Quantitative trading platform

(Platforma do handlu kwantytatywnego)

Artur Skarżyński Maciej Okupnik Szymon Wieczorek

Praca inżynierska

Promotor: dr Marek Adamczyk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

5 września 2021

Abstract

This paper describes the process of implementing a cryptocurrency trading platform. In the first chapters, we describe the process of downloading and processing data as well as training neural networks that predict price changes of selected cryptocurrencies. Based on the generated predictions, we created trading strategies and tested them on historical data. The second part of the paper focuses on the description of a platform that downloads and processes price data live, generates predictions and executes trading decisions made by the strategy we have chosen.

Niniejsza praca opisuje proces implementacji platformy służącej do handlu kryptowalutami. W pierwszych rozdziałach opisujemy proces pobierania i przetwarzania danych oraz trenowania sieci neuronowych przewidujących zmiany cenowe wybranych kryptowalut. Na podstawie generowanych predykcji, stworzyliśmy strategie tradingowe i przetestowaliśmy je na danych historycznych. Druga część pracy skupia się na opisie platformy, która na żywo pobiera i przetwarza dane cenowe, generuje predykcje oraz realizuje decyzje tradingowe podjęte przez wybraną przez nas strategię.

Contents

1 Introduction **7**

1.1 Description and our motivation 7

1.2 Scope of work 8

2 Fetching the data **9**

2.1 Binance REST API 9

2.2 Binance Websocket 9

3 Models **11**

3.1 Convolutional neural networks 11

3.2 Training methods 12

3.2.1 Stochastic Gradient Descent 12

3.2.2 Loss function 12

3.2.3 Data loading 13

3.2.4 Early stopping 13

3.3 One-dimensional convolutional neural network 13

3.3.1 Data 14

3.3.2 Results 14

3.4 Two-dimensional convolutional neural network 16

3.4.1 Data 16

3.4.2 Results 17

4 Trading bot **21**

4.1 Strategy 21

4.2 Results 21

5 Storing the data using Influx **23**

5.1 Introduction to Influx 23

5.2 Usage of the database 24

6 Workflow management using Airflow **25**

6.1 Introduction to Airflow 25

6.2 Workflow for 1-minute time intervals 27

6.3 Workflow for 5-minutes time intervals 28

6.3.1 Calculating indicators 29

6.3.2 Generating model's prediction 29

6.3.3 Trading bot 29

7 Live trading results **31**

Bibliography **33**

Chapter 1

Introduction

1.1 Description and our motivation

Quantitative trading consists of strategies that generate buy and sell signals based on mathematical, statistical and machine learning tools and methods. The algorithms for generating the signals are developed and tested based on the historical data of the market, e.g. prices, total number of shares traded in a given period of time, even news events and Twitter activity. Machine learning methods can be trained on this data to look for the patterns that would prove to be good predictors of the incoming price change of a commodity we want to trade.

The constant growth of computing power for the last few decades, rapid development of deep learning methods and an increasing access to high-speed Internet has caused this type of trading to become more and more popular throughout recent years. According to Ernest Chang [1], in 2008 about one third of the trading volume in United States has been generated by quantitative trading.

We wanted to see how data analysis and neural networks can be used to generate trading signals for cryptocurrencies trading. We have chosen this market since it is fairly new, has seen a sudden rise of popularity in the last few years and what's more important, the data for it is easily available and thanks to multiple platforms such as Binance, trading is a simple process performed via API. Having the access to the data and models that predict price changes, we wanted to develop trading strategies and backtest them on historical data. Finally, we would implement one of them for live trading by creating a platform that fetches the live data, processes it and generates the predictions which can be used as trading signals by the bot.

1.2 Scope of work

Work performed by Maciej Okupnik:

1. Fetching the data using Websocket
2. Creating an interface for Binance API for live trading
3. Processing of the data for 2-dimensional neural networks

Work performed by Artur Skarżyński:

1. Configuration of InfluxDB and Airflow
2. Implementation of the strategies for backtesting
3. Implementation of the final bot for live trading

Work performed by Szymon Wieczorek:

1. Architectures of the neural networks
2. Implementation of data loaders and training loop for the networks
3. Workflow for the BTC 1-minute interval klines

Work completed together:

1. Fetching the data for the training of neural networks
2. Workflow for the BTC 5-minute interval klines

Chapter 2

Fetching the data

We have looked over many platforms for cryptocurrencies trading. We decided to go with Binance - because of our familiarity with it as well as extensive API and existing Python package. [2]

Building over Python-Binance, we implemented our own client - both for getting historical data and for receiving live data via websocket. To follow good practices and produce the cleanest code possible, we implemented *Factory design pattern* that helped us package the data into objects (henceforth called *Klines* or *Candlesticks*).

2.1 Binance REST API

After we initialize our client with Binance API credentials, we can start fetching the data.

To do that, we have used function provided by the python-binance package. We pass the starting date, end date and chosen time interval, then iterate through generated *Klines* objects. Using the Pandas package, we have saved the data into a csv file.

2.2 Binance Websocket

We have implemented *Websocket Handlers* class for dealing with live data. We had been listening to the Binance websocket [4] and initializing *Klines* objects everytime we encountered some new data.

Binance API delivers new objects every few seconds. Because of that, to download 1-minute and 5-minute interval klines, we have to check the open and end time of incoming klines, and only when we receive the one that actually lasted the whole interval, we can save it.

Chapter 3

Models

During the development of the models and methods for data processing, we followed some of the approaches described by Stefan Jansen [3]

3.1 Convolutional neural networks

To predict the price change of a cryptocurrency, we focused on developing and testing multiple convolutional neural networks (CNN). This type of deep learning model excels at computer vision tasks such as classifying images, recognizing objects in videos or generating pictures. However, it can also deliver high-quality results for time-series data.

CNN is based on convolutional layers. Input to each of the layers is an array of shape:

[samples] x *[input height]* x *[input width]* x *[input channels]*

Output has a shape of:

[samples] x *[feature map height]* x *[feature map width]* x *[number of filters]*

Each convolutional layer contain a number of filters (kernels) which are matrices with learnable parameters. To process an input, the filter slides along an image and calculates dot product between the current patch of the image and matrix parameters. Dot product generates a single number and the entire scan of the image generates a feature map. Thus, in contrast to dense feed-forward layer, CNN layer uses local connectivity - inputs that go into a single neuron are all close to each other (they form a fragment of the input image). Because of that, we have to make a key assumption that there exists a meaningful local structure in the data we are dealing with.

3.2 Training methods

3.2.1 Stochastic Gradient Descent

To train our models, we have used Stochastic Gradient Descent. This method iterates over shuffled small subsets (called "mini-bachtes") of the training set. During every iteration, it updates the neural net's parameters based on the gradient evaluated on a mini-batch. A full pass over the training data is called an epoch.

We focused on two crucial hyperparameters: learning rate and momentum factor. First one dictates how big should be the changes applied to the parameters during every iteration. The latter controls the magnitude of a momentum method. Momentum stores the information about previous gradients and calculates the next update of parameters as a linear combination of the new gradient and the previous update:

$$V_n = \beta V_{n-1} - \alpha \frac{\partial LossFun}{\partial \theta_{n-1}}$$
$$\theta_n = \theta_{n-1} + V_n$$
$$where: \ \theta - parameters, \ V - step\ size, \ \beta - momentum\ factor, \ \alpha - learning\ rate$$

We used SGD implementation provided by the torch.optim package [5].

3.2.2 Loss function

As a loss function, we have chosen to use mean squared error (MSE) loss. We are trying to solve a regression task - predict the percentage by which the price will change in the next time interval - and MSE is one of the most popular loss functions to use in such problems. It calculates how close our predictions ($f(x)$) are to the ground truth values: (y):

$$MSE = \frac{1}{n} \sum_i^n (f(x_i) - y_i)^2$$

MSE does a good job as a loss function that can be optimized during the training, but is hard to analyze and is not very informative about the model performance. Because of that, we used two other metrics to measure the effectiveness of our neural networks:

1. Information Coefficient (IC)
   IC is a measure used to evaluate how good an investment analysis is. It should show how close the financial forecast is to the ground truth. We obtained IC by calculating Spearman rank correlation (using SciPy library [9]) between the model's price change predictions and the actual price changes. Spearman correlation ranges from -1 to 1, with 0 implying no correlation and 1 meaning a perfect correlation.

2. Accuracy
   We are solving a regression task, but we can also think of our model as a binary classifier. If the goal was to guess whether the price will rise or fall, our neural network that predicts a price change can try to solve it - any prediction with a positive sign is a forecast of a price increase (True) and a one with a negative sign is a forecast of a drop in price (False). For example, let's say our model predicted that the Bicoin price at 14:10 will change by **0.05%** compared to 14:05, at 14:15 by **-0.2%** compared to 14:10, at 14:20 by **-0.09%** compared to 14:15. If the actual price changes were **-0.1%** at 14:10, **-0.1%** at 14:15 and **0.006%** at 14:20, we would say that accuracy of our model is 33.3%.

3.2.3 Data loading

To enable storing, shuffling and iterating over the mini-batches during the training, we created a DataLoader class that builds upon a PyTorch utils.data.DataLoader [7]. Our DataLoader class transforms the features and targets into torch.tensors and stores them in PyTorch TensorDataset[6] which can be passed and used by Pytorch DataLoader. Since we are working with time-series data, we also had to store the dates of every sample and have a quick and easy access to the whole, not shuffled dataset for later analysis of the results.

3.2.4 Early stopping

The training loop we implemented is using early stopping. Every epoch, we evaluate our neural network performance by calculating Information Coefficient and MSE on the validation set. In our training, this set usually consists of about 10%-20% of the whole dataset. We then compare the IC with the best IC our model has achieved so far during the training. If the model doesn't improve on this metric for more than 20 - 30 (depending on the model) epochs, we stop the training.

3.3 One-dimensional convolutional neural network

The main purpose of this model was to do a sanity check of the idea of predicting price changes using convolutions. The architecture of this neural network is very lightweight which enables quick testing of different training and data processing methods. It uses a single one-dimensional convolutional layer with ReLU activation function, followed by MaxPool layer and a fully connected layer.

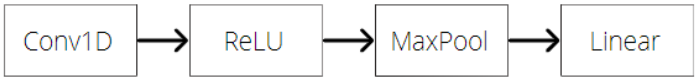


Figure 3.1: Architecture of the 1D Convolutional Neural Network

3.3.1 Data

We have used 1-minute time interval Bitcoin klines to calculate *lagged returns*. A single lagged return is a change between the current price and the price from the last interval. To prepare the data, we had to calculate changes of the close prices so that a single sample consisted of 1-minute returns from the last twenty minutes, and a label which is the lagged return for the next minute.

	label	1	2	3	4	5	6	7	8	9	10	11	12	13
date														
2020-06-01 01:22:00	0.002110	0.050765	0.004644	0.003589	-0.018573	-0.010868	-0.007491	-0.034701	0.008439	-0.039435	-0.009173	-0.026878	-0.020971	0.017391
2020-06-01 01:23:00	0.043987	0.002110	0.050765	0.004644	0.003589	-0.018573	-0.010868	-0.007491	-0.034701	0.008439	-0.039435	-0.009173	-0.026878	-0.020971
2020-06-01 01:24:00	-0.009595	0.043987	0.002110	0.050765	0.004644	0.003589	-0.018573	-0.010868	-0.007491	-0.034701	0.008439	-0.039435	-0.009173	-0.026878
2020-06-01 01:25:00	0.060528	-0.009595	0.043987	0.002110	0.050765	0.004644	0.003589	-0.018573	-0.010868	-0.007491	-0.034701	0.008439	-0.039435	-0.009173

Figure 3.2: Dataframe with lagged returns

3.3.2 Results

We did a 70% - 10% - 20% split between train, valid and test set. Test set consisted of around 100000 samples from 2021-03-21 to 2021-06-02.

We have experimented with different parameters in SGD. Learning rate varied from 0.0001 to 0.1, momentum factor from 0.8 to 0.99 (sometimes we would turn the momentum off completely). Eventually, the best results were obtained by a network trained with a learning rate of 0.009 and a momentum factor of 0.95.

Information Coefficient:	0.1
Accuracy:	51.59%

Correlation of 0.1 reveal that there is a weak connection between model's output and ground truth. Accuracy higher than 50% means that the network is better than a coin flip. Achieving that reveals that a very lightweight neural network that uses one-dimensional convolutions, can in fact find patterns and draw useful information even from very simple data.

To further investigate on how the model behaves, we plotted the predicted price changes and real price changes. Figures 3.3 and 3.4 suggest two things. First of all, the model is capable of correctly predicting the direction of price changes. Unfortunately, its predictions are significantly scaled down in comparison to the ground truth.

Because of transactions' fees, generating trading signals using this model would be very tricky or even impossible. Fee on Binance is 0.1% (deducted from every transaction). To make a profit with a very simple strategy of buying when the price would rise and selling it afterwards, we would have to know when the price would increase by at least 0.2%. In the test set, there were 4828 price changes higher than this threshold. Unfortunately, our model generated only 43 predictions of that magnitude. This would make it very difficult to extract information leading to a profit.

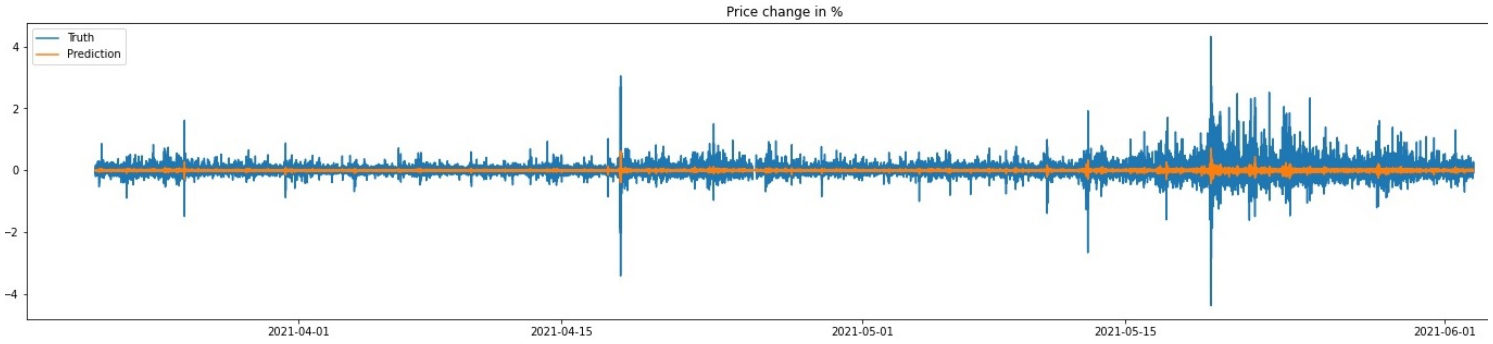


Figure 3.3: Predictions of BTC price changes from 1D Conv Model

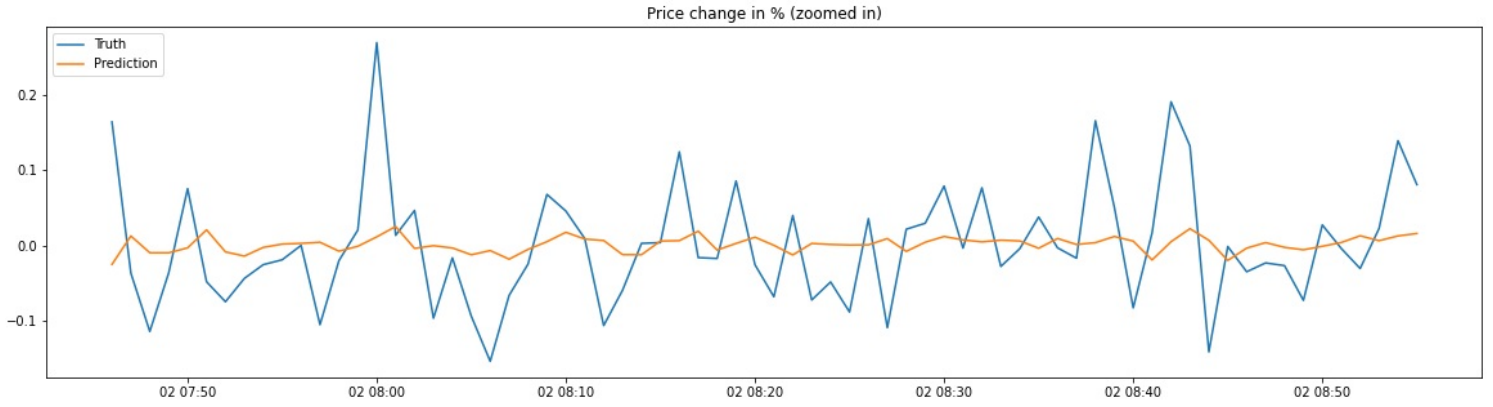


Figure 3.4: Zoomed predictions of BTC price changes from 1D Conv Model

Overall, the results create a promise that a more advanced model that uses more advanced data will be able to generate predictions useful in practical applications if it would overcome the risk of a significant downscaling of the price changes.

3.4 Two-dimensional convolutional neural network

In order to fully exploit the grid-like structure of time-series data, we used two-dimensional convolutions. We processed the data into grids of 15 x 15 shape that may resemble images - something in which convolutional neural networks excel.

Architecture we used consists of two convolutional layers with ReLU activations, followed by MaxPooling and Dropout layers. At the end, there are two fully connected layers with ReLU and Dropout in between.

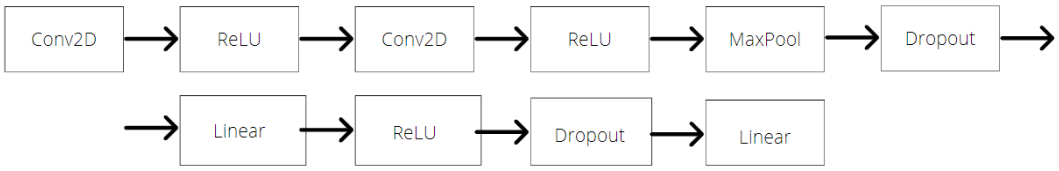


Figure 3.5: Architecture of the 2D Convolutional Neural Network

3.4.1 Data

After collecting around 100 thousand klines of 5-minute time intervals, we wanted to extract features that will enable our model to predict the price changes in the best way possible. Nowadays, successful traders seem to be in touch with every technical indicator, that is why we also consider these as a must-have information.

Creating indicators

Using Python Technical Analysis Library [8] (henceforth called TA-Lib or TA), we were able to compute over 20 technical indicators. For every indicator we varied the time period to obtain 15 distinct measurements (e.g. calculating moving averages for the last 2, 3, .. 16 klines), creating over 300 features.

Composing the grid

Our idea revolves around making a 15x15 grid. That's why we chose 15 different time intervals and that's also the reason why we had to choose top 15 indicators.

We calculated mutual information (MI) between every feature and the target - changes in the price. We obtained the MI for more than 300 features. To get the MI for every indicator, we averaged the results over time intervals for each of them, e.g. MI of the commodity channel index (CCI) is the average of MIs between CCIs calculated from different time intervals and the target. Figure 3.6 shows the best indicators for the Ethereum.

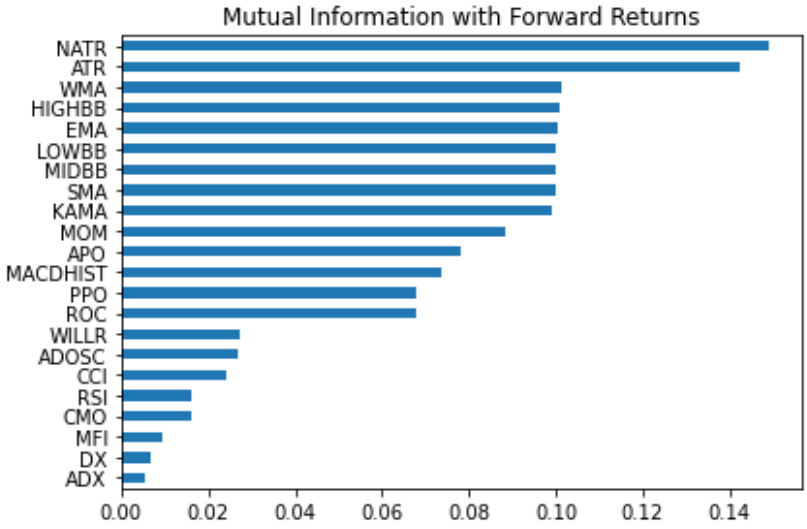


Figure 3.6: Mutual information with Ethereum 5-minute forward returns

Since we're trying to utilize CNN architecture, we had to assert that rows and columns are placed in a way that resembles images, where adjacent pixels tend to have some kind of similarity. That's why we first standardized the features, then computed distances between them and lastly found out the optimal rows and columns order by doing hierarchical clustering. Figure 3.7 shows the optimal ordering.

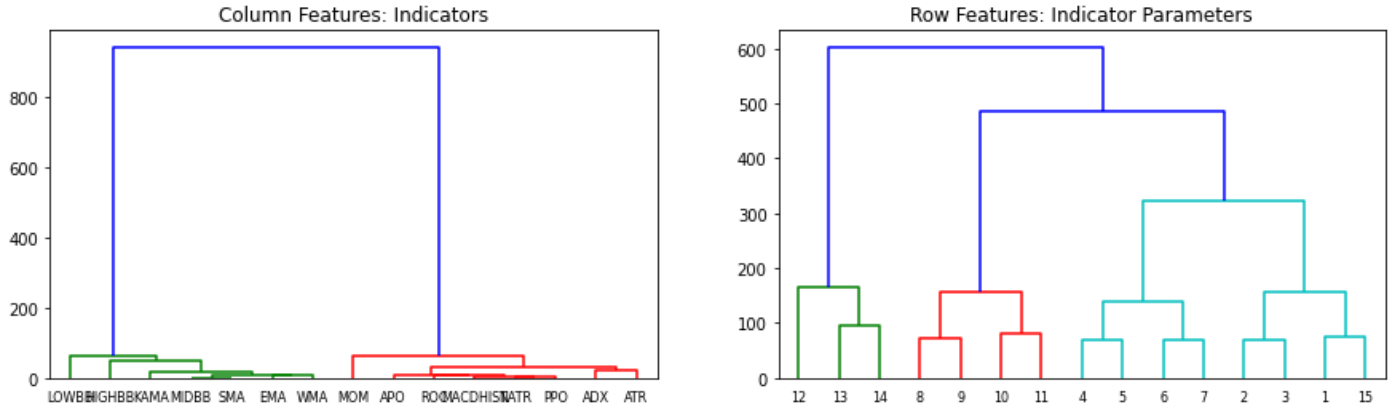


Figure 3.7: Optimal features order

3.4.2 Results

Predicting Bitcoin using Bitcoin

To use as many fresh data for the training as possible, we decided to go with 80% - 10% - 10% split between train, valid and test sets for all of the approaches described below.

First thing we tried was to predict Bitcoin price changes based on grids of indicators generated from Bitcoin data. We experimented with learning rate from 0.01 to 0.3 and momentum factor between 0.8 and 0.97. The best results were achieved when the first was set to 0.02 and the latter to 0.95:

Test set size:	around 10000 (34 days)
Information Coefficient:	0.21
Accuracy:	54.4%

139 predictions generated by this model were higher than 0.2. Since the test set consisted of price changes from 34 days, it gives us on average 4 potential tradable signals per day.

We were happy with the results but decided to test a few other approaches.

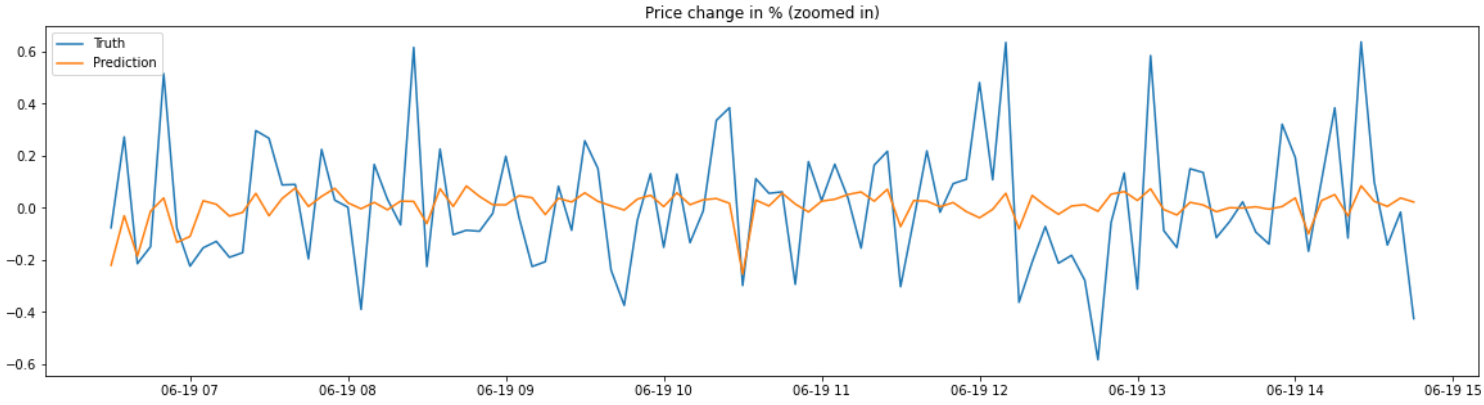


Figure 3.8: Zoomed predictions of Bitcoin by 2D ConvNet using Bitcoin data
We can see that the magnitude of predictions is much higher than in the one-dimensional network.

Predicting Bitcoin using Bitcoin and Ethereum

In the cryptocurrency market, certain currencies are related to each other in terms of shifts in prices. When one of the currencies starts to depreciate sharply, we can assume that we will probably see the same in other currencies. Since the second biggest and most popular cryptocurrency is Ethereum, we wanted to see if introducing it to a model predicting Bitcoin prices would improve the results.

We can consider different time series as channels, similar to the real images where there are channels for red, green and blue colors. We generated grids of indicators based on Ethereum data and used them as a second channel. Now, a single input to our neural network was a 15x15 grid with two channels: Ethereum indicators and Bitcoin indicators (both calculated for the same date).

Since a single sample is now twice as big as before, we had to tweak a little bit with the architecture in order to give the model more expressive power. Before doing so, training of this neural network was constantly failing. We have enlarged the convolutional and linear layers of the network. Number of channels generated by the two convolutional layers were changed from 16 and 32 to respectively 32 and 64. Output size of the first linear layer was changed from 32 to 128. We also had to use a much higher learning rate - 0.15 - with momentum factor of 0.9. Eventually, the results were very similar to those achieved by the network fed with 1 channel input:

Test set size:	around 10000 (34 days)
Information Coefficient:	0.19
Accuracy:	56.5%

Predicting Ethereum using Bitcoin

After experimenting with trading strategies described in the next chapter, we decided to see how well can we predict Ethereum price change looking only at the Bitcoin data. We used the same SGD parameters (learning rate 0.05 and momentum factor 0.95) and layers sizes as in the model that was using Bitcoin to predict Bitcoin price change.

Test set size:	around 10000 (34 days)
Information Coefficient:	0.15
Accuracy:	52.71%

IC shows a small correlation between predictions and ground truth, accuracy is a little bit better than a random classifier. It seems to prove that the Bitcoin and Ethereum price fluctuations are linked together.

Predicting Ethereum using Ethereum

What surprised us was that we couldn't train a network that is trying to predict Ethereum price change based on grids of Ethereum indicators. We have tried multiple SGD parameters' combinations but the best results we got were no better than a coin flip.

Chapter 4

Trading bot

4.1 Strategy

Accurate neural network is undeniably crucial in algotrading. A perfect model would guarantee high income. That being said, a perfect model doesn't exist, therefore proper trading strategy must be chosen carefully. One that takes into consideration neural network's strengths and weaknesses, as well as rules of the exchange.

We decided not to experiment with splitting assets between different cryptocurrencies and play only on one of them. After backtesting different approaches, we settled for strategy that is actually pretty simple, yet lays best results. It goes as follows:

1) If predicted rise in price is above 0.2% and there's enough assets in the wallet, spend the whole budget to buy the cryptocurrency.

2) As long as predicted rise in price is above 0.2%, hold the asset.

3) When predicted rise in price falls below 0.2%, sell everything.

Why 0.2%? Trade fee on Binance is 0.1%. Taking into account that we have to make two trades, more precisely buy and sell, rise in price above 0.2% guarantees a profit.

4.2 Results

The best results were achieved for a bot that trades Ethereum and makes decisions based on predictions of Bitcoin price change - it made 21% return in a period of 34 days.

It turns out that in the test set (on which we performed the backtesting of this strategy) of 10000 samples, Spearman correlation between Bitcoin and Etherum

price changes is 0.74. It shows that Bitcoin's predictions can give as a lot of information about Ethereum's price movement.

For that strategy, we have used predictions generated from 2D CNN fed with 1 channel input of Bitcoin indicators. Figure 4.2 shows that a majority of the trades were made when the market was highly unstable. Standard deviation of ETH price during the first week, when the bot made 19 trades, was 599. During the next 4 weeks there were only 5 transactions and the standard deviation 186.

Most of the revenue was made during the first days of trading. When the market is stable, this strategy keeps more or lees a balanced account. No revenue or heavy losses occur. Longest period without any trades was 8 days.

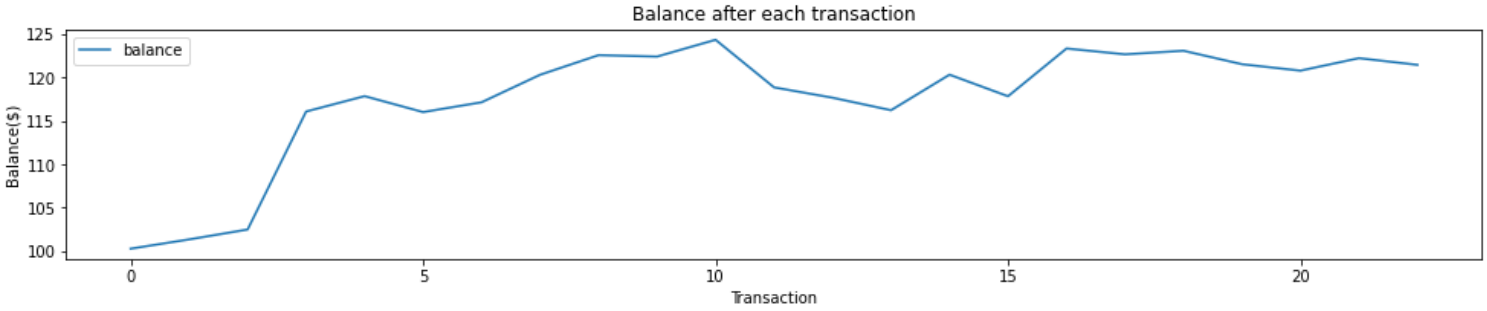


Figure 4.1: Balance of our account after the transactions

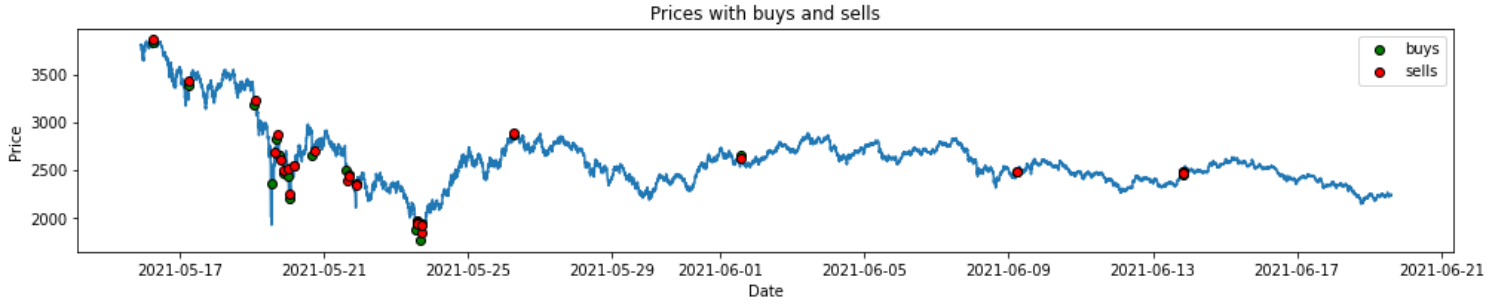


Figure 4.2: Buys (green) and sells (red) executed by the bot

Chapter 5

Storing the data using Influx

5.1 Introduction to Influx

InfluxDB [10] is a time series database designed to handle high write and query loads.

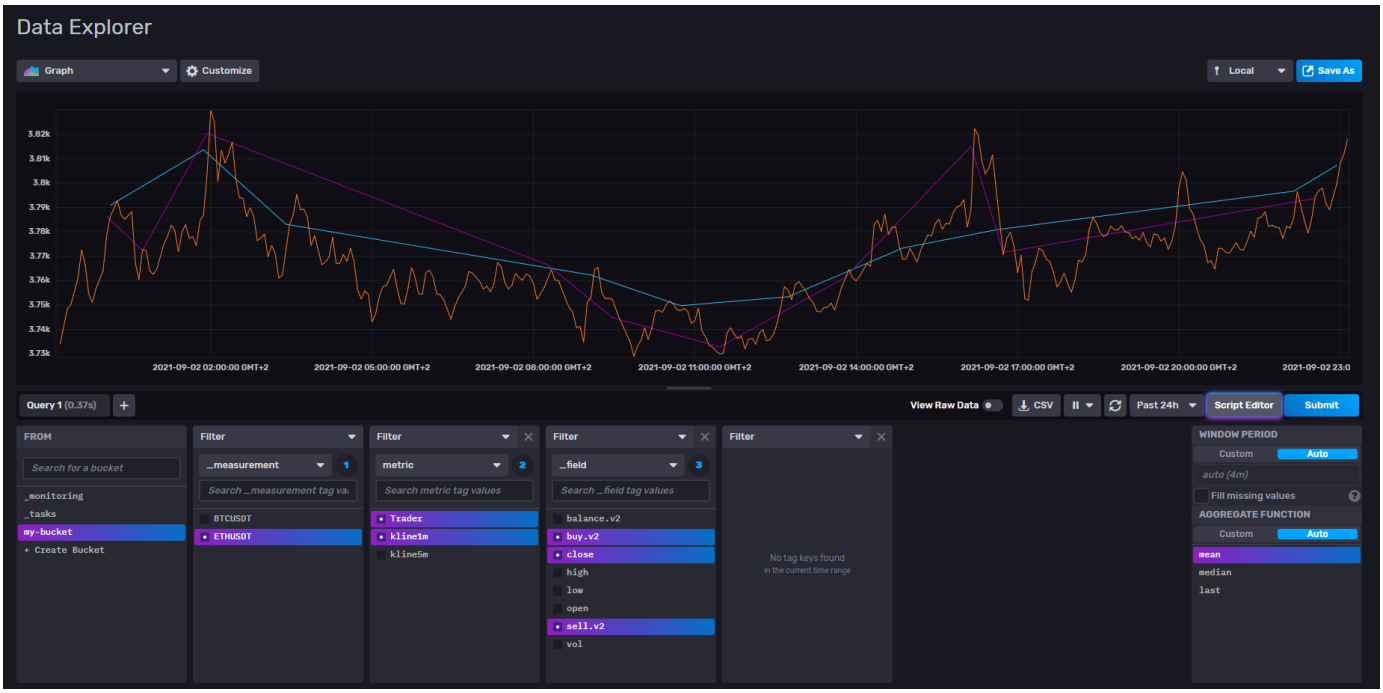
As it is simple to use, it fits well for small projects, just like ours. However, it does not mean it underperforms for big businesses. Influx lets you manage organizations, data buckets and users. Each organization has its own data buckets, which are basically separate databases. Users can belong to one or more organizations, where they can be given access to certain buckets. That is handled using generated tokens, which give permission to query buckets or also edit them.

Influx provides powerful API with client libraries for many commonly used programming languages. It minimizes coding needed to operate the database. Queries are executed using InfluxQL, which stands for Influx Query Language. It is simple and intuitive. Here is an example:

```
from(bucket: "my-bucket")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "ETHUSDT")
  |> filter(fn: (r) => r["metric"] == "kline1m" or r["metric"] == "Trader")
  |> filter(fn: (r) => r["_field"] == "buy.v2" or r["_field"] == "sell.v2" or r["_field"] == "close")
  |> aggregateWindow(every: v.windowPeriod, fn: mean, createEmpty: false)
  |> yield(name: "mean")
```

This query fetches prices at which we bought or sold ETH, as well as continuous price over time.

Big advantage is the InfluxUI, where UI stands for User Interface. It is accessible from the web browser. There, user can log in to his account, then browse, visualise and manage data he has access to. Thanks to the intuitive layout it does not take much time to navigate. Visualisation tool is great, user can query data in matters of seconds using convenient Query Builder. Below, we can see the visualisation of previous query, which was built using Query Builder.



5.2 Usage of the database

Being a group of peers working on a small project, we didn't bother using any users and permissions management tools. Simply, one organization, one bucket, one admin user.

We split our bucket into two measurements: BTCUSDT and ETHUSDT. As we used Bitcoin prices to compute predictions, in BTCUSDT we stored Bitcoin prices, metrics computed from those, and most importantly, our predictions. In ETHUSDT, we kept prices at which bot made transactions and our account's balance after each sell. Lastly, Ethereum prices for reference.

InfluxUI served us for data visualisation and keeping track of all the trades that occurred.

Chapter 6

Workflow management using Airflow

6.1 Introduction to Airflow

Data flows. While this sentence may sound obvious, when deciding to start a project oriented around data, one has to realize what comes with it. Before any outcomes can be seen, raw data has to be obtained, processed and then turned into a final product. That is what is called a Data Pipeline. And it is only a very simple example. In reality, what often occurs is missing records, outliers, download errors, hardware fails and many more. Everything has to be taken care of, otherwise the whole system fails. With new data flowing in regularly, it is nearly impossible to repeat this process manually.

Apache Airflow [11] is a workflow management platform, that lets you schedule and execute sequences of tasks using DAGs, which stands for Directed Acyclic Graph. They are programmed as Python scripts. Each DAG is an independent set of tasks to be performed in a specified order and has it's own schedule, which is defined using cron [12]. It is easiest to explain on an example. Figure 6.1 shows a tutorial DAG that comes with the installation.

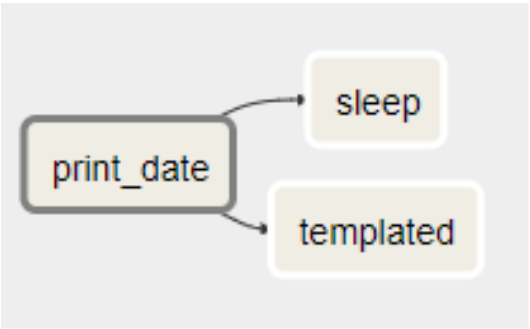
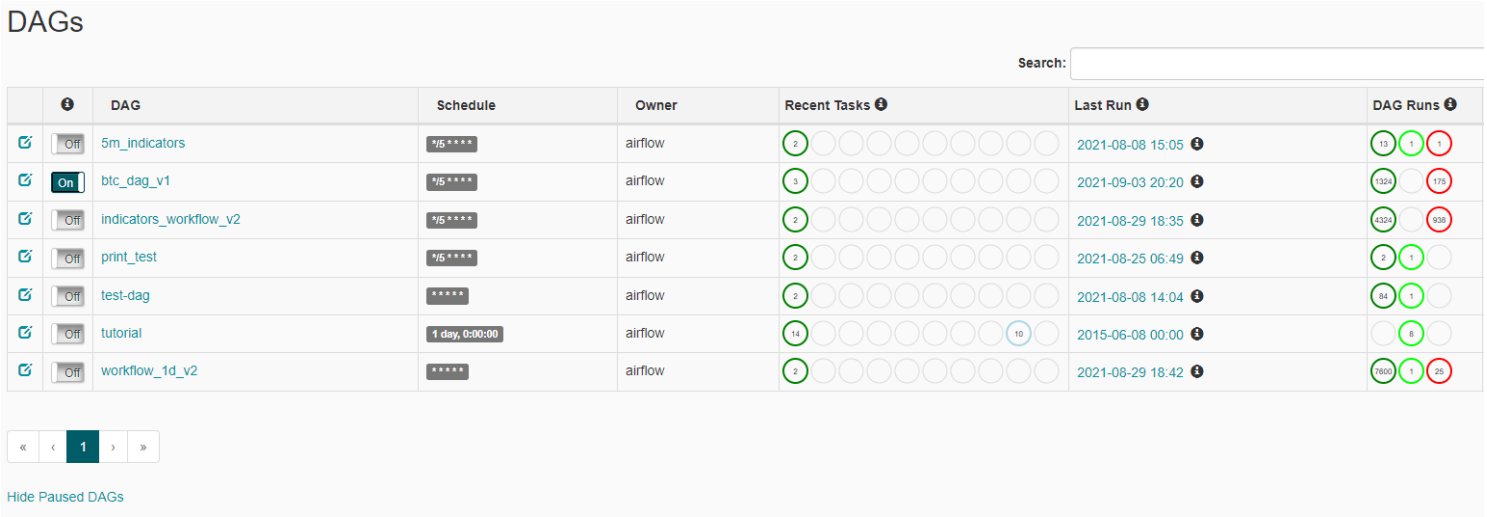


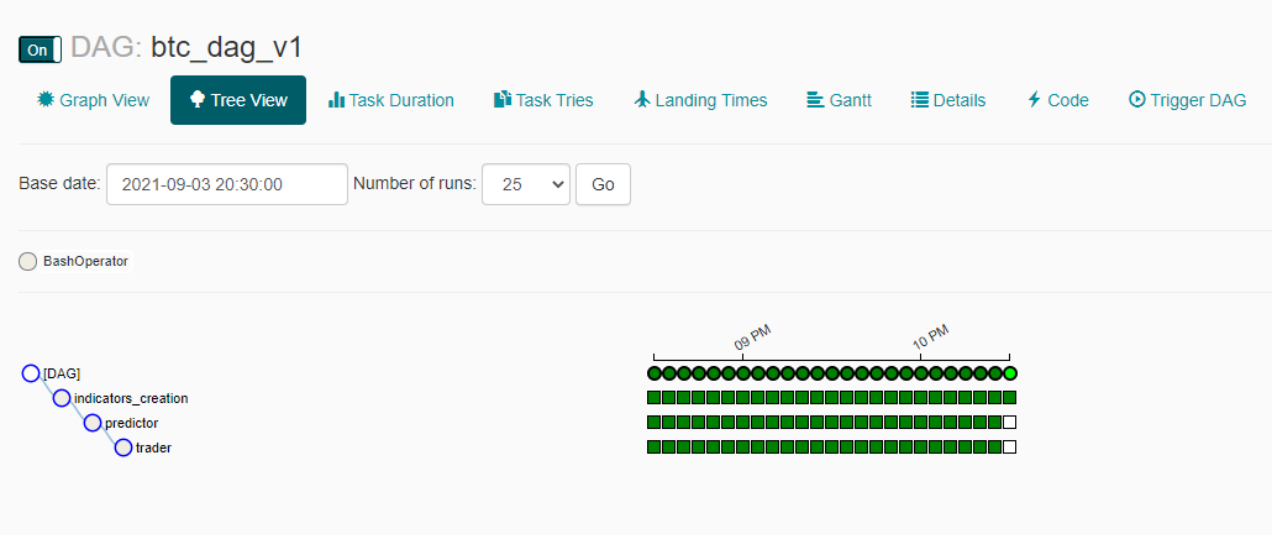
Figure 6.1: Exemplary DAG

We have three tasks to perform. First is the *print_date*, and only after it completes, following tasks are run. Notice that those don't depend on each other, so even if *sleep* fails or gets into an infinite loop, *templated* will be executed anyway. Important thing to notice is that schedule interval is specified for the whole DAG, not on individual tasks. Therefore, every task that belongs to a DAG will be executed the same amount of times.

Another big advantage of using Airflow is user friendly UI. On the top level, it lets you monitor and manage all DAGs.



You can also investigate individual DAGs. Multiple useful tools allow you to trigger DAG runs, rerun individual tasks or see console logs. All of that with an intuitive visualization.



We have used open source docker image of Airflow [13].

6.2 Workflow for 1-minute time intervals

Similarly to the development of the models, we used one-dimensional CNN as a way to test different approaches. Our goal was to have a workflow that starts every minute and completes two tasks:

1. Calculate and save to the database the newest lagged return.

2. Fetch the data needed for the 1D CNN (20 past lagged returns), then make and save to the database a prediction of the lagged return for the next minute.

Our first idea was to make separate Docker images for each of the tasks. Container created from such image would try to complete the task on the startup. We can achieve that by specifying in the Dockerfile what Python script should be executed after the creation of the container [14]. Airflow's DAG would then be composed of two consecutive Docker Operators [15]. This type of operator creates and starts a container from the indicated image. After the task is completed, Airflow deletes the container.

This approach has proven to be too costly in terms of time and resource. Execution of a single Docker Operator on our computer takes on average about 30 seconds. Thus, model's prediction wouldn't be useful at all because it would be generated after about 60 seconds. By the time we would save the predicted price change to the database, the ground truth price change will be known to us, because the new minute starts.

Because of that, we decided to implement both of the tasks on a single Docker image. On the startup, it runs a script that invokes the function responsible for the first task and after its completion, it invokes the function responsible for the second task. To deal with the time delay of around 30 seconds, the first task waits for the current minute to end. When the new minute starts and the newest price kline is in the database, it starts the execution of the task. Because of that, when the DAG scheduled for 14:53:00 starts at 14:53:30, the script waits for 14:54:00 and proceeds with the workflow that makes a price prediction for 14:55:00.

We completely solved the problem of time delays but stopped taking advantage of what Airflow can give us - the tools to execute multiple jobs and inspect their execution. Transferring the logic of how the tasks are run into a script and having a DAG with a single operator that starts this script is not a good way of using Airflow, it is a waste of its potential and a bad practice. We decided to do a trade-off and try to implement the next workflow in a more correct way, but with some time delays in execution.

6.3 Workflow for 5-minutes time intervals

This workflow works on the Bitcoin data from 5-minutes time intervals. It uses the pre-trained 2D CNN that predicts Bitcoin price change based on Bitcoin indicators and a trading strategy for Ethereum. It is composed of three distinct tasks:

1. Calculating indicators for the 2D CNN.
2. Fetching the indicators and generating a prediction of a price change for the next 5 minute time interval.
3. Executing a trading strategy on Binance based on the newest prediction.

The first and the second tasks are implemented on a single Docker image and the third task has its own image. This time, there is no Entrypoint that starts a Python script after creating a Docker container. We manually create and start containers from those two images. Then, the DAG can use Bash Operator [16] which we use to execute a dockerized script responsible for a given task. Code below shows the implementation of the DAG:

```
with DAG(
    'btc_dag_v1',
    default_args=default_args,
    schedule_interval="*/5 * * * *",
    catchup=False
) as dag:

        t1 = BashOperator(
            task_id = 'indicators_creation',
            bash_command=
            "docker exec -t btc_5m_dag python create_indicators.py {{ ts }}"
        )

        t2 = BashOperator(
            task_id = 'predictor',
            bash_command=
            "docker exec -t btc_5m_dag python make_prediction.py {{ ts }}"
        )

        t3 = BashOperator(
            task_id = 'trader',
            bash_command=
            "docker exec -t alpha_trader python Trader_runner.py {{ ts }}",
            trigger_rule='all_done')
```

```
t1>>t2>>t3
```

Executing a Bash command on an already started container on the computer we are deploying this workflow takes on average 4-6 seconds.

6.3.1 Calculating indicators

During the development of the neural network used in this workflow, we have saved the information about the best indicators and the grid ordering to a JSON file.

After the task has been started and we made sure that a new Bitcoin kline has been saved to the database, we fetch the required data and compute the indicators. Then, we save them via a simple query.

To compute the grid, we read the optimal arrangement from a file. Then, we rearrange the data accordingly and the grid is ready to be passed to the model.

6.3.2 Generating model's prediction

We stored a pre-trained 2D CNN using PyTorch save and load functions [17]. After the indicators are saved to the database and a grid is generated, we can easily load this model from a file, generate a prediction on the newest data available and save it to the database.

Figure 6.2: Predictions stored and visualized in InfluxDB

6.3.3 Trading bot

Our trading bot is contained in a simple script that acts accordingly to the current prediction. It runs right after the model's task. However, as oppose to the previous task, this one runs no matter if upstream tasks succeeded. That's because, in case of missing predictions, a safe bet is to sell our assets. If not for that, worst case scenario would be a bot holding his assets until someone notices the problem, which may result in high loss. To execute the trades, we have again used the Binance API.

Chapter 7

Live trading results

The trading strategy we have backtested is very passive. Because of that, to test our platform, we decided to introduce a few changes that would result in a higher rate of trades. The changes are as follows:

1. Smaller buy threshold - previously 0.2%, now 0.04%.
2. New holding mechanism - we keep our assets until predicted return is negative.

This strategy has proven to be much more aggressive. On average, it executes 20 transactions a day as opposed to one every few days with previous strategy. On the other hand, it is less stable and tends to make risky transactions that sometimes result in loses. Eventually, it leads to bad transactions diminishing the profits made out of good ones.

After five days of trading Ethereum, the bot made 3.5% revenue. However, during that time the Ethereum price has been growing steadily which was a great help for the strategy that promotes holding the assets.

Figure 7.1: Balance of the account during 5 days of trading

Bibliography

[1] Ernest Chang - *Quantitative Trading*

[2] python-binance wrapper for the API [https://python-binance.readthedocs.io/en/latest/]

[3] Stefan Jansen - *Machine Learning for Algorithmic Trading: Predictive Models to Extract Signals from Market and Alternative Data for Systematic Trading Strategies with Python*

[4] Websockets provided by Binance [https://python-binance.readthedocs.io/en/latest/websockets.html]

[5] PyTorch SGD implementation [https://pytorch.org/docs/stable/generated/torch.optim.SGD.html#torch.optim.SGD]

[6] PyTorch TensorDataset [https://pytorch.org/docs/stable/data.html]

[7] PyTorch DataLoader [https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader]

[8] Ta-Lib Package [https://www.ta-lib.org/]

[9] SciPy function for Spearman correlation [https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html]

[10] InfluxDB [https://www.influxdata.com/]

[11] Apache Airflow [https://airflow.apache.org/]

[12] Cron scheduling [https://en.wikipedia.org/wiki/Cron]

[13] Open source image of the Airflow [https://github.com/puckel/docker-airflow]

[14] Dockerfile's Entrypoint [https://docs.docker.com/engine/reference/run/#entrypoint-default-command-to-execute-at-runtime]

[15] Airflow's Docker Operator [https://airflow.apache.org/docs/apache-airflow-providers-docker/stable/_api/airflow/providers/docker/operators/docker/index.html]

[16] Airflow's Bash Operator [https://airflow.apache.org/docs/apache-airflow/stable/howto/operator/bash.html]

[17] PyTorch tools for saving and loading models [https://pytorch.org/tutorials/beginner/saving_loading_models.html]