of the system is as follows:

1) Downloading data from the web: (downloadData/main.py)
    a) Download datasets of coins. (downloadData/dataBases.py)
    b) Download google trends data. (downloadData/googleTrends.py)
2) Generating data for learning algorithms out of the downloaded data: (genData/main.py)
    a) For coins value data sets: (genData/aggregateDataSets.py - part a and b)
        i) For each data set:
            (1) Make the data to be in our defined common conventions.
            (2) Padding the datasets to have the same time structure.
            (3) Cleans data.
            (4) Trim the data to common defined times.
            (5) Transform currency units to USD (only in step b)
        ii) Generate weights for each dataset.
        iii) Aggregate all weighted datasets to a single set.
        iv) Fill missing / dirty data.

b) Repeat a) for coins with original price exchange rate in Bitcoin\Ethereum while exchanging their rate to USD.
c) For Google trends datasets:
    i) Convert to common convention.
d) Feature generation: (genData/genFeatures.py -include part c)
    i) Merging google trends data (and creating one new feature out of it).
    ii) Creating a lot of statistical features (technical indicators) using ta library.
e) Splitting to train, validation and test. (genData/splitDataSets.py)
f) Normalizing features: (genData/normalizeFeatures.py)
    i) For each feature, deciding if its distribution is more similar to uniform or to binumic, and normalizing accordingly.
g) Selecting features: (genData/selectFeatures.py)
    i) Cleansing gurbish features.
    ii) Using Pearson correlation to remove redundant features.
    iii) Selecting best k features.
    iv) Greedy hill climbing for choosing best features.

3) Choosing classifier: (classifying/main.py)
    a) Generating classifier (classifying/classifiers.py):
        i) For every type: tune parameters.
        ii) Choose the best among the tuned classifiers.
    b) Optimizing certainty threshold for investment (classifying/thresholdForInvestment.py)
    c) Test the results (classifying/finalClassifying.py)