



Ca' Foscari University of Venice  
Department of Environmental Sciences, Informatics and Statistics

---

Master's Degree Programme in Computer Science

Master's Thesis

## **A data-driven approach to forecasting Bitcoin price using on-chain metrics**

Candidate:

Vrabii Simion - 867882

Supervisor:

Prof. Giovanni Fasano

Academic Year 2024-2025

*To all those who have been close to me during difficult times!*

## Acknowledgements

I would like to spend a few words to thank all those people that have helped me and supported me during this journey.

First of all I would like to thank my supervisor, Giovanni Fasano, for his guidance, continuous support, patience, motivation, enthusiasm and knowledge.

Finally, I would like to express my deepest appreciation to my family for being there for me during hard times.

## **Abstract**

Cryptocurrencies represent a rapidly expanding market, every day more and more people are looking for new investment opportunities and new strategies to identify trends and their prices. Most investors mainly use technical analysis to operate on the markets. Bitcoin is one of the most valuable cryptocurrencies and was introduced in 2008 by Satoshi Nakamoto with the introduction of the blockchain, considering the growing value based on blockchain, today bitcoin is considered to large extent. In addition to revolutionizing the idea of money, the blockchain has opened the door to the study of new metrics to be able to identify the market direction of a specific cryptocurrency. These metrics are called chain metrics and are mainly based on the study of transactions and tokenomics of the blockchain itself. The following research aims to identify and couple on-chain metrics with machine learning and deep reinforcement learning models, in order to predict the price of bitcoin for short and long term price models.

# Summary

<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Blockchain and Metrics</b>	<b>17</b>
2.1	Bitcoin's Blockchain . . . . .	17
2.2	On-Chain Metrics . . . . .	18
2.2.1	MVRV (Market value to Realized value) . . . . .	19
2.2.2	Hash Ribbon . . . . .	19
2.2.3	Coin Days Destroyed and Liveliness . . . . .	19
2.2.4	Net Unrealized Profit/Loss (NULP) . . . . .	20
<b>3</b>	<b>Mid and Long Term Forecasting</b>	<b>21</b>
3.1	Stock to flow . . . . .	21
3.1.1	Dataset . . . . .	23
3.2	Regression and Least Square Optimization . . . . .	23
3.3	Support Vector Machines (SVM) . . . . .	24
3.3.1	Optimization Problem . . . . .	24
3.3.2	Soft Margin SVM . . . . .	25
3.3.3	Dual Formulation and Support Vectors . . . . .	25
3.3.4	Kernel Trick . . . . .	25
3.3.5	Geometric Interpretation . . . . .	26
3.3.6	Support Vector Regression (SVR) . . . . .	26
3.3.7	Evaluation Metrics . . . . .	27
3.3.8	Models Result . . . . .	27
3.4	K-means Clustering . . . . .	28
3.4.1	Algorithm Description . . . . .	29
3.4.2	Convergence and complexity . . . . .	29
3.5	Gaussian Mixture Models (GMM) . . . . .	30
3.5.1	Model Definition . . . . .	30
3.5.2	Learning via Expectation-Maximization (EM) . . . . .	30
3.6	Identifying Stock-to-Flow Signals via Unsupervised Learning . . . . .	32
3.6.1	Model Training . . . . .	32
3.6.2	Model Training Steps . . . . .	32
3.6.3	Model Performance . . . . .	33
3.7	SVR-Based Approach to Next-Day Trading Decisions" . . . . .	36
<b>4</b>	<b>Short Term Forecasting</b>	<b>40</b>
4.1	Introduction to Neural Networks . . . . .	40
4.1.1	The Perceptron . . . . .	42
4.1.2	Activation functions . . . . .	43
4.2	The Loss Function . . . . .	44
4.3	Deep Learning . . . . .	46
4.4	Convolution Neural Networks . . . . .	46
4.5	Long Short Time Memory . . . . .	48
4.5.1	CNN-LSTM . . . . .	50
4.6	Deep Reinforcement Learning . . . . .	53
4.6.1	Q-learn Algorithm . . . . .	55

4.7	k-Nearest Neighbors (k-NN)	57
4.7.1	Mathematical Foundation	57
4.7.2	Distance Metrics	57
4.7.3	Algorithm Pseudocode	58
4.8	Multi Expert Trading System	58
4.8.1	System Architecture Overview	59
4.8.2	Stabilization	63
4.8.3	Results	64
4.9	SVM Model	70
4.10	Stock To Flow Model	72
4.11	Multi Agent System	74
4.12	Single Agent Environment	74
4.13	Agent	77
4.14	CnnLstmNetworkNetwork	79
4.15	ReplayMemory	81
4.16	Multi Agent System	82

## Glossary of Terms

<b>Term</b>	<b>Description</b>
Addresses	Unique identifiers used to send and receive cryptocurrency.
Agent	Entity that takes actions in an environment to maximize rewards.
Blockchain	A decentralized, immutable ledger of transactions grouped in blocks.
Block	A group of verified transactions added to the blockchain.
Bitcoin	The first cryptocurrency, based on proof-of-work and blockchain technology.
CDD	Coin Days Destroyed, measuring economic activity of spent coins.
CNN-LSTM	Hybrid model combining convolutional networks with LSTMs.
Crab	Addresses holding at least 1 BTC but less than 10 BTC.
Cryptocurrencies	Digital currencies that use cryptography for secure transactions.
Environment	The external system with which an agent interacts.
Fish	Addresses holding at least 10 BTC but less than 100 BTC.
GMM	Gaussian Mixture Model, probabilistic clustering based on Gaussian distributions.
Hash Ribbon	Indicator based on Bitcoin's mining difficulty and hash rate.
JSON	Lightweight data-interchange format used for structured data exchange.
K-means	Clustering algorithm grouping data into k clusters.
Kernel	Function that transforms data for use in SVMs.
LSTMs	Long Short-Term Memory networks, specialized RNNs for sequence data.
MAE	Mean Absolute Error, average of absolute prediction errors.
Mega whale	Addresses holding 10,000 BTC or more.
Miners	Participants who validate transactions and create new blocks by solving cryptographic puzzles.

<b>Term</b>	<b>Description</b>
MSE	Mean Squared Error, average of squared prediction errors.
Multi-expert trading system	A system combining multiple models/strategies for trading.
MVRV	Ratio of market value to realized value, used to detect over/undervaluation.
Nodes	Computers that maintain and validate the blockchain.
NUPL	Net Unrealized Profit/Loss, shows whether investors are in profit or loss.
On-chain metrics	Indicators derived from blockchain data (e.g., transactions, addresses).
Peer-to-peer system	A distributed network where participants communicate directly without intermediaries.
Plankton	Addresses holding very small amounts of BTC (non-zero but below 0.001 BTC).
Policy	Strategy followed by an agent in reinforcement learning.
Proof-of-stake	Consensus mechanism where validators are chosen based on staked tokens.
Proof-of-work	Consensus mechanism requiring computational effort to validate blocks.
$R^2$	Coefficient of determination, shows how well a model explains variance.
RBF	Radial Basis Function, a popular kernel for SVMs.
Reinforcement learning	Learning by interaction with an environment through rewards.
$\sigma(\cdot)$	Sigmoid activation, maps values to range (0,1).
SHA256	A cryptographic hash function used in Bitcoin for block verification.
Shark	Addresses holding at least 100 BTC but less than 1,000 BTC.
Stock-to-flow	Model comparing scarcity (stock vs. new supply) to predict price.
Supervised learning	Training models with labeled data.
SVR	Support Vector Regression, regression version of SVM.
SVMs	Support Vector Machines, used for classification tasks.
$\tanh(\cdot)$	Hyperbolic tangent activation, maps values to (-1,1).
$\tilde{C}_t$	Candidate cell state in LSTM, created by tanh activation.

<b>Term</b>	<b>Description</b>
$\odot$	Element-wise (Hadamard) product between vectors.
Transactions	Records of value transfer stored in the blockchain.
Unsupervised learning	Learning patterns from unlabeled data.
UTXO	Unspent Transaction Output, representing available spendable coins.
Whale	Addresses holding at least 1,000 BTC but less than 10,000 BTC.
$[h_{t-1}, x_t]$	Concatenation of previous hidden state and current input.

## Chapter 1

### Introduction

In recent years, the crypto markets have emerged as a rapidly growing market and many investors are attracted by these new investment opportunities. Bitcoin is one of the most valuable cryptocurrencies and was introduced in 2008 by Satoshi Nakamoto with the introduction of the blockchain. The value of bitcoin has on average grown over time. In fact on May 22, 2010 Laszlo Hanyecz bought two pizzas paying 10 000 bitcoins and on September 13, 2025 considering that the price of bitcoin was around 115 586 USD, the value of those pizzas would be 1 155 860 000 USD. Considering that in September 2025 a pizza costs around 10 USD, with 10 000 bitcoins today we are able to buy at least a pizza for each person of Italy, France and Moldova, since the population of Italy is around 58 millions, France 68 millions and Moldova 2,38 millions. It is impressive how much the price has increased in the last years, in such a way that today bitcoin is often considered to be digital gold. On the other hand it is challenging to build a system that can predict the movement of stock prices with a high accuracy, because stock prices are volatile, complex and they could be influenced by external factors. In the case of cryptocurrencies, these challenges are even more pronounced because unlike traditional markets, cryptocurrency markets:

- operate 24/7
- are more susceptible to manipulation since they are relatively less regulated
- are more vulnerable to the movements of a few large investors (whales) since they usually have a smaller market capitalization than stocks or bonds.

There are multiple methodologies to study the stock market and predict the behavior of stock price movement. Most speculators use technical analysis to operate on the markets, an analysis based on the study of the price, volume, and some technical indicators in order to identify patterns.

Cryptocurrencies are based on blockchain, which is revolutionizing the concept of electronic money, web and industries through the concept of asset tokenization.

Blockchain is a type of digital ledger, but with a unique characteristic: it isn't stored on a single computer, but is distributed across a network of computers. We can imagine it as a shared, indelible notebook, where each page (a block) contains a series of transactions. Once a page is written, it can no longer be modified, because each new page includes a reference to the previous one, creating a secure, chronological chain.

The most famous example of blockchain is given by the one associated to Bitcoin, which relies on a network of computers which forms a distributed peer-to-peer system in order to maintain its ledger. To ensure that transactions are valid, the network uses a mechanism called Proof-of-Work (PoW). This means that the special nodes called miners have to perform a computationally expensive task which consists in solving a complex mathematical puzzle: whoever succeeds first can add the next block to the chain and receives a reward.

Other blockchains, however, use a different system, e.g the Proof-of-Stake (PoS). This doesn't require enormous computing power. The computers that can validate

blocks, called validators, are chosen based on how much cryptocurrency they have staked. Essentially, the higher their stake, the greater their chances of being chosen to validate the next block.

Unlike usual financial transactions, blockchain transactions are accessible to everyone if the blockchain is public. In this distributed peer-to-peer system, nodes maintain a copy of the blockchain. Each block contains transactions, and transactions must be validated by miners before they can be added to the blockchain. Miners add transactions to blocks, and then add blocks to the blockchain. This process is not immediate in case of PoW because it's computationally expensive. We can represent the transaction workflow with the below image:

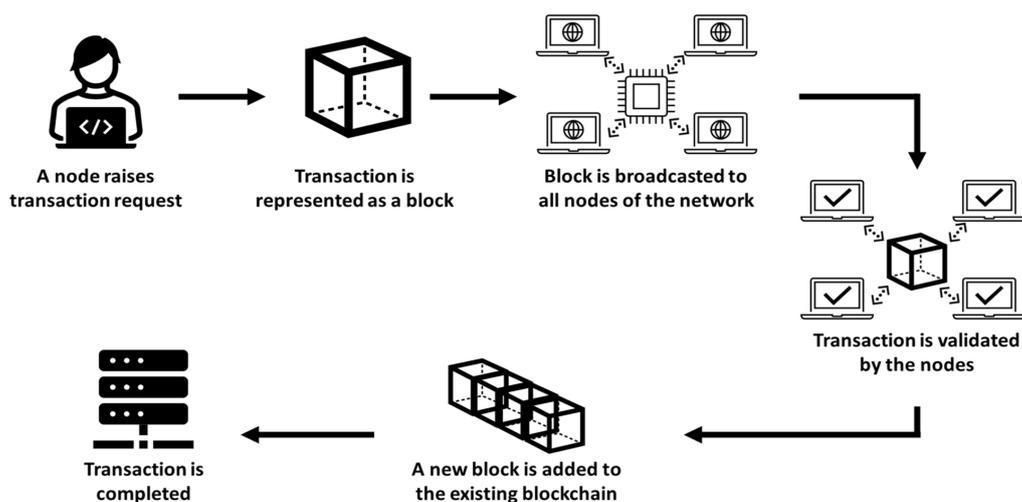


Figure 1.1: Bitcoin Transaction Flow (Diagram)

Given that blockchains record several types of information like transactions, blocks and wallet addresses that are publicly available, this opens the doors to the study of new metrics that can give useful information to be analysed and included in prediction model. In order to keep track of the coin ownership, the Bitcoin payment system uses the so called unspent transaction output UTXO: we can think of them as individual digital coins. When we check the Bitcoin wallet balance, the static number represents the total sum of all these UTXOs a given address have. When a user makes a payment, he does not spend his entire balance but chooses one or more of these UTXOs as input. The transaction consumes them and creates new ones:

- One or more new UTXOs will go to the recipient.
- The other UTXOs represents the change and is returned to the sender.

This model is fundamental to Bitcoin's reliability because it allows for transparent balance management and it makes double spending impossible, since a UTXO can only be spent once. A small technical detail not to be overlooked is that the amount of UTXOs you use affects transaction fees. For example, spending many small UTXOs in a single transaction can increase costs. By tracking UTXOs, it is possible to reconstruct coin movements and infer investor behavior. Anyone can view on blockchain the details of a Bitcoin or other cryptocurrency transaction: the sender's

address, the recipient's address, the amount exchanged, and fees. This transparency is crucial because it allows us to track exactly how many tokens an address holds at any given time, as well as how often and how much it spends. Analysing the funds held by an address is a keynote aspect, as it allows us to classify investors based on the amount of cryptocurrency they own. The most well-known categories of owners are often associated with animal names to convey the size of the investment: from Fish to Crab, all the way to Shark, Whale, and, for the largest, Mega Whale. Essentially, the larger the animal, the more significant the investor in terms of capital. The performance of these categories allows us to identify investor behavior over time. If the number of large whales decreases, it means that investors with larger capital are closing positions. If, on the other hand, the number of coins held by a given address increases, it means they are buying. This information is extremely valuable and has piqued the interest of numerous quants and researchers.

The following thesis aims to study the functioning of the blockchain and, above all, identify on-chain metrics that can provide valuable information to be used for training machine learning models of Bitcoin price. Indeed, from the analyses conducted, there are several indicators that can be used to extract information, including the realized cap or MVRV (Market Value Over Realized Value), long-short-term holding, the number of blocks created, the reward of miners, hash ribbon, etc. The second step is to research which models can be used to create an automated trading system that can give appealing results. Machine learning models are evolving rapidly, particularly neural networks and deep learning. There are three main machine learning paradigms (supervised learning, unsupervised learning, and reinforcement learning), which depend on how the dataset is constructed, how the training process is carried out and how they are used to make the predictions. The supervised model is primarily used to address classification and regression problems, and it requires the dataset to consist of two fundamental elements: a series of features, or columns, that will be used to make the prediction, and a corresponding label for each list of feature usually named record. In this way, the algorithms learn to predict a given label and then use a loss function to compare the actual value with the predicted value. A concrete example in the financial world is given by market price forecasting, in which a vector might represent the market price over the last few days, while the label represents the price of the day we're forecasting. Unfortunately, not all problems can be solved this way. In some cases, it's not always possible to have a dataset with labels, which is precisely why the second machine learning paradigm, i.e. unsupervised learning, comes into play. It allows us to identify potential groups, called clusters, present in the data without knowing previously the labels. In some cases, however, there is a need to find new patterns and strategies to achieve a specific goal. Reinforcement learning is a learning paradigm that differs significantly from the previous two in terms of the way it learns. It involves several interacting components, including the agents, which can perform actions into an environment. For each action, it receives a positive reward if the action is correct and a negative reward if the action is incorrect. The choice of actions is determined by a policy that indicates the best action to take given the agent's current state. The agent learns by interacting with an environment, and each action it takes alters the state of the environment, and for each action, it receives a reward. In this way, thanks to a sufficiently high number of iterations, the agent is able to update the policy to correct errors and understand the best action to take in different states.

Reinforcement learning has proven highly effective in strategy games, which is one of the main reasons for which researchers tend to use reinforcement learning to model financial trading as a game. The task can be viewed as an agent that interacts with the market periodically, such as every minute, hour, or day, taking actions that are considered tied to a specific market. For example, using a certain portion of its capital to place a limit order for a specific asset, or selling a certain amount of an asset already held at a certain price. Finally, the agent can also consider taking no action at a specific time because it's considered the best thing to do at that moment, meaning maintaining open positions and not opening new ones. Periodically, the Environment produces new data. This data is collected and represents a state based on which the agent will execute actions. Each action the agent performs is necessary to achieve a specific objective, such as maximizing capital or reaching a specific annual target. Every action the agent performs alters the state of the environment and can also alter the state of the market. If the agent has significant capital available and the asset it is acting on has a low capitalization, this can alter the market. However, in reality, it is often unrealistic for a single agent to significantly alter the market. Our goal is to create a promising trading system with positive results that can provide guidance on the actions to take (buy, sell, do nothing) to maximize the initial capital. To make short-term forecasts (i.e., say the next hour) and medium-to-long term forecasts (i.e., say the next day, next week), various supervised, unsupervised, and deep reinforcement learning models were combined, working together to achieve the common goal.

Support Vector Machines (SVMs) have been a valuable tool for classification and regression tasks in the previous years. In fact, in some cases, when the classes in the dataset are linearly separable, they have achieved excellent performance, and in some cases they have even outperformed neural networks. This is because, unlike neural networks, SVMs are much simpler to train and, above all, because the solution that SVMs provide represents a global minimum. This is not for neural networks whose training is based on gradient descent. In some cases, when parameters such as learning rate and momentum are not configured correctly, they can end up in a local minimum that degrades performance. SVR (Support Vector Regression) is a modified extension of SVM adapted for regression tasks. It is used to forecast the next day's price by considering the market price, on-chain metrics, and other data from the previous 7 days. As shown in the next sections, the choice of the sliding window was decided based on several tests. To choose the sliding window size, we analysed the model's performance using sizes ranging from 7 to 45 days in the past. We found that, as partially expected, the smaller the window size, the better the performance in terms of mean square error (MSE), average annualized average (MAE), R2, and gain and loss (P&L). Given that our primary goal is to provide insight into the actions an investor might choose (buy, sell, or do nothing), we transformed the forecast price value information into an indicator that returns a 'buy' if the forecast price is higher than the forecast price for the previous day and a 'sell' if the price is lower than the forecast price for the previous day. Since transactions have a cost, performing operations if the predicted price does not exceed a certain threshold could degrade the results. For this reason, a threshold was introduced. Beyond this threshold, the agent executes the actions; otherwise it returns the action 'do nothing'.

In recent years, several models have been created to predict the long term price

of bitcoin. The stock-to-flow model is one of those that has attracted the most interest, as it is based on the concept of bitcoin's rarity, meaning that every 210,000 blocks, the reward given to miners for creating blocks is halved, causing the price of the coin itself to rise. The stock-to-flow model has been widely used in economics as a method for tracking the scarcity of goods in commodity markets (such as gold, silver, oil, etc.). Given the nature of bitcoin, it has also been introduced into the world of cryptocurrencies for goods that have this scarcity effect, such as bitcoin. The stock-to-flow value is calculated as the ratio between the existing stock and the flow, i.e the annual production of that good. In our implemented model, stock represents the number of bitcoin blocks mined to date, while flow represents the number of bitcoins mined in the year. To consider the relationship between price and stock-to-flow value, two methods were used: linear regression and support vector regression, which use a linear kernel. Both methods revealed that the price of bitcoin is closely correlated with stock-to-flow. The almost linearity between stock-to-flow and price allows us to understand the long-term trend, but to obtain contrary indications on what action to take on a given day, this approach needs to be improved. To do this, the distance between a given point and the regression line was calculated, resulting in a distance vector on which two clustering algorithms, K-means and GMM, were then applied to identify three clusters. These three clusters represent the best indication in order to buy bitcoin, sell it, or do nothing. Using unsupervised learning, it was possible to assign a buy, sell, and 'do nothing' signal to better interpret the results of the stock-to-flow model. Given the widespread popularity of neural networks, and particularly the success of deep learning in computer vision, speech recognition, finance, and many other tasks, they have also attracted considerable interest in current research. LSTMs are type recurrent neural networks that have proven highly effective at identifying temporal patterns thanks to their structure. Modern deep learning models use convolutional networks to perform the so-called feature extraction part, which allows significant data and patterns to be extracted through a process called convolution. This consists of sliding and applying a specific matrix, called a kernel, to the input data. This process as the rule thumb "reduces" the input dimension to reveal the most significant data. In the following research, LSTM-type neural networks were used and compared with CNN-LSTM neural networks to forecast the next day's price based on the previous day's price. Using CNN-LSTM for forecasting immediately demonstrated improvements in terms of MSE, MAE, and R2. Another widely used approach for classification and regression problems is random forests. These are machine learning algorithms based on decision trees, i.e., models that loosely speaking are based on conditional rules. Since a single tree can be unstable, random forests are composed of a set of trees, each trained on a random sample of data from the dataset. This makes the machine learning model more stable and allows it to generalize better to new data. The principle behind random forests is a powerful inspiration for the models created for this text. This idea also draws inspiration from our own problem-solving approach. Imagine a person with specific symptoms who wants to discover the cause and consequently find a cure. The person might go to a random doctor, who might make a diagnosis that might differ from another doctor's one. However, if the person compares the opinions and diagnoses of several doctors, the most common diagnosis is much more likely to be the correct one. The concept of listening to the opinions of multiple experts inspired the trading system implemented in this thesis. Developing

multiple models to perform a given task could lead to better performance. The trading system developed in this thesis is based on the use and combination of different models from different learning paradigms, specifically unsupervised learning, supervised learning, and Deep reinforcement learning. The main goal is to integrate all the models into a single system that can deliver promising performance, meaning avoiding capital losses and getting as close as possible to the benchmark. Given that bitcoin has very positive annual returns, the benchmark represents the percentage gain an investor would have had if they had bought bitcoin at a given time on the first day of the year and sold it on the last. The trading system's objective is to return the action (buy, sell, do nothing) that an investor must perform every hour, considering bitcoin as an asset. To develop the system, performance tests were conducted on the supervised models. The task was divided into several parts. The first step was to identify the metrics and models to consider, and especially the data granularity. Next, the models to be used were combined into a single system. The data granularity depends greatly on the type of system and trading being performed. In the case of the following thesis, two different granularities were used: daily and hourly. The output from the models trained on an hourly basis and those trained on a daily basis were merged into a single model that will return the action the investor must perform. To develop the system, performance tests were performed on the supervised models. The task was divided into several parts. The first step was to identify the metrics and models to consider, and especially the data granularity. This was followed by the models to be used and their integration into a single system. The granularity of the data depends greatly on the type of system and trading being conducted. In the case of the following thesis, two different granularities were used: daily and hourly. The output of the hourly-trained models and those trained on a daily basis were merged into a single model that will return the action the investor should take. The implemented system consists of five models: four of these are called experts, while the fifth model, called the master, and is responsible for unifying the model results and returning the best action for that hour. Two of the four experts were trained on a daily-granularity dataset, and two were trained on an hourly-granularity dataset. For daily-granularity models, the stock-to-flow model was used, which adopts a hybrid model (i.e., a supervised model and an unsupervised learning model) to predict the action to be taken at a given time. To address the linearity between the price and the stock-to-flow value, two models were tested: linear regression and the SVR model with a linear kernel. These models predict the price of bitcoin by considering the stock-to-flow value, but they do not provide explicit indications of the action the investor should take. To explain the action, the distance between the current price and the regression line price was calculated by taking the difference between the two values. The set of all distances produced a vector on which two clustering algorithms were trained and tested: the K-means and the GMM (Gaussian Mixture Model). Clustering allows us to identify the distances within which to buy, sell, or do nothing. This allows us to understand which action an investor should take based on a given stock-to-flow value. The second model trained on daily granularity is an SVR model with an 'RBF' kernel. It predicts the following day's price using a sliding window with a size of 7. Since the SVR model predicts the next day's price, in order to translate the price into a buy, sell, or do nothing action, a threshold was introduced to discard price changes that are too small and could degrade performance due to transaction

costs. The model returns a buy action if the next day's price is forecast to increase or decrease by a percentage greater than the threshold. If, however, the predicted price is less than the threshold, the action will be a do nothing action. The other two among four experts were trained using reinforcement learning on a daily-granularity dataset. These models use a CNN-LST neural network as the policy approximation function. One of them is called a bull agent and was trained using the years when Bitcoin performed better than the previous year. The other expert is called a bear agent and was trained on the years that performed worse than the previous year. To unify the results of all these aspects, a final expert called the master agent based on reinforcement learning was created, whose task is to analyse the market data (current observation) and the predicted results of the four previously introduced experts, and then make the final decision. This approach of using on-chain metrics and creating a multi-expert trading system that utilizes different types of models for long-term forecasting those trained on a daily scale, as well as models trained for short-term forecasting on an hourly scale, has allowed us to achieve promising results and opened the door to further potential improvements. To get a visual idea of how the components of our multi-expert system interact with each other, we created the following images:

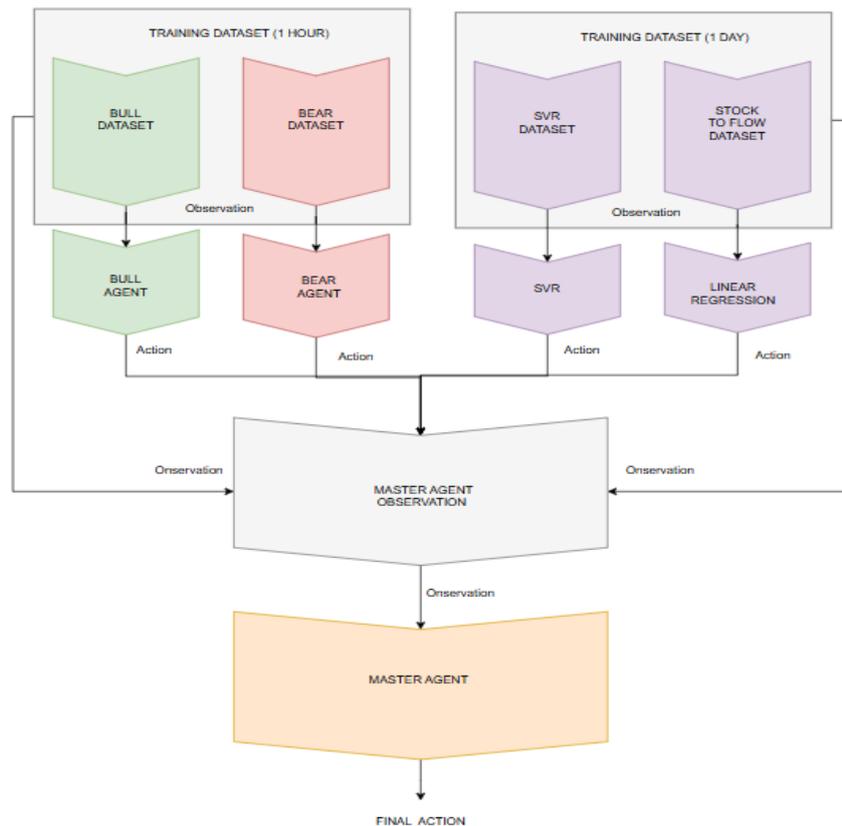


Figure 1.2: Multi-expert system

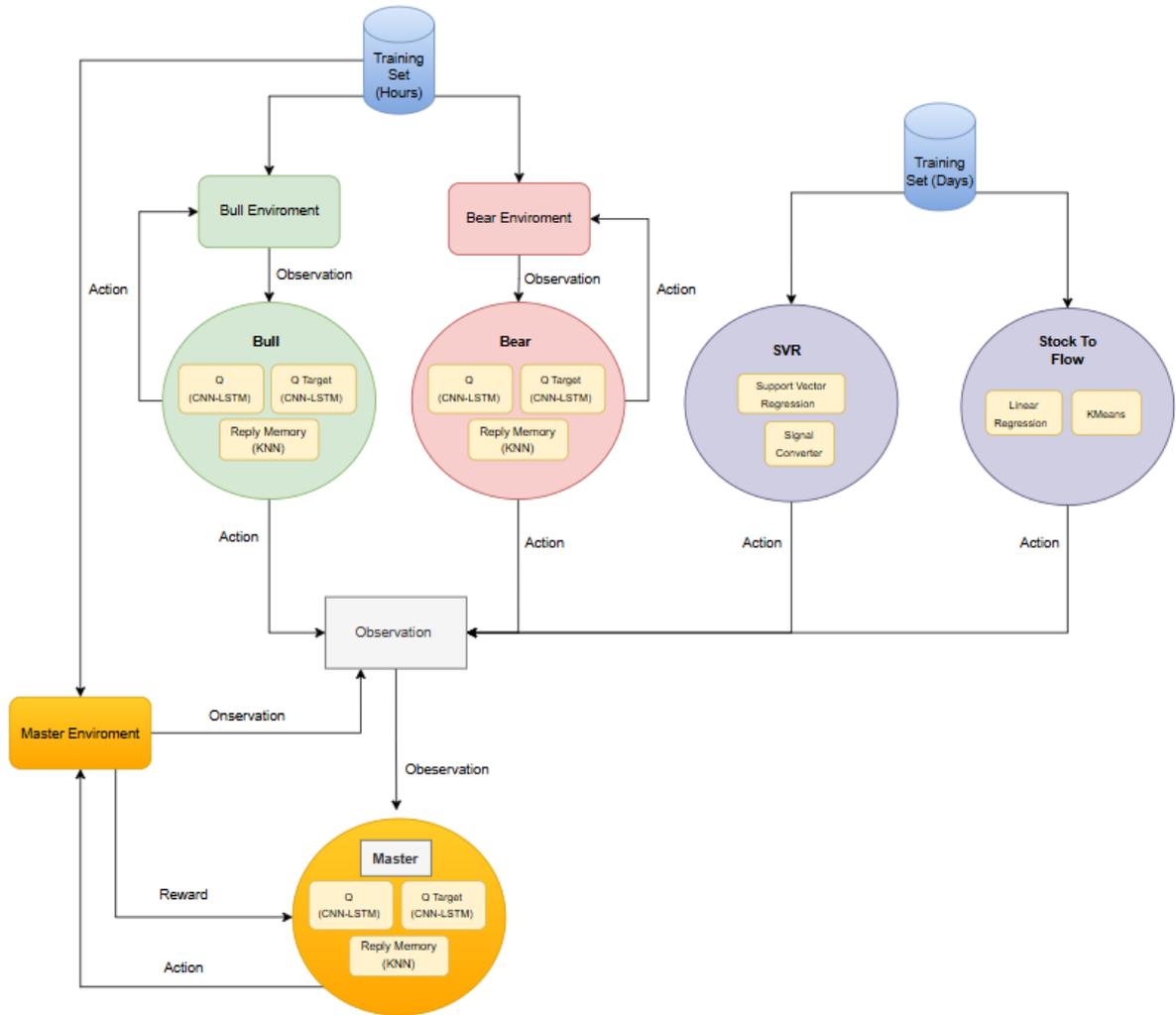


Figure 1.3: Multi-expert system

## Chapter 2

# Blockchain and Metrics

### 2.1 Bitcoin's Blockchain

The term blockchain was introduced in 2008 by Satoshi Nakamoto in the bitcoin white paper [16]. The main goal for which blockchain was introduced is to create an electronic payment system based on cryptographic proof rather than trust, which allows two willing parties to transact directly with each other without the need for a third party. To date, there are several blockchains and in the following research, the term blockchain will refer to the public bitcoin blockchain. The blockchain is nothing more than a public distributed ledger that uses a peer-to-peer architecture and complex encryption mechanisms to guarantee data integrity [3]. It is made up of blocks identified by a hash and contains transactions. Each block is concatenated to the previous block through the hash of the previous block. Transactions are at the core of the bitcoin ecosystem and all transactions end up in the memory pool, from where miners pick up transactions based on their priority to include them in the proposed block. Each transaction is composed of at least one input and output. Inputs can be thought of as coins being spent that have been created in a previous transaction and outputs as coins being created. If a transaction is to send coins to a bitcoin address, then it needs to be signed by the sender with their private key and a reference is also required to the previous transaction in order to show the origin of the coins [16]. Coins represent unspent transaction outputs (UTXO) represented in Satoshis which is the smallest unit of the cryptocurrency Bitcoin. There are many types of transactions, one the most popular is the “Pay to Public Key Hash” (P2PKH) which is used to send transactions to bitcoin addresses [3]. Another famous transaction is the coinbase or generation transaction which is always created by a miner and is the first transaction in a block and, it is used to create new coins. Transactions may have fees dependent upon the size and weight of the transaction which are charged by the miners. To solve the problem of double spending of coins, transactions are not encrypted and are publicly visible in the blockchain. Furthermore, a consensus mechanism called Proof of Work (PoW) is used to add a new block. PoW is the backbone of the blockchain and represents proof that enough computational resources have been spent in order to build a valid block. It requires finding a value that when hashed using a hashing algorithm (for example SHA256) produces a hash that starts with a certain number of zero bits [16]. The work required to find this value is exponential to the number of zero bits required, and once found, it can be verified by performing a single hash. To implement PoW, a value called “nonce” is used which is incremented until a value is found that causes the block hash to satisfy the zero bit requirement. The only way to find this nonce is the brute force method. The difficulty of the Pow is adjusted using a moving average in order to be able to produce an average number of blocks every hour. If the blocks are generated too quickly it means the computational level of the network has increased and the difficulty increases. In addition to the transactions and the nonce, each block contains the hash of the previous block, thus creating a chain of blocks from which the name derives. Each time a transaction is created it is propagated to all nodes and each node that receives it adds the new

transaction to a block. Each node works to find the proof-of-work or the hash of the block and when it finds it, it propagates the block to all the other nodes in the network. Nodes receiving the block only accept it if the transactions inside are valid and have not already been spent. Finally, the nodes that accept the received block express their consent by working to create a new block that contains the hash of the received block. Nodes always consider the longest chain as the correct one and continue working to extend it. If two nodes simultaneously transmit different versions of the next block, some nodes may receive one version before the other and in this case they work on the version they received first, but keep the other version in case this becomes the longest chain. The conflict will be resolved when the next proof-of-work is found, and one of the two chains becomes longer and finally the nodes that were working on the other chain will switch to the longer one. The process of adding new blocks to the blockchain is called mining and it consumes a lot of electricity and requires a lot of computing power. The nodes commonly called miners have the objective of synchronizing with the network, validating block transactions and above all creating new blocks by solving the PoW [16]. To motivate miners to support the blockchain they receive a reward for each block mined, in fact the first transaction of a block creates a new coin which is assigned to the creator of the block together with the transaction fees. This system, in addition to being an incentive for the nodes, also represents the method of issuing coins. On average every 10 minutes a new block is added to the blockchain, every 210,000 blocks the production is halved or approximately every 4 years, this phase is called Halving and it is precisely this that gives bitcoin the rarity effect. Bitcoin production is limited and it is estimated that the limit of 21 million bitcoins will be reached in 2140, beyond which they will no longer be produced [3].

## 2.2 On-Chain Metrics

Blockchains record several types of data like transactions, blocks and wallet addresses that are publicly available and can be useful for making market forecasting. Transactions store information about the sender, receiver and the amount that was moved while blocks contain information about the miners reward, the difficulty for mining the block and the number of transactions. Transactions are at the core of the bitcoin ecosystem and each transaction is composed of at least one input and one output, except for the coinbase transaction. The transaction inputs can be thought of as coins being spent that have been created in a previous transaction while outputs as coins being created. In order to keep track of the coin ownership the bitcoin protocol uses a base unit which is the unspent transaction output “UTXO”. If an UTXO is used as input for a transaction it is destroyed and generally two new UTXO are created, one for the fees which will be owned by the miner and one for the recipient of the transaction. The UTXO represents one of the fundamental building blocks of on-chain analysis because it’s possible to find important information about investor behavior based on the history of UTXOs and by studying the time and price at which they are created, and destroyed.

### **2.2.1 MVRV (Market value to Realized value)**

The MVRV is an indicator computer as the ratio between the market cap and the Realized cap. The Market cap represents the value of an asset and it is calculated by multiplying the total number of coins by the price. On the other hand the realized cap is a variant of the market cap which is computed by evaluating each UTXO with the price when they were last moved. The realized price is increasing when coins last moved at a cheap price are spent and will decrease when coins last moved at a expensive price are spent. UTXOs can belong to long or short-term investors, taking into consideration the number of days for which a given UTXO has been kept we can distinguish between short and long investors and consequently it is possible to create two MVRV type indicators that track their behaviour. The STH-MVR indicator is calculated taking into account UTXOs that have a duration of less than 155 days, while the LTH-MVRV takes into account UTXOs with a duration greater than 155 days.

### **2.2.2 Hash Ribbon**

Miners contribute to the maintenance of the blockchain by validating transactions and adding new blocks, these actions requiring computational resources and electricity. A miner's computing power is measured in the number of hashes per second. The Hashrate is a metric that measures the number of SHA256 calculations that all the miners on the network are able to do per second. This value is not constant but changes over time based on the computing power of the miners and above all based on the difficulty of the mining sector. Mining difficulty is a parameter that is recalculated by the bitcoin protocol every 2016 blocks (approximately every 2 weeks) to ensure an average block construction speed of approximately one block every 10 minutes. The big downward trend of the price can significantly reduce the hashrate of the network as miners tend to stop mining when the coin loses value and it is not profitable to mine it. The hash ribbon is an indicator that takes into consideration the hashrate and the difficulty to create an SMA (simple moving average) for each of them, when the SMA of the mining difficulty crosses with the SMA of the hashrate this means that miners are capitulating and we are close to a market bottom.

### **2.2.3 Coin Days Destroyed and Liveness**

The CDD is an indicator that allows us to track the behaviors and interventions of entities that are in possession of coins kept for a very long period and also of those that have a large availability of coins. The CDD uses the concept of "money day", for every day that a coin has not been spent, it accumulates one day and when it is spent or rather destroyed the days are reset to zero. This indicator tracks only coins that are government expenditures over a given period of time and is calculated by multiplying the value of the coin by the number of days it has been held. A very high value indicates that long-term investors are spending the coins and is related to a market low or top. Liveness is a metric that highlights whether long-term holders are accumulating or spending and it is based on CDD, it is calculated as the ratio of CDD to the sum of all coins created up until then.

#### **2.2.4 Net Unrealized Profit/Loss (NULP)**

The realized profit and loss represents a fundamental piece in on chain analysis, it is based on the study of the price of UTXOs at the moment of creation and destruction. By calculating the difference between the price at the time of creation of the utxo and the current price we can establish the profit and loss. Coins acquired at a lower price than the current price have a certain unrealized profit while coins that were acquired at a higher price have an unrealized loss. The NULP is calculated by subtracting the realized cap from the market cap and then dividing everything by the realized cap. As with the MVRV, it is also possible to distinguish between long term holders and short term holders for the NULP. All UTXOs with a life exceeding 155 days are considered long term holders, while those with a shorter duration are considered short term holders.

## Chapter 3

# Mid and Long Term Forecasting

### 3.1 Stock to flow

The stock to flow model has been widely used in the literature as a reference model to evaluate the scarcity of a given asset. Various approaches have been proposed to study the relationship between stock-to-flow and the price of bitcoin. Among the most well-known is the study conducted by Andrea Pontiggia and Giovanni Fasano in the paper [9], which introduces a machine learning paradigm that combines classification, ranking, and sorting to analyze systems. Another study conducted by Giovanni Fasano and Marco Corazza on the bitcoin price, in which they show the power of linear regression and SVMs is the following one [9].

The stock represents the total flow existing to date while the flow represents the flow produced in a specific time interval, generally it is calculated on an annual basis. A high stock flow value indicates that the asset has a higher scarcity and therefore a higher value as it takes much more time to reproduce the current flow. While a low value indicates that the asset is produced more quickly and this is attributed to a lower value:

$$SF = \frac{Stock}{Flow} = \frac{TotalSupply}{AnnualFlow}$$

By analysing the tokenomics of bitcoin we know that a block is produced on average every 10 minutes and that every 210,000 blocks the reward given to miners is halved, this happens approximately every four years. The halving of the reward given to the miners, makes bitcoin an asset with a scarcity effect because as the time passes the number of the new mined bitcoins will decrease and this causes its value to grow over time .

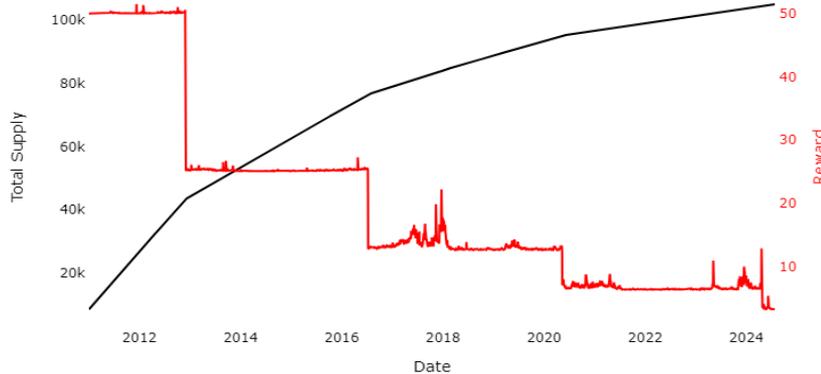


Figure 3.1: Miner Reward and Total Supply

The reward and the number of blocks can be calculated using the information from the blockchain tokenomics, however to have more precise data for the construction of the stock to flow model we obtained the data using Glassnode and blockchair API, a provider that provides APIs for receive information about blocks, addresses and transactions of different cryptocurrencies. The total supply was calculated by adding the reward obtained day after day while the flow was calculated on an annual basis. The charts below shows a strong correlation between Bitcoin's price and the Stock-to-Flow (S2F) ratio. A correlation coefficient of 0.89 it's almost perfect, this indicates that the scarcity has a relevant influence over the market value.

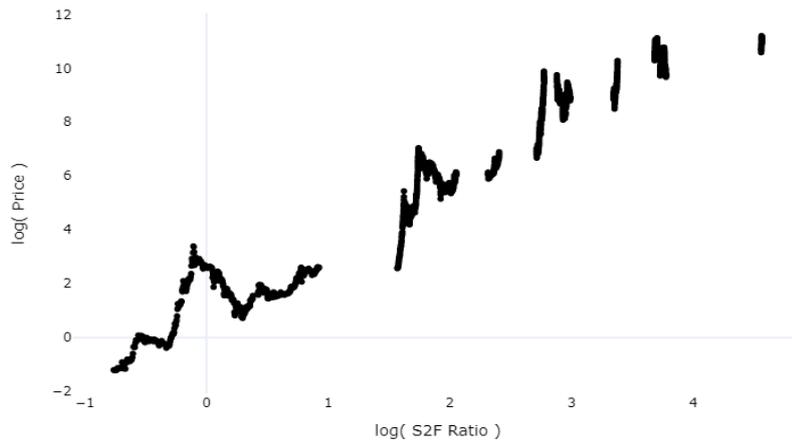


Figure 3.2: log(Stock To Flow Ration) and log (Price)

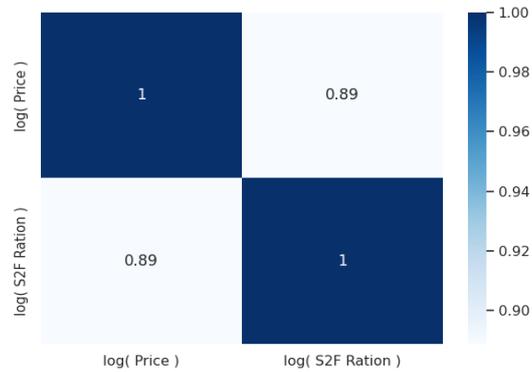


Figure 3.3: log(Stock To Flow Ration) and log (Price) Correlation

### 3.1.1 Dataset

The dataset was created by combining different sources, in particular to obtain the number of blocks and for the reward we used blockchain, a website that provides APIs to extract information on blocks and transactions, while for the closing price of bitcoin, we downloaded the data in csv format directly from the glassnode website. The range from ‘2011-01-01’ to ‘2025-08-01’ was considered as dataset to be divided into train and test. The dataset  $D$  can be represented as follow:

$$D = \{(\text{date}_i, \text{year}_i, \text{close\_price}_i, \text{reward}_i, \text{blocks}_i) \mid i = 2011 - 01, \dots, 2024 - 07\}$$

Where:

- $\text{date}_i$  represents the  $i$ -th day.
- $\text{year}_i$  represents the year extracted from  $\text{date}_i$ .
- $\text{close\_price}_i$  is the closing price on day  $i$ .
- $\text{reward}_i$  represents the average reward given to miners on day  $i$ .
- $\text{blocks}_i$  represents the number of blocks mined on day  $i$ .

The stock to flow ratio was computed as follow:

$$R_y = \sum_{i \in \{j: \text{year}_j = y\}} \text{reward}_i$$

$$\text{total\_supply}_i = \sum_{j=1}^i \text{reward}_j$$

$$\text{annual\_flow}_i = R_{\text{year}_i} \quad \forall i \in \{2011, \dots, 2024\}$$

$$\text{SF}_i = \frac{\text{total\_supply}_i}{\text{annual\_flow}_i}$$

### 3.2 Regression and Least Square Optimization

Linear regression is a predictive model that is used to establish the relationship between a dependent variable and one or more independent variable by using the following equation:

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p + u$$

this formula can be write in a more compact way as follow:

$$y = \sum_{i=1}^p \beta_i X_i + u_i, \quad \beta_i \in \mathbb{R}, i = 1, \dots, p$$

where:

- $y$  is the dependent variable,
- $X_i$  are the independent variables,

- $u_i$  is the statistical error.

If the variables  $Y, X_1, \dots, X_p$  are independent and identically distributed, then least square model can be used for estimating the coefficients  $\beta_0, \beta_1, \dots, \beta_p$  by solving the following linear least squares problem:

$$\min \sum_{i=1}^p (y_i - \hat{y}_i)^2, i = 1, \dots, p$$

where

$$\hat{y} = \sum_{i=1}^p \beta_i X_i + u_i, \quad \beta_i \in \mathbb{R}, i = 1, \dots, p$$

Considering the correlation between price and stock-flow value, linear regression appears to be a promising tool capable of capturing the relationship between stock-flow and price. However this model have theoretical assumptions like the normality distribution of the sample which may not fulfilled. An alternative machine learning model that can be used in order to find the linear relationship between the variable is the SVR which uses the same principles as SVM but focuses on predicting continuous outputs rather than classifying data points.

### 3.3 Support Vector Machines (SVM)

SVM is a supervised learning algorithms widely used for classification, regression, and outlier detection. The core idea behind of SVM is to find an optimal decision boundary, called a hyperplane, that separates data belonging to different classes with the maximum possible margin. The closest data points to the hyperplane are called support vectors [9].

Consider a binary classification dataset:

$$\{(x_i, y_i)\}_{i=1}^m, \quad x_i \in \mathbb{R}^n, y_i \in \{-1, +1\},$$

where  $x_i$  represents feature vectors and  $y_i$  represents class labels. A hyperplane in  $\mathbb{R}^n$  is defined as:

$$w^T \cdot x + b = 0,$$

where  $w \in \mathbb{R}^n$  is the weight vector and  $b \in \mathbb{R}$  is the bias term.

The goal of SVM is to find  $w$  and  $b$  such that the separating hyperplane maximizes the margin, that is the distance between the hyperplane and the nearest training points named support vectors [9]. For linearly separable data, the constraints are:

$$y_i(w^T \cdot x_i + b) \geq 1, \quad \forall i.$$

#### 3.3.1 Optimization Problem

Maximizing the margin is equivalent to minimizing  $\|w\|^2$ . Thus, the primal optimization problem can be written as follows:

$$\min_{w,b} \frac{1}{2} \|w\|^2 \quad \text{subject to } y_i(w^T \cdot x_i + b) \geq 1. \quad \forall i.$$

### 3.3.2 Soft Margin SVM

In real-world applications, the perfect linear separation could sometimes not be achieved due to outliers. In order to allow misclassification, the SMV model uses slack variables  $\xi_i \geq 0$  as follows:

$$y_i(w^T \cdot x_i + b) \geq 1 - \xi_i.$$

The optimization problem becomes the following.

$$\min_{w,b,\xi} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m \xi_i,$$

where  $C > 0$  can be viewed as a threshold that controls the trade-off between maximizing the margin and minimizing classification errors.

### 3.3.3 Dual Formulation and Support Vectors

The optimization problem is usually solved in its dual form using Lagrange multipliers  $\lambda_i \geq 0$ :

$$\max_{\lambda} \sum_{i=1}^m \lambda_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \lambda_i \lambda_j y_i y_j (x_i \cdot x_j),$$

subject to:

$$\sum_{i=1}^m \lambda_i y_i = 0, \quad 0 \leq \lambda_i \leq C.$$

Only data points with  $\lambda_i > 0$  influence the final decision boundary. The linear classifier function can be written as follow:

$$f(x) = \text{sign} \left( \sum_{i=1}^m \lambda_i y_i (x_i \cdot x) + b \right).$$

### 3.3.4 Kernel Trick

When data is not linearly separable in the input space, SVM can map it into a higher-dimensional feature space using a nonlinear transformation  $\phi(x)$ . Instead of explicitly computing  $\phi(x)$ , SVM employs a kernel function  $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$ .

Common kernel functions:

- **Linear kernel:**  $K(x, x^T) = x \cdot x^T$ ,
- **Polynomial kernel:**  $K(x, x^T) = (x \cdot x^T + c)^d$ ,
- **Radial Basis Function (RBF):**  $K(x, x^T) = \exp(-\gamma \|x - x^T\|^2)$ ,
- **Sigmoid kernel:**  $K(x, x^T) = \tanh(\kappa x \cdot x^T + c)$ .

This allows SVM to create highly flexible nonlinear decision boundaries while still relying only on support vectors.

### 3.3.5 Geometric Interpretation

- The **hyperplane** is the decision boundary between classes.
- The **margin** is the distance between the hyperplane and the nearest support vectors.
- The **support vectors** are the critical data points that determine the position of the hyperplane.
- Increasing the margin improves generalization, while allowing slack variables increases robustness to noise.

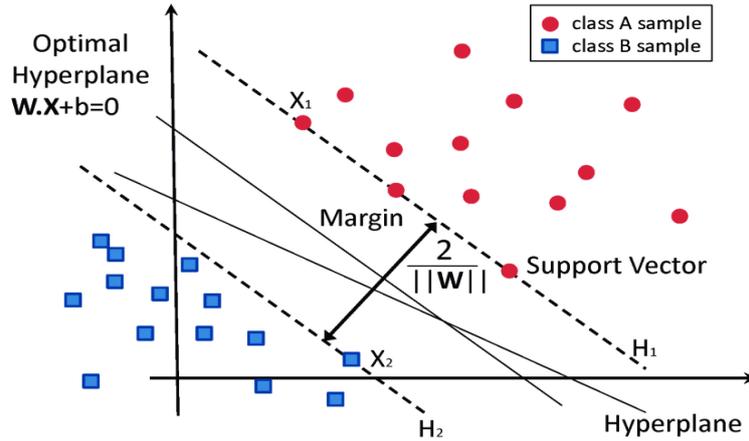


Figure 3.4: Geometric Interpretation

### 3.3.6 Support Vector Regression (SVR)

SVR is an extension of SVM applied to the regression tasks, where the goal is to predict continuous values instead of class labels.

Instead of maximizing the margin, which is the distance between the hyperplane and the support vectors, SVR attempts to fit a regression function within a tolerance margin  $\epsilon$ . Points within the  $\epsilon$ -tube are ignored in the loss function, while points outside contribute to the error [2].

Given training data  $\{(x_i, y_i)\}_{i=1}^m$ , the SVR optimization problem is:

$$\min_{w, b, \xi_i, \xi_i^*} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m (\xi_i + \xi_i^*)$$

subject to:

$$\begin{aligned} y_i - (w \cdot x_i + b) &\leq \epsilon + \xi_i, \\ (w \cdot x_i + b) - y_i &\leq \epsilon + \xi_i^*, \\ \xi_i, \xi_i^* &\geq 0. \end{aligned}$$

Here:

- $\epsilon$  is the tolerance margin (insensitive zone),
- $\xi_i, \xi_i^*$  are slack variables for deviations outside the margin,
- $C$  is the threshold which controlling the trade-off.

### 3.3.7 Evaluation Metrics

Metric	Formula
$R^2$	$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$
MSE	$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
MAE	$MAE = \frac{1}{n} \sum_{i=1}^n  y_i - \hat{y}_i $

Table 3.1: Evaluation Metrics

Where:

- $n$  is the number of observations.
- $y_i$  is the actual value.
- $\hat{y}_i$  is the predicted value.
- $\bar{y}$  is the mean of the actual values.

### 3.3.8 Models Result

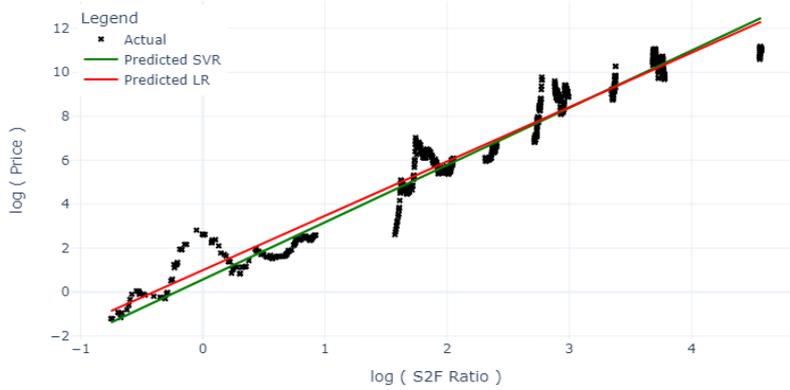


Figure 3.5: Long Term Forecasting

As we can see from the evaluation metrics both models have demonstrated an excellent performance in identifying the long-term trend of the bitcoin price and confirm again that the long-term bitcoin price follows the stock to flow model. They allow us to clearly identify whether the current value of bitcoin is undervalued or overvalued. Although the performance of the linear regression model is slightly better, we should take into account that it has strong assumptions like the fact that residuals (the differences between observed and predicted values) should follow a normal distribution.

Linear regression provides clear indications of long-term performance, but to provide daily signals (buy, sell, or do nothing) to a specific investor, the model must be expanded. The approach used to determine which stock to execute on a given day

Model	$R^2$	MSE	MAE
Linear Regression	0.948360	0.479768	0.564715
SVR	0.945056	0.510462	0.551536

Table 3.2: Performance Metrics

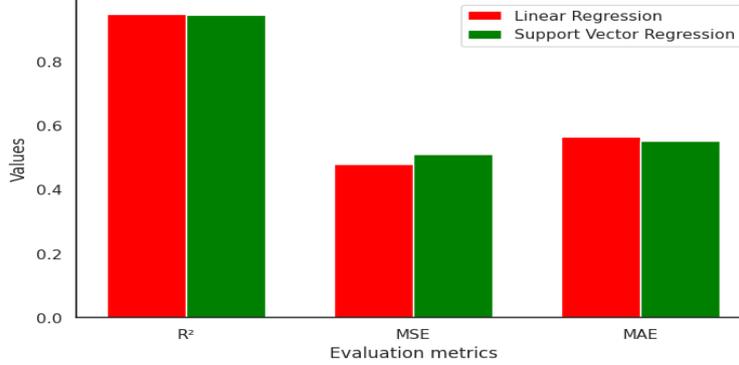


Figure 3.6: Performance Metrics Histogram

is to calculate the distance between the price predicted by linear regression and the actual price. Once the distances are calculated, we obtain a distance vector on which we apply a clustering algorithm to identify the best regions for buying, selling, or doing nothing. Several clustering algorithms exist in the literature; for this analysis, two promising algorithms were tested: Kmeans and the Gaussian Mixture Model. Furthermore, a mathematical optimization model was developed that aims to solve both the classification and regression problems. In the following pages there is a detailed description of the clustering algorithms used.

### 3.4 K-means Clustering

K-means is an unsupervised machine learning algorithm used for clustering data into  $k$  groups. Its primary goal is to partition  $n$  data points into  $k$  distinct and separate clusters such that the within-cluster variance (inertia) is minimized.

Given a dataset  $\{x_1, x_2, \dots, x_n\}$  where  $x_i \in \mathbb{R}^d$ , the goal of K-means is to split the data into  $k$  groups named clusters  $C_1, C_2, \dots, C_k$  by finding cluster centroids  $\{\mu_1, \mu_2, \dots, \mu_k\}$  which represents the most significant point of the cluster.

The objective function to minimize is the total within-cluster sum of squared distances:

$$J = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|^2,$$

where:

- $C_i$  is the set of points assigned to cluster  $i$ ,
- $\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$  is the centroid of cluster  $i$ ,
- $\|x - \mu_i\|^2$  is the squared Euclidean distance between point  $x$  and centroid  $\mu_i$ .

### 3.4.1 Algorithm Description

The K-means algorithm works iteratively in two main steps:

1. Choose  $k$  initial centroids  $\{\mu_1, \dots, \mu_k\}$  randomly
2. Assign each data point  $x_j$  to the nearest centroid:

$$C_i = \{x_j : \|x_j - \mu_i\|^2 \leq \|x_j - \mu_l\|^2, \forall l \in \{1, \dots, k\}\}.$$

3. Recompute each centroid as the mean of points in the cluster:

$$\mu_i = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j.$$

4. Alternate between assignment and update until there is no change in assignments or the change of centroids is minimal.

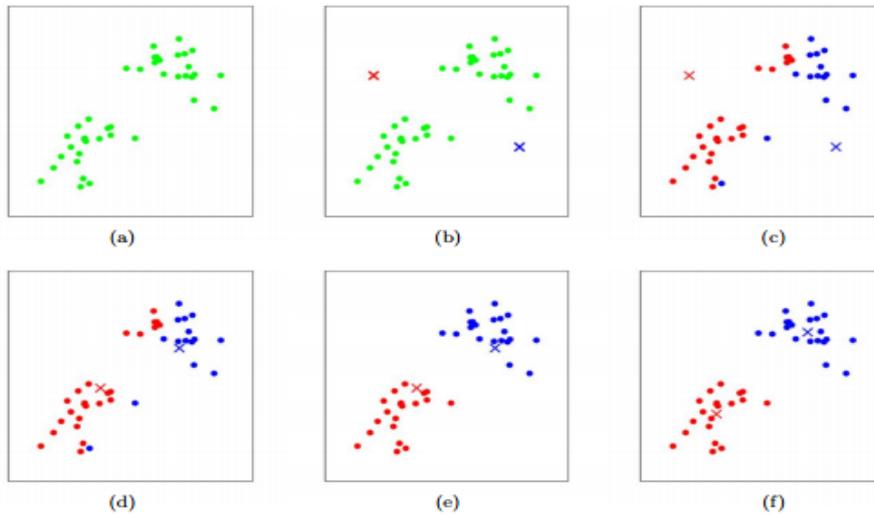


Figure 3.7: Evolution of K-means centroid assignments over successive iterations

### 3.4.2 Convergence and complexity

K-means is guaranteed to converge to a local minimum in a finite number of steps because the objective function  $J$  decreases monotonically with each iteration. However:

The computational complexity of K-means per iteration is:

$$\mathcal{O}(n \cdot k \cdot d),$$

where  $n$  is the number of data points,  $k$  is the number of clusters, and  $d$  is the dimensionality. It is generally efficient for medium-to-large datasets, though it may struggle with very high-dimensional or large-scale data without optimization.

K-means is widely used in image segmentation, document clustering, text mining, and other machine learning tasks. Even if the algorithm has several limitations like the sensitivity to the choice of  $k$  (number of clusters), the outliers and the assumption of spherical clusters, it represented one of the most used clustering algorithms.

### 3.5 Gaussian Mixture Models (GMM)

The Gaussian Mixture Model (GMM) is a probabilistic model used for clustering, density estimation, and pattern recognition. Unlike K-means, which partitions data into hard clusters, GMM assume that the data is generated from a mixture of several Gaussian (normal) distributions and provide a soft clustering approach where each data point has a probability of belonging to each cluster.

#### 3.5.1 Model Definition

A Gaussian Mixture Model assumes that the probability density function of data  $x \in \mathbb{R}^d$  is a weighted sum of  $k$  Gaussian components:

$$p(x | \Theta) = \sum_{i=1}^k \pi_i \mathcal{N}(x | \mu_i, \Sigma_i),$$

where:

- $\pi_i \geq 0$  are the mixture weights, with  $\sum_{i=1}^k \pi_i = 1$ ,
- $\mu_i \in \mathbb{R}^d$  is the mean of the  $i$ -th Gaussian,
- $\Sigma_i \in \mathbb{R}^{d \times d}$  is the covariance matrix of the  $i$ -th Gaussian,
- $\mathcal{N}(x | \mu_i, \Sigma_i)$  is the multivariate Gaussian density:

$$\mathcal{N}(x | \mu_i, \Sigma_i) = \frac{1}{(2\pi)^{d/2} |\Sigma_i|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_i)^\top \Sigma_i^{-1} (x - \mu_i)\right).$$

The parameter set of the model is:

$$\Theta = \{\pi_i, \mu_i, \Sigma_i\}_{i=1}^k.$$

#### 3.5.2 Learning via Expectation-Maximization (EM)

Since cluster memberships are latent variables, GMMs are typically estimated using the Expectation-Maximization (EM) algorithm.

1. Choose initial random values for  $\pi_i$ ,  $\mu_i$ , and  $\Sigma_i$
2. **E-step (Expectation).** Compute the posterior probability that point  $x_j$  belongs to cluster  $i$ :

$$\gamma_{ij} = \frac{\pi_i \mathcal{N}(x_j | \mu_i, \Sigma_i)}{\sum_{l=1}^k \pi_l \mathcal{N}(x_j | \mu_l, \Sigma_l)}.$$

3. **M-step (Maximization).** Update the parameters using the probabilities:

$$N_i = \sum_{j=1}^n \gamma_{ij},$$
$$\pi_i = \frac{N_i}{n}, \quad \mu_i = \frac{1}{N_i} \sum_{j=1}^n \gamma_{ij} x_j, \quad \Sigma_i = \frac{1}{N_i} \sum_{j=1}^n \gamma_{ij} (x_j - \mu_i)(x_j - \mu_i)^\top.$$

4. Repeat E-step and M-step until the log-likelihood:

$$\log L(\Theta) = \sum_{j=1}^n \log \left( \sum_{i=1}^k \pi_i \mathcal{N}(x_j \mid \mu_i, \Sigma_i) \right)$$

converges or changes below a threshold.

### 3.6 Identifying Stock-to-Flow Signals via Unsupervised Learning

As previously mentioned, linear regression provides clear indications of long-term performance, but to provide daily signals (buy, sell, or do nothing) to a specific investor, the model must be expanded using unsupervised learning. The approach used to determine which stock to execute on a given day is to calculate the distance between the price predicted by linear regression and the actual price. Once the distances are calculated, we obtain a distance vector on which we apply a clustering algorithm to identify the best regions to buy, sell, or do nothing. For the following thesis, two very popular algorithms in the literature were used: KMeans and the Gaussian Mixture model. Both models were able to identify the peak and trough points that represent the right time to buy, sell, or do nothing.

#### 3.6.1 Model Training

The implemented stock-to-flow model involves training two algorithms: linear regression and KMeans. Regression takes as input a dataset consisting of stock-to-flow and price, both transformed to a logarithmic scale. Once the model is trained, we have the actual price values and the value predicted by the model, and with these two we can calculate the distance vector that will be used by KMeans ( $k = 3$ ). The distance vector was transformed using the standard scaler, a data preprocessing tool used to standardize features. The Standard Scaler transforms the data by subtracting the mean and dividing by the standard deviation, so each feature has a mean of 0 and a deviation of 1. This makes the data comparable and facilitates scale-sensitive algorithms, such as KMeans or regression. Once we have trained KMeans on the distance vector, we can use the cluster centers to translate the predicted value into an actual signal by ordering the cluster centers in ascending order and assigning the buy action to the first cluster center, the do nothing action to the central one, and the sell action to the last cluster center.

#### 3.6.2 Model Training Steps

Given the training data  $x_{\text{train}}$  and  $y_{\text{real}}$ , the signal predictor is trained as follows:

1. **Fit Linear Regression Model:** Train a linear regression model on  $x_{\text{train}}$  to predict  $y_{\text{real}}$ .
2. **Compute Predictions:** Use the trained model to calculate predictions  $\hat{y} = y_{\text{pred}}$ .
3. **Calculate Distance:** Compute the prediction error (distance) for each sample:

$$\text{distance}_i = y_{\text{real},i} - y_{\text{pred},i}$$

4. **Create DataFrame:** Construct a DataFrame containing  $x$ ,  $y_{\text{real}}$ ,  $y_{\text{pred}}$ , and distance.
5. **Standardize Distances:** Scale the distance column using StandardScaler [15]:

$$\text{distance}_{\text{scaled}} = \frac{\text{distance} - \mu}{\sigma}$$

6. **KMeans Clustering:** Apply KMeans with 3 clusters on the scaled distances and assign each sample to a cluster.
7. **Map Clusters to Signals:** Compute cluster centers in the original scale, sort them from smallest to largest, and assign: smallest  $\rightarrow$  Buy (1), middle  $\rightarrow$  Hold (0), largest  $\rightarrow$  Sell (2).
8. **Assign Signals:** Add the signal column to the DataFrame according to cluster membership.
9. **Return Result**

### 3.6.3 Model Performance

The clustering algorithm's structure is of fundamental importance as it determines the signal the investor must execute. K-Means assumes that each point belongs to only one cluster and that the implicit shape of the clusters is spherical. GMM, on the other hand, assumes that the data comes from a combination of Gaussian distributions and that each point has a probability of belonging to each cluster. The silhouette coefficient was used to measure the algorithm's performance and find clusters. The silhouette coefficient measures how well each point is grouped with respect to its cluster and other clusters and its computed as follows.

For a point  $i$ :

$$a(i) = \frac{1}{|C_i| - 1} \sum_{\substack{j \in C_i \\ j \neq i}} d(i, j)$$

where  $C_i$  is the cluster containing  $i$ , and  $d(i, j)$  is the distance between points  $i$  and  $j$ .

$$b(i) = \min_{C \neq C_i} \frac{1}{|C|} \sum_{j \in C} d(i, j)$$

that is, the smallest average distance between  $i$  and any other cluster  $C \neq C_i$ . The silhouette coefficient is then defined as:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

The coefficient takes into account internal cohesion and external separation. Internal cohesion measures how close a point is to other points in its own cluster, while external separation measures how far a point is from the points in the closest cluster other than its own.

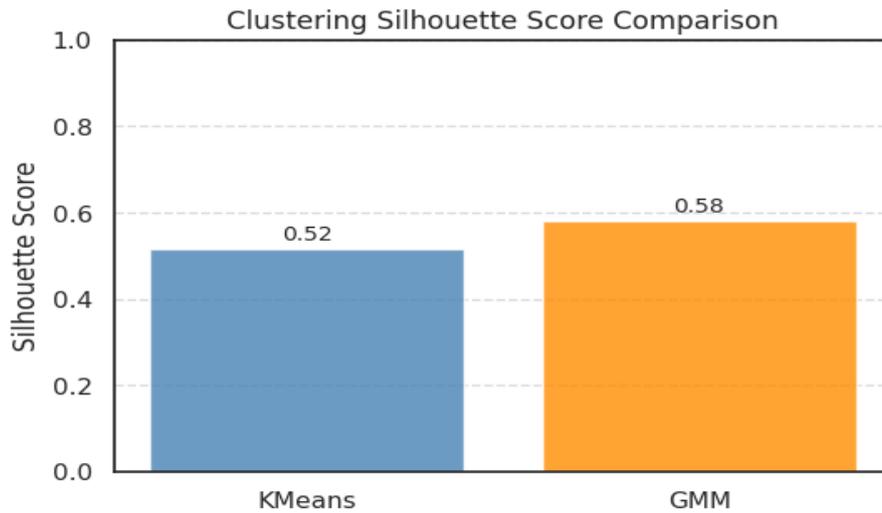


Figure 3.8: Silhouette coefficient comparison

The value of the silhouette coefficient ranges from -1 to 1. The closer the value is to 1, the better the assignment of points to the clusters. If the value is negative and close to -1, it is very likely that the points are assigned to the wrong cluster. As we can see in the image below, the two algorithms have very similar coefficients; GMM has a slightly higher coefficient. However, KMeans allows us to better identify the peak points of the Bitcoin price.

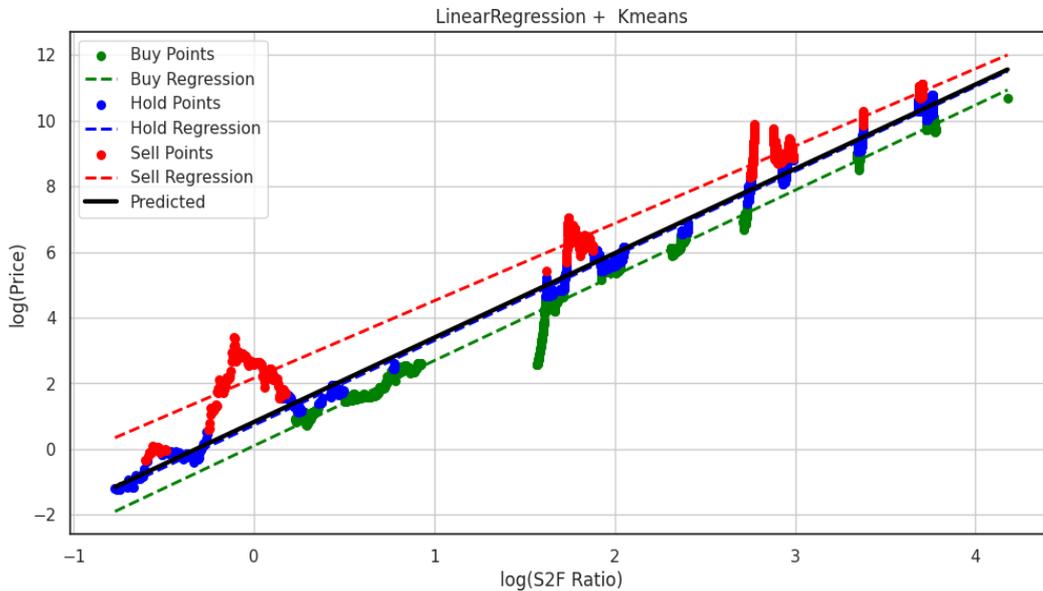


Figure 3.9: Linear Regression and KMeans

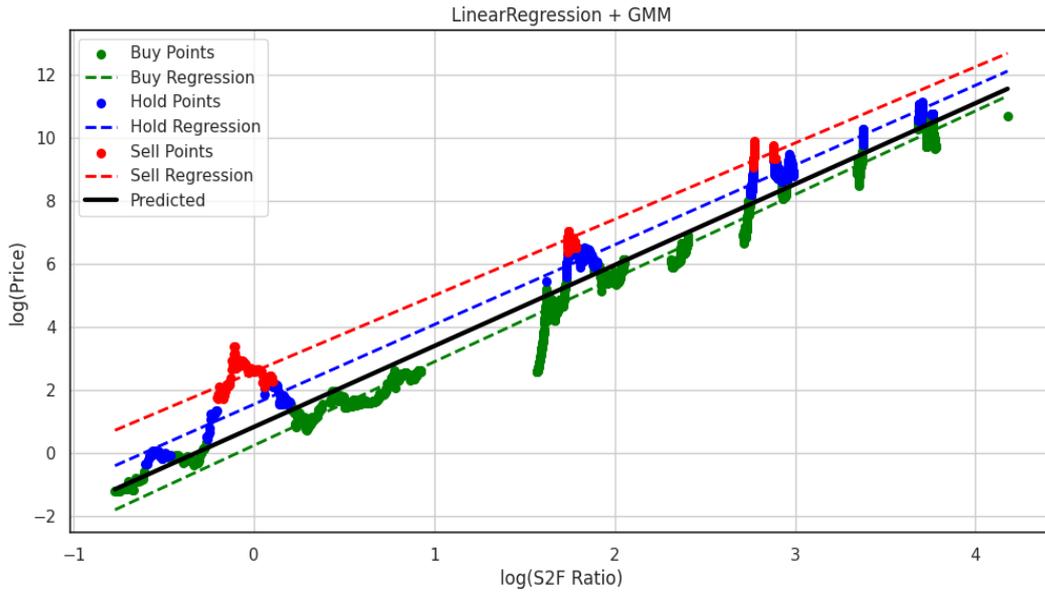


Figure 3.10: Linear Regression and Gaussian Mixture Model

The two graphs above show the linear relationship between the stock-to-flow model and the price of bitcoin in USD. Furthermore, the two clustering algorithms discussed above, KMeans and the Gaussian Mixture Model, were used for the two graphs. As we can see, clustering algorithms are able to find groups of peaks and troughs. However, despite having a slightly lower silhouette coefficient than the GMM, KMeans is able to better identify peaks. The graph below shows the daily price of bitcoin, and for each day, the action the investor must perform is shown. Specifically, green was used to identify the days on which the investor must buy, blue for the days on which no action must be performed, and red to identify the days on which the investor must sell.

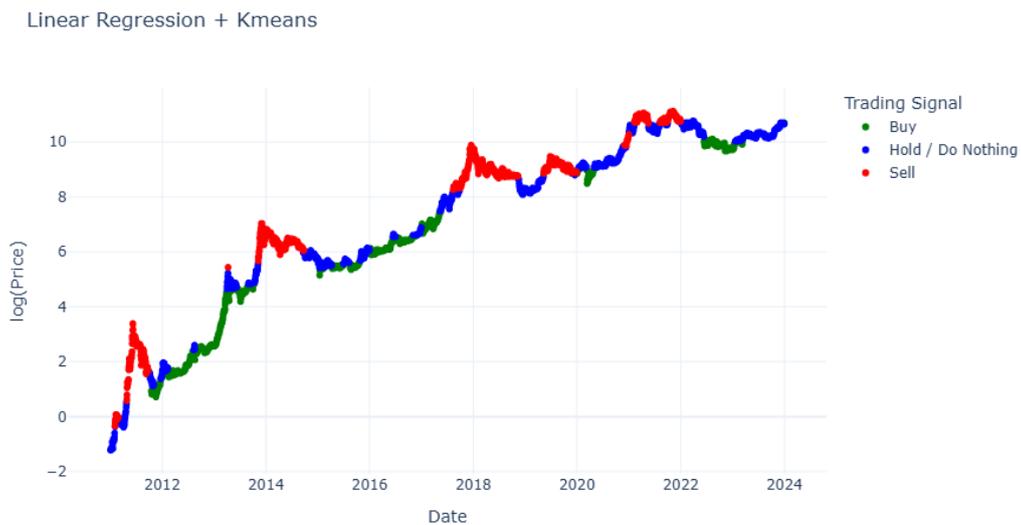


Figure 3.11: Stock To Flow Train Prediction

### 3.7 SVR-Based Approach to Next-Day Trading Decisions”

The second implemented model is based on the use of Support Vector Regression (SVR) which is a supervised learning algorithm that represents an extension of SVM applied to the regression tasks, where the goal is to predict continuous values instead of class labels. Instead of maximizing the margin, which is the distance between the hyperplane and the support vectors, SVR attempts to fit a regression function within a tolerance margin. Since the task is to predict the next day’s price by analysing previous days, it was decided to use the ‘rbf’ kernel because the price tends to follow a non-linear trend in the short term. The kernel is calculated as follows:

$$K(x, x') = \exp(-\gamma\|x - x'\|^2)$$

The dataset used for forecasting was created from daily data downloaded from Glassnode. Each indicator, or rather metric, was downloaded and saved in a JSON file. To merge the different metrics, it was necessary to create a custom Dataset-Loader. This aims to merge all the data, allow for normalization, and ultimately produce a clean dataset that will be used for forecasting. The dataset consists of basic information such as the day, price, on-chain metrics, stock-to-flow value, and some technical indicators, which are summarized in the following table:

Table 3.3: Dataset

Category	Metrics
<b>Base Info</b>	Timestamps (date_time) Price (price) Stock-to-Flow
<b>Technical Indicators</b>	Moving averages (fast, slow) RSI values Technical signal (buy/sell/neutral)
<b>Hash Ribbon Signals</b>	Buy / Capitulation / Crossed signals 30-day and 60-day hash rate moving averages
<b>On-chain Metrics</b>	MVRV Z-Score Long-term and Short-term holder supply Delta cap and Realized cap
<b>Address Counts</b>	Raw counts (wallets $\geq 0.01$ , $\geq 1$ , $\geq 10$ , etc.) Derived group counts (plankton, shrimp, crab, fish, shark, whale, mega-whale)
<b>Address Count Dynamics</b>	Changes in wallet count over 7, 14, and 30 days

Given that the data has very different scales, it was necessary to use mechanisms to homogenize the scales as much as possible. To this end, three different forms of normalization were tested, on which the algorithm was trained and its performance was then verified. The normalization methods used are the following:

Table 3.4: Data Normalization Methods

Method	Description / Formula
<b>Z-score (Standardization)</b>	$z = \frac{x - \mu}{\sigma}$
<b>Min-Max Scaling</b>	$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$
<b>Log Transformation</b>	$x' = \log(x + c)$

As we can see from the image below, the transformation with which the svr model obtained the lowest error is the logarithmic transformation. This is the best choice for the model as it has an R2 very close to one and the lowest MSE and MAE compared to the z-score and the min-max scaler.



Figure 3.12: Comparison of SVR Errors Across Dataset Transformations

In addition to choosing the normalization or, more specifically, transformation strategy, it is crucial to identify the size of the sliding window the algorithm should use to predict the next day. In fact, for this very reason, the model's performance was iteratively tested with sliding windows of varying sizes. As can be seen in the graph below, the larger the sliding window size, the worse the performance. For this reason, we decided to use a slide window of size 7, as it exhibits the best performance.

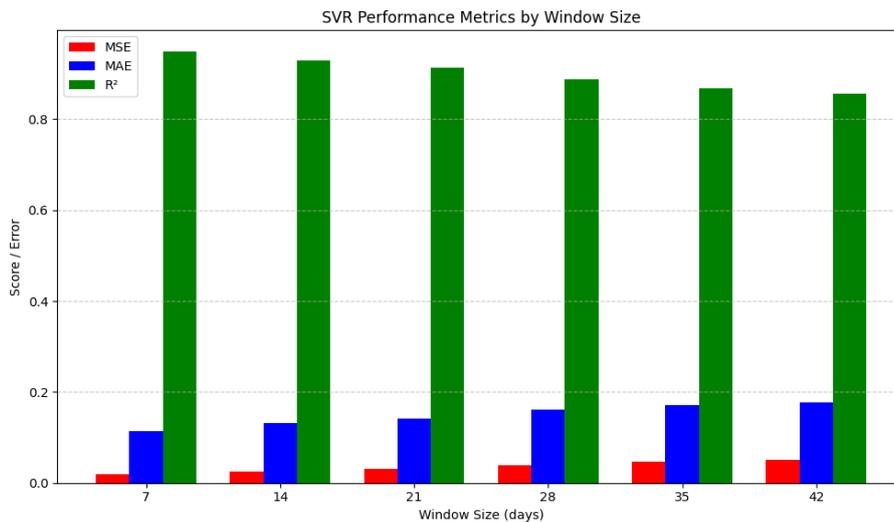


Figure 3.13: Impact of Window Size on SVR Error

The SVR model predicts the next day's price, and since the primary objective is to provide explicit guidance on the actions the investor should take (buy, sell, and do nothing), it was necessary to use a mechanism to transform price information into action, also taking into account the potential effect of transaction costs. Since transactions have a cost, a threshold representing a percentage was introduced that allows the investor to decide whether it is worth buying or selling. If the percentage increase or decrease in the price expected for the next day is greater than the threshold, the action will be a buy or sell; if it is less than the threshold, the recommended action will be a do nothing action. As you can see from the graph below, the best performances in terms of gains and losses over the test period were achieved using a threshold of 1%.

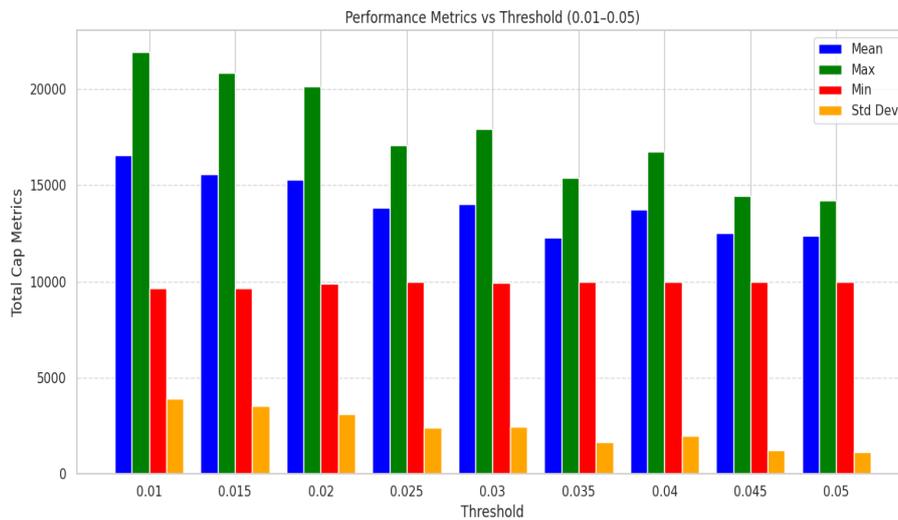


Figure 3.14: Effect of Threshold on Profit and Loss



Figure 3.15: Test Set Results of SVR Model

The model's performance in terms of gains and losses was tested over the period from September 2022 to August 2025, demonstrating promising results. In fact, within three years, the model managed to make almost 160% of its capital. The idea of the back test is to simulate a prudent investor who buys and holds a certain

amount of bitcoin until the model's signal is "buy." When the signal changes to "sell," the investor sells everything he has accumulated up to that point. The back test was conducted following the model's signal with an initial capital of €10,000. The maximum amount for each buy order is defined as the input parameter for the back test and is equal to 0.2% of the available capital. In the graph below you can see the trend of the investor's capital during the test period.

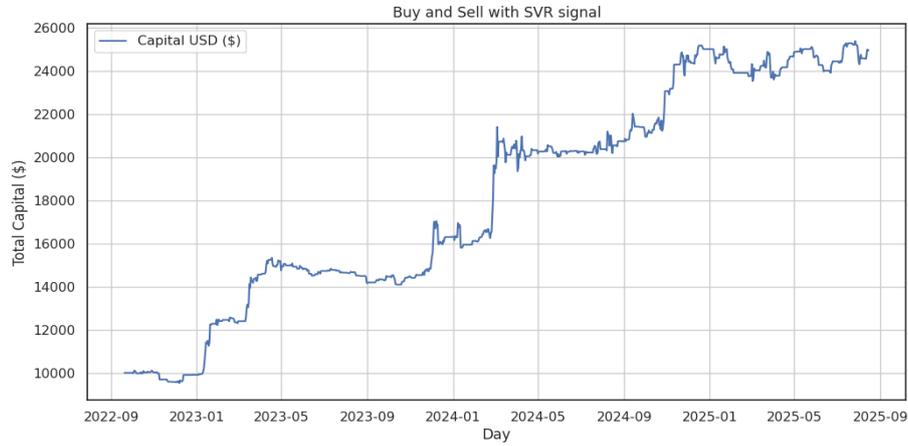


Figure 3.16: SVR Model P&L Performance on Test Dataset

## Chapter 4

### Short Term Forecasting

In recent years, artificial intelligence has been used extensively to help investors understand how financial markets move. Machine learning, and even more so, deep reinforcement learning (DRL), have become a focus of interest in algorithmic trading because they allow you to find new patterns that humans cannot. An investor's goal is to grow their capital by analysing the market and deciding whether it's the right time to buy or sell. The beauty of reinforcement learning is that it allows you to model this problem perfectly. You have an agent (in our case, one or more machine learning models) that continuously interacts with the environment (the market). The agent observes the state of the market, decides what to do, and receives feedback, a reward, which in our case is the change in the value of the portfolio. By playing this game over and over, it learns a strategy that maximizes total gains. But there's a problem: markets are incredibly complex and volatile. Using a single model may not generalize, or rather, work across all conditions. For this reason, in the following thesis, we decided to develop a Multi-Expert Trading System that combines several specialized models with varying granularity. It's a bit like assembling a team of experts: Three reinforcement learning agents working on high-frequency (hourly) data to capture rapid market changes. Two machine learning models (one supervised and one based on supervised and unsupervised learning) that examine daily data to understand long-term trends. Within the RL agents, there are two agents, or rather specialists: a bullish expert and a bearish expert (a bear expert). But there is also a third agent, the master, whose job is to combine all the forecasts (including those from other models like SVR and stock-to-flow) to make the final decision. This structure, a bit like a learning system, makes the entire system much more robust and adaptable than a single model. To understand how well it works, I'll compare it to a basic strategy: the classic annual buy & hold, which would be our benchmark. To understand how the models work, the main theoretical concepts involved will be introduced in the next section.

#### 4.1 Introduction to Neural Networks

The human brain is one of nature's most fascinating wonders because it's composed of billions of neurons that work nonstop, communicating with each other to make our thoughts, emotions, actions, and, most importantly, life possible. And it is precisely this incredible natural architecture that has inspired scientists to create artificial neurons and neural networks, attempting to replicate this ingenious mechanism for processing information. Biological neurons are, essentially, the fundamental building blocks of our nervous system. They are highly specialized cells, designed to receive, process, and transmit electrical and chemical signals. This ability to communicate with each other underlies everything we do, from the perception of a scent to the act of running, to the most complex processes of thought. Each neuron has a well-defined structure, with three main components:

- **Dendrites:** These are the branches that act like small antennas, receiving signals from other neurons through connections called synapses. Their branched

shape serves to increase their surface area, allowing the neuron to collect input from many different sources.

- **Soma (cell body):** This is the actual control center. Here, the neuron adds up all the signals it has received from its dendrites. If the total exceeds a certain threshold, the soma generates an electrical impulse, known as an action potential.
- **Axon:** A sort of long cable that carries the action potential from the soma to other neurons, muscles, or glands. At the end of the axon, the signal is passed to the next cell via the synapses.

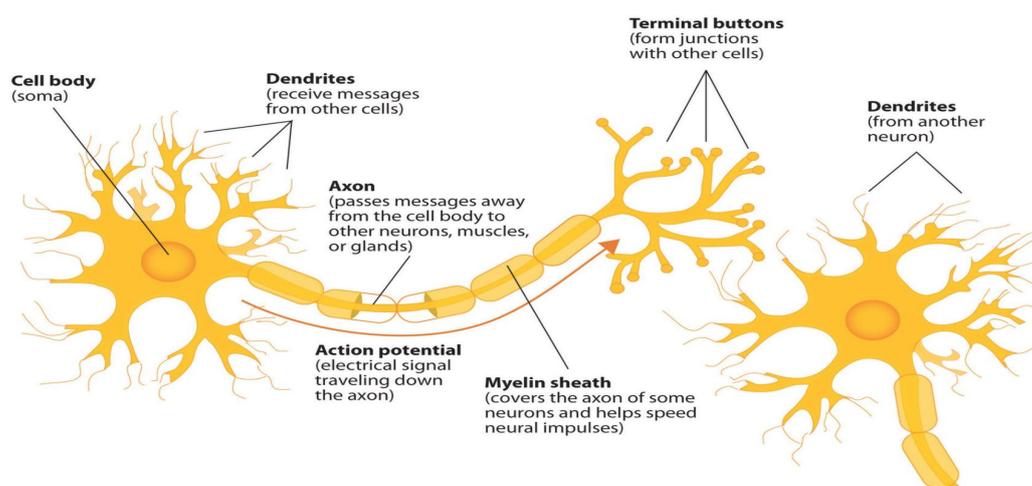


Figure 4.1: The Architecture of a Biological Neuron

The biological neuron works as follow:

- **Receives inputs:** Dendrites take signals from neighboring neurons. These signals can be excitatory (if they push the neuron to fire) or inhibitory (if they slow it down).
- **Integrates the signals:** The soma makes a quick calculation, adding up all the inputs. If the result is strong enough, it lights up and fires the action potential.
- **Transmits the signal:** The electrical impulse travels along the axon and reaches the synapse, ready to pass the baton to the next neuron.

We can think of the biological neuron as nothing more than a super-efficient processor that makes a decision (to fire or not) based on the sum of all the signals it receives. And it is precisely this concept, so simple yet revolutionary, that gave rise to the perceptron, the model of an artificial neuron that underpins the neural networks we use today.

### 4.1.1 The Perceptron

The mathematical model of the artificial neuron that is still used today was introduced in 1943 by Warren McCulloch and Walter Pitts. Its inspired by the biological neuron and can be seen as a computational unit that performs two functions. In the first part the neuron performs a weighted sum of the inputs and then the result of the sum is passed to a special function called activation function. The activation function can be of different nature and has a crucial role in the model because it decide the output of the neuron.

$$S = \sum_{i=1}^n w_i x_i = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

where:

- $x_i$  are the input values,
- $w_i$  are the corresponding weights for each input,
- $n$  is the total number of inputs.

The output is determined using a following activation function:

$$g(x) = \begin{cases} 1 & \text{if } S \geq \theta \\ 0 & \text{if } S < \theta \end{cases}$$

where  $\theta$  is the threshold that decides whether the perceptron 'fires'.

The final output  $y$  can also be expressed as follows:

$$y = g\left(\sum_{i=1}^n w_i x_i - T\right)$$

where  $T$  is the threshold. The perceptron outputs 1 if the weighted sum of inputs exceeds the threshold, and 0 otherwise.

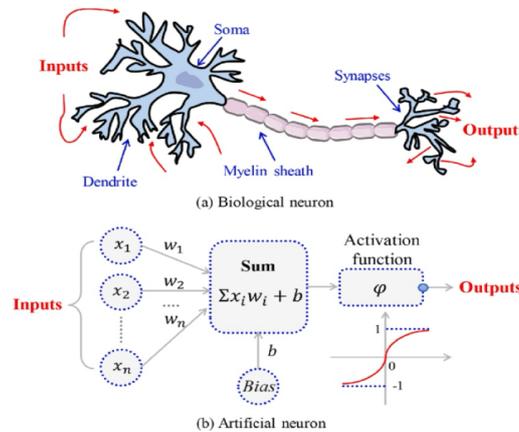


Figure 4.2: The Architecture of a Biological and Artificial Neuron

As we can see in the above image, each neuron receives a series of input values, which are  $x_1, x_2, \dots, x_n$ . Each of these inputs has its own specific weight, which we denote by  $w_i$ . This weight tells us how important that input is for the neuron. To make the model more flexible, we also add an extra value called bias, denoted by  $b$ . In practice, the bias allows the neuron to be activated even in the absence of significant input, or conversely, to require stronger input before firing. If a weight ( $w_i$ ) is large, the input ( $x_i$ ) has a very strong influence on the output of the neuron. If a weight is small or close to zero, those inputs matter little and have minimal influence. The bias acts as a constant value that shifts the neuron's entire function, helping the network to better adapt to the data. Without bias, the network would be less flexible and would be unable to learn more complex patterns.

### 4.1.2 Activation functions

Activation functions are a crucial element for a neural network, they enable it to learn complex patterns. Without them, the network would be merely a linear combination and unable to learn intricate relationships. These functions introduce non-linearity, allowing the model to understand much more complex patterns.

#### Sigmoid

The Sigmoid function compresses any value into an output between 0 and 1. Historically, it was widely used, especially for calculating probabilities, but it has a major flaw: saturation. When the input values are very large or very small, the function flattens, making it difficult to update the weights and causing the vanishing gradient problem.

$$f(z) = \frac{1}{1 + e^{-z}}$$

#### ReLU (Rectified Linear Unit)

ReLU is one of the most popular functions, especially for deep networks. It is incredibly computationally efficient because it simply returns 0 if the input is negative, and the same value if it is positive. This helps solve the vanishing gradient problem typical of Sigmoid.

$$f(z) = \max(0, z)$$

#### Softmax

The Softmax function takes a set of values and transforms them into a set of probabilities that add up to 1. It's the ideal choice for the output layer when performing a multiclass classification because it shows the probability that the input belongs to each class.

$$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

### Tanh (Hyperbolic Tangent)

This function is similar to the Sigmoid, but has one important difference: its output ranges from -1 to +1, not from 0 to 1. Its graph is centered on zero, which can help the network learn faster because the output can be both positive and negative. Despite this advantage, the Tanh also suffers from saturation and the vanishing gradient problem, just like the Sigmoid, when the input values are too large or too small.

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

### Linear

As the name suggests, this function simply returns the input as is. The output is identical to the input, without modification. The linear function is especially useful in the last layer of a network, particularly for regression problems, where the goal is to predict a continuous numerical value.

$$f(z) = z$$

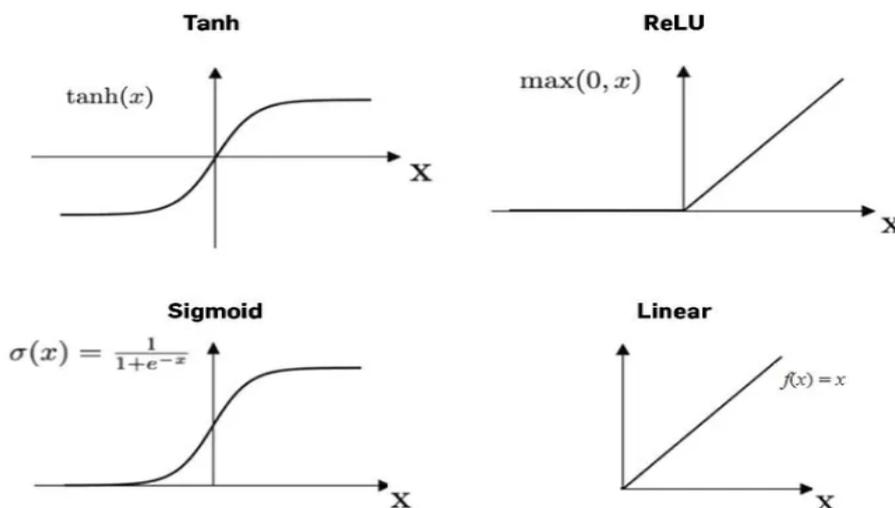


Figure 4.3: Activation functions

## 4.2 The Loss Function

The loss function is a fundamental tool used to measure the discrepancy between the value predicted by a model and the actual or correct value. Its primary role is to quantify the model's error in a single training iteration, returning a numerical value. A higher loss value indicates poorer model performance, while a low value, tending to zero, indicates high accuracy. The primary goal of training a neural network is to minimize this loss function.

To minimize this loss, back-propagation algorithm is commonly used, which is the heart of neural network learning. It is the mechanism that allows a model to learn efficiently by systematically correcting its errors.

The process is divided into several phases, which follow one another in a continuous cycle during training. It all begins with a forward pass operation: an input is fed into the network and passes through each layer until it produces a predicted output. This output is then compared with the correct value, or ground truth, using a loss function. The value of this function quantifies the model's error in that specific iteration.

This is where backpropagation comes into play. The calculated error is propagated backward, starting from the last layer of the network and ending with the first. During this process, backpropagation calculates the gradients of the loss function for each individual weight and bias in the network. Gradients are essentially instructions: they tell the network which direction and by how much each weight should be adjusted to reduce error.

Once the gradients have been calculated, an optimizer (such as SGD or Adam) uses this information to update the model's weights and biases according to the gradient descent rule. This adjustment makes the network more precise, improving its ability to generate accurate predictions in subsequent iterations. In this way, the neural network learns progressively, minimizing loss and optimizing its performance on the task for which it was trained.

The training loop of a backpropagation based model can be summarized as follows:

1. **Forward Propagation:** The model processes an input and generates a predicted output.
2. **Loss Calculation:** The loss function compares the predicted output with the actual output and calculates an error value.
3. **Backpropagation:** The error value is used to calculate the gradients of the loss function with respect to the model weights. These gradients indicate the direction and magnitude needed to adjust the weights to reduce the error.
4. **Weight Update:** An optimizer (such as SGD, Adam) uses gradients to adjust the model weights.

This iterative process allows the model to gradually learn, reducing the error with each cycle and improving its predictive ability.

The choice of loss function depends on the type of problem being addressed; the following two loss functions are the most commonly used:

- **Mean Squared Error (MSE):** Calculates the mean of the squared differences between predicted and actual values. It penalizes large errors quadratically, making it sensitive to outliers.

$$L = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- **Mean Absolute Error (MAE):** Calculates the mean of the absolute values of the differences. It is more robust to outliers than the MSE, as the penalty

is linear.

$$L = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

### 4.3 Deep Learning

Deep learning is a branch of machine learning that uses neural networks, but with a unique feature: they're deep, meaning they have many layers and have an additional part for the feature extraction. Thanks to this depth, they can learn on their own to recognize the most complex features of data. This is why they're so good at solving problems like image recognition, understanding what we say when we speak, or predicting stock market trends. But where does this idea of a deep neural network come from? Well, it's directly inspired by our nervous system, specifically the brain and, even more so, the retina. The retina is the thin film of nerve cells at the back of our eyes that captures light, transforming it into visual information to send to the brain. It's made up of different types of cells, which work together in an orderly, hierarchical fashion. If you think about it, the retina works incredibly similarly to a neural network: Photoreceptors (rods and cones) are like input neurons, taking in the raw signal, i.e., light. Bipolar cells perform the first level of processing and behave just like the hidden layers of a network. Ganglion cells, finally, collect all the processed information and send it to the brain, acting as output neurons. This hierarchical structure, which extracts increasingly complex information from a simple signal, has inspired one of the most powerful deep learning architectures: convolutional neural networks (CNNs). These networks use filters to recognize local elements in images, such as edges and shapes, just as retinal cells extract important information from light.

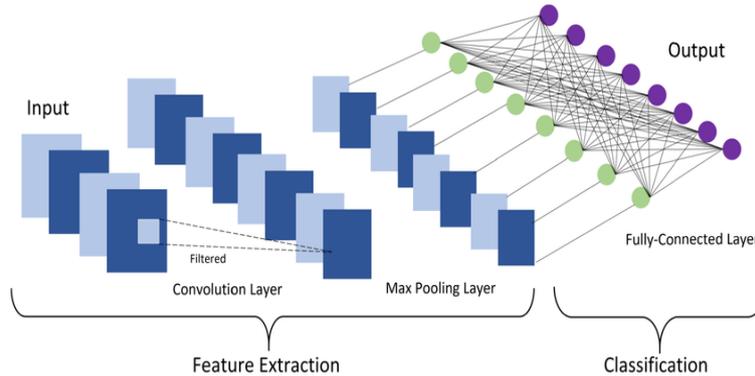


Figure 4.4: Deep Neural Network Ba

### 4.4 Convolution Neural Networks

Convolutional neural networks are based on the convolution process and are commonly used for image classification, segmentation, object detection, natural language processing, some research has shown their potential also in for time series prediction. The architecture of the CNN differs from other neural networks because in the first part of the network it performs a feature extraction process which involves the use

of convolution layers, pooling and RELU, while the second part is generally formed by fully connected layers.

## **Convolution**

The process of applying a convolution kernel to the input data is referred to as the convolution operation. This operation involves sliding a kernel across the input and computing the dot product at each position in order to generate an output feature map. The kernel is mainly a small 2d matrix that acts like a filter. In order to define by how much the kernel has to move forward on the input, a specified parameter named stride can be used.

## **Pooling**

The pooling is also known as subsampling and plays a crucial role in downsampling feature maps while retaining important information. As for the convolution process the pooling process involves sliding a filter over the feature map. Some of the most used pooling layers are the Max Pooling and the Average Pooling. The Max Pooling selects the maximum element from the region of the feature map covered by the filter while the Average Pooling computes the average of the elements present in the region.

## **Fully Connected Layers**

The Fully Connected Layers are the final component of many neural networks, including convolutional neural networks (CNNs). Their function is to integrate the various features extracted by the previous layers to reach the model's final decision. In practice, each neuron in this layer is connected to every neuron in the previous layer, and each connection has its own specific weight. This structure, combined with an activation function, allows for a final prediction, such as the probability that an image belongs to a certain category. By their nature, these layers require a large number of parameters, but they work efficiently because they work on pre-processed features, without having to analyse the image from scratch.

## 4.5 Long Short Time Memory

Long Short-Term Memory (LSTM) is a special type of neural network capable of learning long-term dependencies in sequential data. At each time step, the LSTM takes as input a vector  $x_t$  that represents the current observation, and maintains a hidden state  $h_t$  and a cell state  $c_t$ . The hidden state vector represents the current memory of the network, and the cell state  $c_t$  is a vector that stores long-term information.

In order to control which information to add and remove, the LSTM uses the following gates:

### Input Gate

The input gate performs three main operations to add useful information to the cell state:

- The gate applies a sigmoid function in order to filter the values to be remembered from  $h_{t-1}$  and  $x_t$ :

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

where:

- $i_t$ : This is the input gate vector at time  $t$ . Each value is between 0 and 1 and represents how much of a given piece of information should be let through to be added to the cell's state.
  - $\sigma$ : The sigmoid function, which returns values between 0 and 1, is essential here. It acts as a gate, deciding whether the information is important (value close to 1) or not (value close to 0).
  - $W_i$ : The weight matrix for the input gate.
  - $[h_{t-1}, x_t]$ : This is the concatenation of the previous hidden state and the current input. Essentially, it is the combination of all the relevant information the input gate must consider.
  - $b_i$ : The bias vector for the input gate.
- It creates a vector using the tanh function that gives an output from  $-1$  to  $+1$ , which contains all the possible values from  $h_{t-1}$  and  $x_t$ :

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

where:

- $\tilde{C}_t$ : candidate cell state, potential new information to add to the cell.
  - $\tanh$ : hyperbolic tangent function, outputs values in  $[-1, 1]$ .
  - $W_C$ : weight matrix for the candidate cell state.
  - $b_C$ : bias vector for the candidate cell state.
- The values of the vector and the regulated values are multiplied to obtain the useful information to update the cell state:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

where:

- $C_t$ : cell state at time  $t$ , stores long-term memory.
- $C_{t-1}$ : previous cell state.
- $f_t$ : forget gate vector, determines what information to forget from  $C_{t-1}$ .
- $\odot$ : element-wise multiplication (Hadamard product).
- $i_t \odot \tilde{C}_t$ : new information to be added to the cell state.

### Forget Gate

The forget gate is responsible for removing information that is no longer useful in the cell state. Two inputs,  $x_t$  and  $h_{t-1}$  (previous cell output), are fed to the gate and multiplied with weight matrices. The result is passed through a sigmoid activation function, producing a binary output between 0 and 1 that represents how much of the previous cell state to forget and how much to retain:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where:

- $f_t$ : forget gate vector at time  $t$ , values in  $[0, 1]$ , determines which parts of the previous cell state  $C_{t-1}$  to retain.
- $W_f$ : weight matrix for the forget gate.
- $[h_{t-1}, x_t]$ : concatenation of previous hidden state and current input.
- $b_f$ : bias vector for the forget gate.

### Output Gate

The output gate takes as input the previous hidden state,  $h_{t-1}$ , the current input,  $x_t$ , and the current cell state,  $c_t$ . It outputs a vector of values between 0 and 1 that represents how much of the current cell state to output as the current hidden state:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

The new hidden state  $h_t$  is calculated as:

$$h_t = o_t \odot \tanh(C_t)$$

where:

- $o_t$ : output gate vector at time  $t$ , values in  $[0, 1]$ , controls which information from  $C_t$  is sent to the hidden state.
- $h_t$ : new hidden state at time  $t$ .
- $W_o$ : weight matrix for the output gate.
- $b_o$ : bias vector for the output gate.

### 4.5.1 CNN-LSTM

CNNs are widely used in the literature to perform feature extraction, especially in the context of object detection. Time series contain sequential dependencies and this leads to the need to create models that take into account the order of the data to predict future values. LSTMs are a great model to analyze time sequences and have been proposed countless times in the literature as a model to predict the short-term market price.

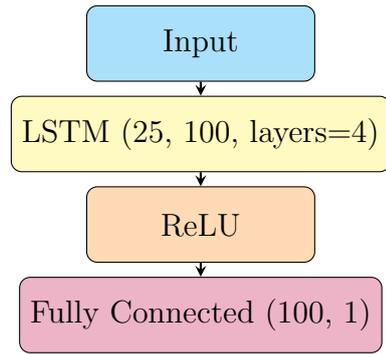
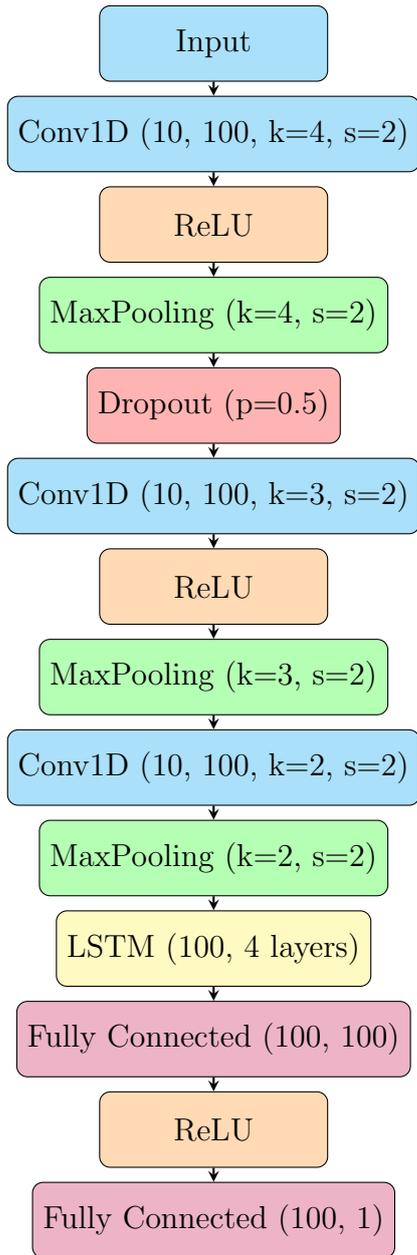
Considering that the price of bitcoin depends on what happened before, we decided to use a sliding window to create sequences of length 10, this sequence represents the number of previous days that the model will analyze to predict the next day. The following on-chain metrics were used for the prediction:

Variable	Description
close price	Close Price
total supply	Total Supply
annual flow	Annual Flow
sf ratio	Stock-to-Flow Ratio
market value	Market Value
market cap	Market Capitalization
realized cap	Realized Capitalization
mrvv	Market-Value-to-Realized-Value
nulp	Net Unrealized Profit/Loss
sopr	Spent Output Profit Ratio
cdd	Coin Days Destroyed
liveliness	Liveliness Ratio
sth mrvv	Short-Term Holder MVRV
lth mrvv	Long-Term Holder MVRV
sth nupl	Short-Term Holder NUPL
lth nupl	Long-Term Holder NUPL
ma60 hash ribbon	60-Day Moving Average Hash Ribbon
ma30 hash ribbon	30-Day Moving Average Hash Ribbon
mwa count 30d change	Mega Whale 30-Day Address Change
addrs balance ge100	Addresses with $\geq 100$ BTC
sa count 100 1k	Shark Address Count (100-1k BTC)
sa count 30d change	Shark 30-Day Address Change
wa count 1k 10k	Whale Address Count (1k-10k BTC)
wa count 30d change	Whale 30-Day Address Change

Table 4.1: Dataset with homogenized variable names (spaces instead of underscores)

Casella in the following article, [7], seeks to demonstrate how on-chain data can be used to detect cryptocurrency market patterns, such as bottoms and tops, bear markets, and bull markets, and how forecasting this data can provide optimal asset allocation for long-term investors. The authors collect six time series of on-chain data from Glassnode (new addresses, active addresses, block height, fees, hash rate, Spent Output Profit Ratio). These metrics show correlations with Bitcoin price

cycles, suggesting that their prediction can support long-term investment strategies. To estimate the future trend of the series, the authors compared statistical (SARIMA) and deep learning (LSTM and CNN) models, highlighting that deep learning models, particularly CNNs, achieve the best performance in almost all metrics. Combining different neural networks has proven to be a successful approach in many cases. In fact Yan and Wei in the following article [13], used a hybrid model to predict the price of bitcoin, composed of a combination of CNN layers with LSTM-type networks. The hybrid model they used performed better than individual models composed only of CNNs or only LSTMs. In order to test which network is better for the prediction, we decide to compare two models: an LSTM and a CNN-LSTM. The LSTM is formed by an LSTM module containing 4 layers and a fully connected layer. The CNN-LSTM was created using the bottom up strategy, we started from a simple architecture and continued to increase the complexity in order to achieve better performance. The CNN-LSTM is formed by two main parts, a part dedicated to feature extraction and a part used for prediction. The CNN contains 3 layers three Conv 1D layers and some Max Pooling and Dropout layers. While the LSTM is formed by an LSTM module of 4 layers and 2 fully connected layers.



Both models were trained using the MSE as loss function and a learning rate of 0.0005. The LSTM model considering that it has a simpler architecture was trained for 200 epochs while the CNN-LSTM for 350 epochs. As we can see from the results, the CNN-LSTM model shows much higher performances than the simple LSTM model.

	R2	MSE	MAE
LSTM	0.734723	0.001559	0.033847
CNN-LSTM	0.879855	0.000706	0.022461

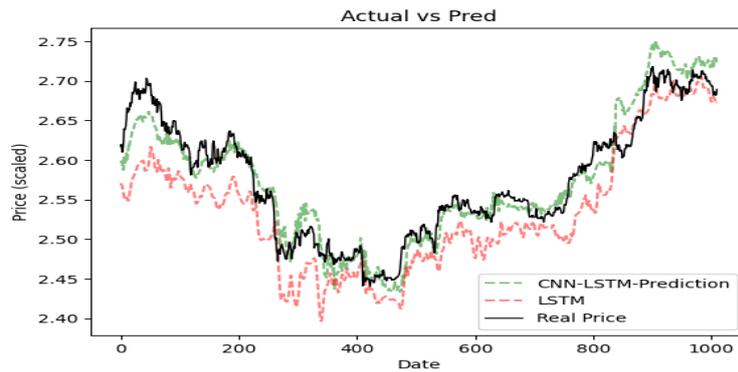


Figure 4.5: LSTM and CNN-LSTM Comparison

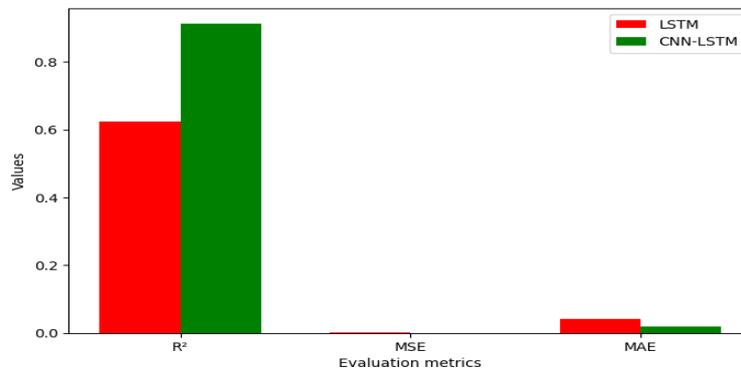


Figure 4.6: Evaluation

## 4.6 Deep Reinforcement Learning

In trading and stock investing, a given investor’s goal is typically to maximize their invested capital by executing a series of buy or sell transactions on one or more assets. This process can be broken down into a series of sequential decisions made by the investor based on an analysis of the information in their possession, the surrounding factors, and, most importantly, the strategy they have developed based on this information analysis. In recent years, reinforcement learning has proven to be a highly efficient learning method for solving sequential decision-making problems.

Many real-world problems such as playing video games, playing sports, driving, optimizing inventory, controlling robots can easily be framed within this paradigm. When we solve these problems, we have a goal or objective, such as winning a game, arriving safely at a destination, or minimizing product production costs. We take actions and receive feedback from the world about how close we are to achieving a goal: our current score, the distance to our destination, or the unit price. Reaching our goal typically involves executing many actions in sequence, each of which modifies the environment around us. RL problems can be expressed precisely as a system consisting of an agent and an environment. The environment produces information describing the state of the system. As the agent interacts with the environment, it observes the state and uses this information to select an action. The environment accepts the action and modifies the state, thereby returning the next state and a reward to the agent. We can summarize this by saying that a reinforcement learning system is a feedback control loop in which an agent and an environment interact and exchange signals, while the agent seeks to maximize the goal. The set of actions, states, and rewards represents the agent's experience. The control loop can repeat infinitely or terminate by reaching a terminal state or a maximum time interval  $t = T$ . The time horizon from  $t = 0$  until the environment ends is called an episode, while the set of experiences in an episode is called a trajectory. An agent typically needs many episodes to learn a good policy, depending on the complexity of the problem. An agent's action depends on what's called a policy, which is simply a function that maps states to actions. Each action will modify the environment and influence what an agent observes and does in the next steps. The exchange between an agent and an environment takes place over time, so it can be thought of as a sequential decision-making process. The Markov Decision Process (MDP) is generally used to model sequential decision-making processes. Agents do not have access to the function that allows them to understand how the environment transitions from one state to another, nor even to the reward function; the only thing they can do is take an action and wait for a reward. The only way an agent can gain information about these functions is through the states, actions, and rewards it actually experiences in the environment.

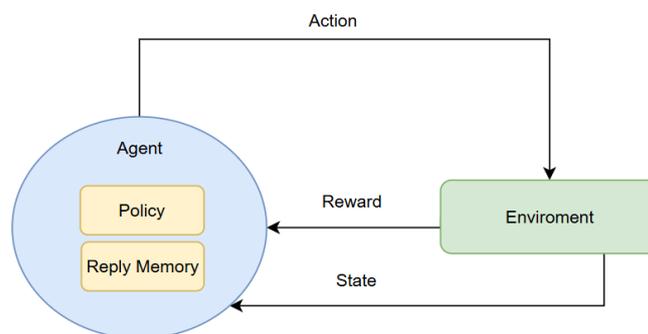


Figure 4.7: Reinforcement Learning Components

## Agent

An agent is an entity that observes the environment, captures details and information, and then executes one or more actions based on those available. The actions it executes are not random; they are determined based on a specific policy that determines the best action to take in that state. The agent must exploit what it already knows to obtain a reward, but it must also explore how to make better choices in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failure. The agent must try a variety of actions and progressively favour those that seem best [19, 10].

## Environment

The environment is defined as the agent's universe. It represents the set of information, also called observations, that the agent has available to take actions. The environment is not fixed; it can change continuously due to external sources or each time an action is taken [19, 10].

## Reward

The reward is the reward the environment gives to the agent following an action. It is generally positive if the action taken by the agent is correct or if the agent is brought closer to its goal. However, if the action is incorrect, the reward is generally negative or zero [19, 10].

## Policy

A policy is a function that maps states to action probabilities, used to sample an action. The agent learns a policy and uses it to act in an environment. A good policy is one that maximizes rewards; the key idea of the algorithm is to learn a good policy, and this means performing a functional approximation. Neural networks are powerful and flexible function approximates, so we can represent a policy using a deep neural network made up of learnable parameters [19, 10].

## Objective Function

An objective can be thought of as an agent's goal, such as winning a game or achieving the highest possible score. There are several algorithms for training reinforcement learning models, among the most well-known being Q-learn [19, 10].

### 4.6.1 Q-learn Algorithm

Q-learn is a model-free reinforcement learning algorithm, with the goal of finding a policy that makes the best decision, or rather, the best action, in a given state, maximizing the final reward.

Before proceeding with the algorithm itself, let's introduce some key concepts:

- State (s): Represents the agent's current state
- Action (a): Represents the action the agent can take
- Reward (r): Represents the reward the agent receives following an action

- Q-value  $Q(s, a)$ : The expected future rewards for taking an action in state  $s$ , following the best policy thereafter.
- Policy ( $\pi$ ): The policy the agent uses to make the decision on which action to perform.

The algorithm updates the Q-value using the Bellman equation as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

- $\alpha$  = learning rate
- $\gamma$  = discount factor that balances immediate vs. future rewards
- $r$  = reward after taking action  $a$  in state  $s$
- $s'$  = the new state after the action
- $\max_{a'} Q(s', a')$  = the best estimated future value from the next state

### The Algorithm

1. Initialize the Q-table by setting an arbitrary value, in most cases zero.
2. Set  $T_{\text{start}} = 0$  as the starting point and  $T_{\text{end}} = N$  as the ending point of each episode.
3. Repeat each episode / epoch:
  - (a) Set  $T_{\text{start}} = 0$ .
  - (b) While  $T_{\text{start}} < T_{\text{end}}$ :
    - i. Randomly select an action with probability  $p$  using the exploration strategy, or take the best action with probability  $1 - p$ .
    - ii. Perform the action and receive the reward  $r$  and the new state  $s'$ .
    - iii. Update  $Q(s, a)$  using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

- iv. Set  $T_{\text{start}} \leftarrow T_{\text{start}} + 1$ .

After trial and error, Q converges to the optimal solution. However, this solution is feasible when the states and actions are discrete, finite, and not very large. If the states are continuous, creating and updating the Q table becomes complicated. Deep Q learning is an evolution of the Q-learn algorithm that addresses the problem of updating the Q table when there are many states. This new approach uses a neural network to approximate the Q table. The algorithm works very similarly, the main difference being that the state  $s$  is fed as input to the neural network, which produces a probability vector representing the probability of performing a given action in that state. Furthermore, instead of updating the Q table after each action, Deep Q learning uses a loss function based on the Bellman equation, and the result is then used to train the neural network usually with Back-propagation [19, 10].

## 4.7 k-Nearest Neighbors (k-NN)

The k-Nearest Neighbors (k-NN) algorithm is a simple, non-parametric, instance-based learning method used for both classification and regression tasks. Unlike parametric models, k-NN does not assume an underlying probability distribution of the data. Its primary principle is that similar data points exist close to each other in feature space, making local patterns useful for prediction.

### 4.7.1 Mathematical Foundation

Given a dataset:

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\}$$

where  $\mathbf{x}_i \in \mathbb{R}^n$  is an  $n$ -dimensional feature vector and  $y_i$  is the label (for classification) or value (for regression), the goal is to predict  $y$  for a new data point  $\mathbf{x}_q$ .

### 4.7.2 Distance Metrics

Distance measurement is crucial in k-NN, as it defines “closeness.” Common metrics include:

- **Euclidean distance:**

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{l=1}^n (x_{il} - x_{jl})^2}$$

- **Manhattan (L1) distance:**

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sum_{l=1}^n |x_{il} - x_{jl}|$$

- **Minkowski distance (generalized form):**

$$d(\mathbf{x}_i, \mathbf{x}_j) = \left( \sum_{l=1}^n |x_{il} - x_{jl}|^p \right)^{1/p}$$

- **Mahalanobis distance:** accounts for correlations between features:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{S}^{-1} (\mathbf{x}_i - \mathbf{x}_j)}$$

where  $\mathbf{S}$  is the covariance matrix.

The value of  $k$  affects performance:

- **Small  $k$ :** High variance, sensitive to noise.
- **Large  $k$ :** High bias, smoother decision boundary.

### 4.7.3 Algorithm Pseudocode

Below we introduce a short sample code of how KNN works based on the type of prediction, i.e. whether it is a regression or classification task.

1. Compute distances:  $d_i = \text{distance}(x_q, X_{\text{train}}[i])$
2. Sort distances and select  $k$  nearest neighbors
3. If classification:
  - Count votes and assign the class with majority  
(optional: use distance-based weighting)
  - If regression:
    - Compute average or weighted average of  $k$  neighbors
4. Return  $y_{\text{pred}}$

### 4.8 Multi Expert Trading System

The development of predictive models for financial markets has always faced the complex challenge of finding the right balance between accuracy, robustness, and adaptability. Traditional approaches often rely on a single model that attempts to generalize and perform well in any market condition. The problem is that financial markets are extremely dynamic and constantly changing, often experiencing actual regime shifts: consider bullish phases, characterized by sustained upward momentum, and bearish phases, marked by prolonged declines. Expecting a single model to optimally capture such diverse dynamics often leads to overfitting or underfitting, and consequently, a poor ability to generalize. To overcome this limitation, the multi-expert approach leverages the principle of ensemble learning, where multiple specialized models (experts) are trained and combined to achieve more reliable and accurate forecasts. Ensemble techniques have long existed in machine learning (like bagging, boosting and random forest) where several weaker predictors are aggregated to create a stronger one. The underlying idea is simple: although individual models may be biased or make errors under certain conditions, their collective output can reduce variance and lead to much more stable decisions.

The multi-expert trading system that we propose is inspired mainly by the way ensemble methods work. They use different models in order to improve the overall model performance. In our context, each expert is like an investor with their own training history, specific learning strategy, or different time horizon.

For example, bull and bear experts specialize in market phases with positive or negative annual returns, respectively, while the master agent integrates their results with those of long-term models, such as stock-to-flow and support vector regression (SVR) experts.

Adopting such a system offers two main advantages:

- **Robustness:** Using different models makes the system less sensitive to the errors of a single expert. This helps the system perform consistently under changing market conditions.
- **Agent specialization:** Dividing the prediction task into specialized experts allows each model to focus on a subproblem (e.g., bullish vs. bearish regimes, short-term vs. long-term dynamics).

This motivation underpins the design of the Multi-Expert Trading System we propose, in which multiple heterogeneous models interact to develop a unified trading strategy. By aligning itself with robust ensemble theory and adapting it to the specificities of financial forecasting, the system aims to achieve predictive reliability that is higher than that achieved with a single model.

#### 4.8.1 System Architecture Overview

The Multi-Expert Trading System was designed as a completely modular architecture, capable of integrating various predictive components into a single decision-making framework.

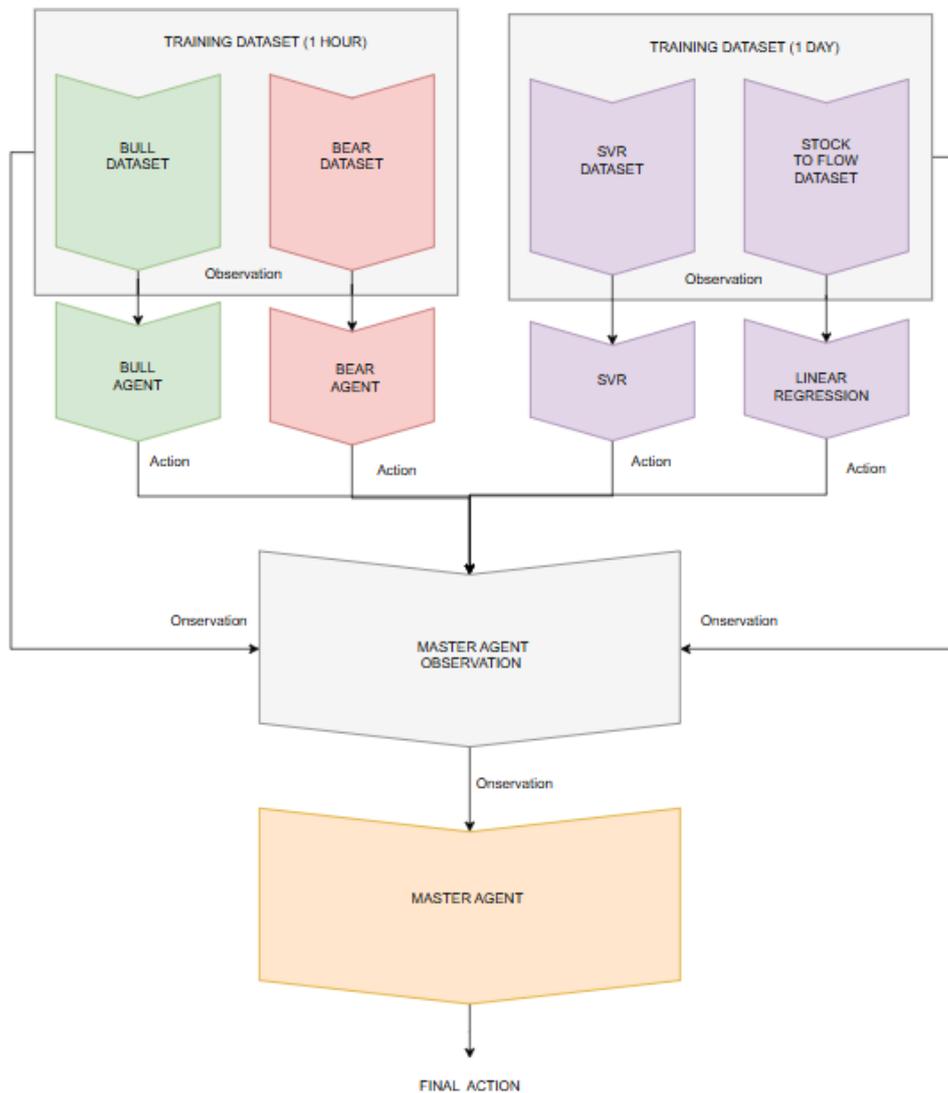


Figure 4.8: Multi-expert system

Its entire structure is divided into three fundamental blocks: the Data Loader, the Expert Models, and the Master Agent.

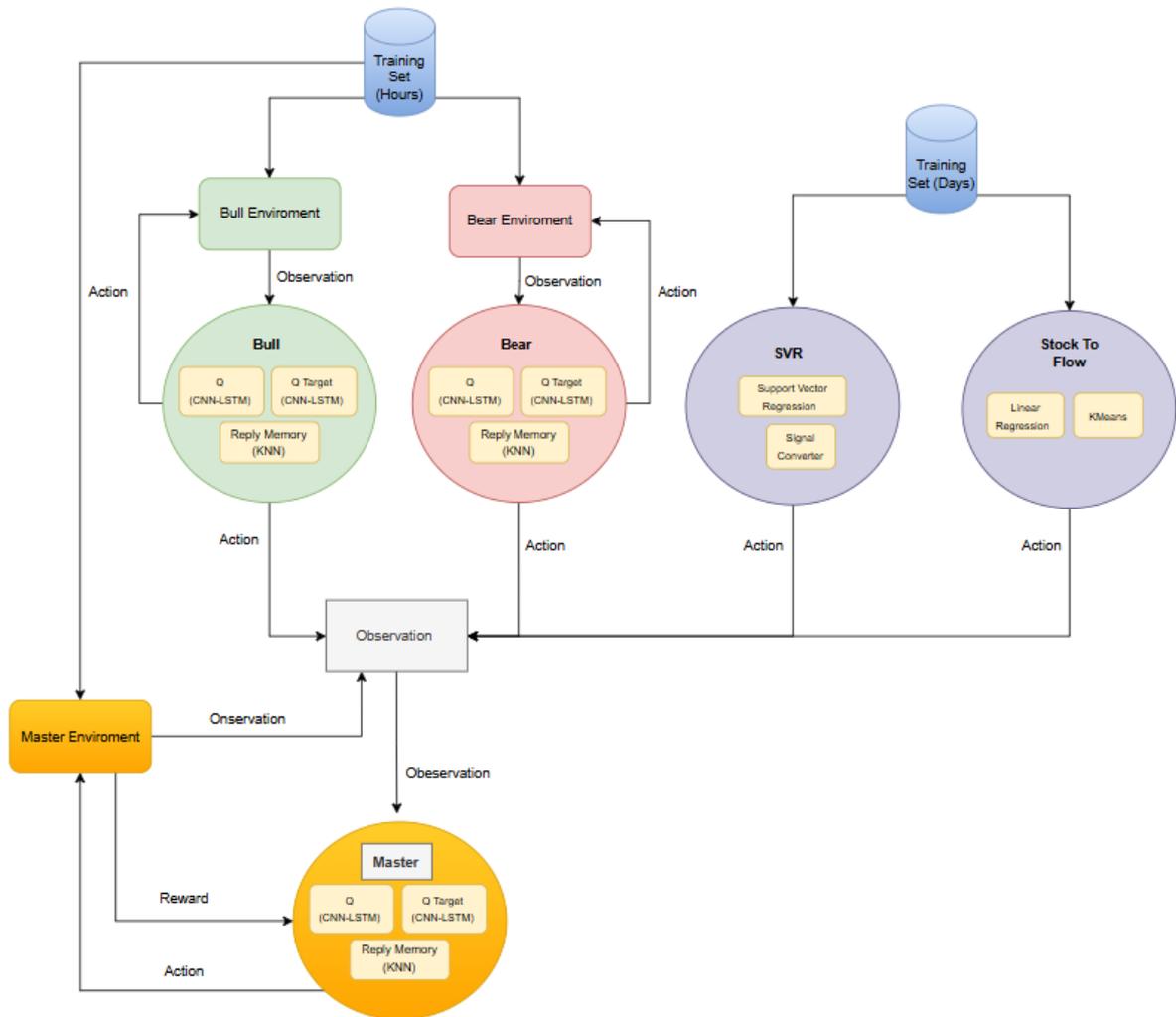


Figure 4.9: Multi-expert system

## Data Loader

The Data Loader's task is to carefully prepare the input dataset. Since our sources are from Glassnode, we receive each indicator in a separate JSON file. The Data Loader then reads the files, aligns the time series, carefully handles any missing data, normalizes variables, and applies any transformations. A crucial aspect was the choice of data frequency, which we differentiated depending on the expert:

- **Hourly Data:** Used for Reinforcement Learning (RL) models. This high frequency is essential for capturing short-term dynamics and providing useful signals for high-frequency trading.
- **Daily Data:** Used by the supervised machine learning models in order to capture the long term trend.

To ensure consistency between variables, we applied a normalization step. In particular, the logarithmic transformation proved extremely effective. This approach reduces data variance, dampens the impact of extreme values, and, most importantly, helps models better capture nonlinear relationships. Since preliminary tests

showed that this transformation significantly improves SVR performance, we also extended it to the hourly datasets used by RL agents.

After cleaning and normalization, the data was divided into training and testing sets. Maintaining sequentiality is crucial for financial data, so we adopted a chronological train/test split scheme, avoiding randomly mixing the samples. In this way, the models are evaluated on scenarios subsequent to those they saw during training, simulating much more realistic operating conditions.

Finally, to create observations for the RL agents, we applied a sliding window technique. Each state provided to the agent is not a single moment, but corresponds to a sequence of data from the last 24 hours. This includes market information and on-chain indicators, providing the agent with a complete dynamic context that allows it to make more informed decisions (buy, sell, or hold) based on recent market developments. In summary, rigorous data preparation ensured that the information fed into the models was consistent, comparable, and statistically stable.

### **Expert Models**

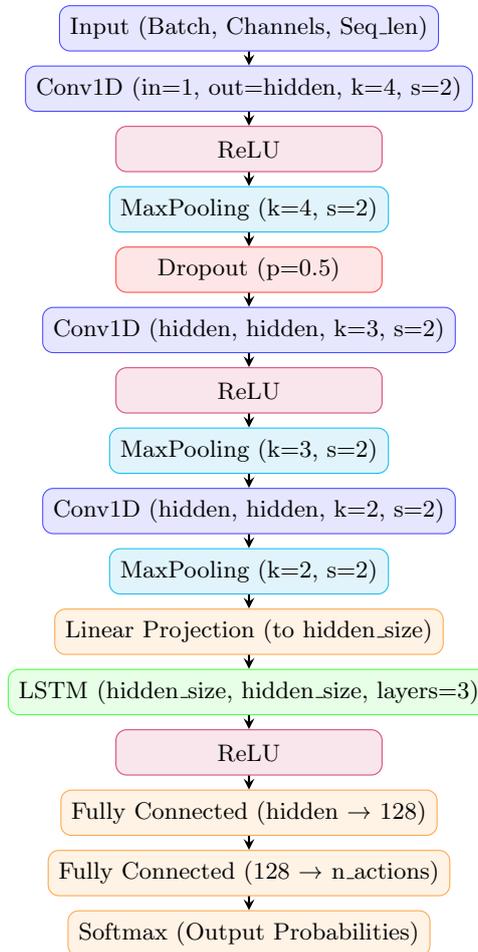
The true heart of the system is the Expert Models, each specialized in a specific task or market phase. We have implemented a total of five experts, combining three Reinforcement Learning (RL) models with two Machine Learning (ML) models. This combination is not random, but is based on two main motivations:

Reinforcement Learning agents are useful for learning new patterns and especially for sequential decision making, those where today's actions directly influence future states and rewards. This makes them perfect for short-term, high-frequency trading. Traditional ML models like SVR trained with on chain data and Linear Regression applied to stock to flow are useful for long-term forecasting.

The three Reinforcement Learning experts who make up the tactical team are:

- **Bull Expert:** Is trained on periods characterized by positive annual returns, he specializes in identifying and exploiting bull market phases.
- **Bear Expert:** Is trained for years with negative or decreasing returns, he is designed to deal with bear or declining markets.
- **Master:** The team leader integrates the forecasts provided by the bull and bear experts with the outputs of the ML models. He analyzes both the raw observations and the recommendations of the other experts to generate the final trading decision.

## Neural Network Architecture (Bull, Bear, Master)



The two Machine Learning experts are responsible for providing balance and a long-term view:

- The SVR (Support Vector Regression) is a regression model with a clear objective: to predict the future price of Bitcoin using market indicators and on-chain metrics. This model was trained on a daily dataset. This choice is strategic because it allows the SVR to focus on medium- to long-term trends and ignore the noise of intraday volatility. We used a Radial Basis Function (RBF) kernel function because it is exceptional at modeling nonlinear relationships between input variables and the target price. As we've already seen, applying the logarithmic transformation to the input data was crucial: by reducing variance and mitigating extreme values, the SVR was able to generate significantly more stable and consistent forecasts.
- Stock-to-Flow (S2F) Expert: This expert is based on a combination of Linear Regression and Kmeans algorithms. The Linear Regression model is applied to find the relation between the stock to flow and the Bitcoin price, while the Kmeans is used in order to compute the signal (buy, sell, do nothing) that the model should return.

By combining agents specialized in short-term tactical trading with models that focus on long-term structural dynamics, the system tries to find a balance between adaptability and stability. This layered design reflects the principles of ensemble learning: the more diversity there is among the underlying models, the greater the robustness and overall predictive power.

The true engine of the Multi-Expert Trading System is Reinforcement Learning (RL) agents, which are designed to mimic, in a structured way, the behavior of an investor. An RL agent learns to make sequential decisions simply by interacting with the environment (which in this case is the cryptocurrency market). Each decision (action) corresponds to a reward proportional to the portfolio's performance.

### **Environment**

The environment we modeled simulates the market in which the agent invests its initial capital. The agent receives an hourly state vector with information useful in order to decide the next state move, this information includes: Market and on-chain data from the last 24 hours Status of open positions Allocated and residual capital Opening price of active positions

### **Actions**

The action space is simple and discrete, with only three possible choices for the agent:

- action =0: No action (hold, i.e., wait).
- action=1: Open or maintain a long position (buy).
- action=2: Open or maintain a short position or close the long position (sell).

### **Rewards**

The reward function we created is defined as the percentage change over time in the portfolio value compared to the initial capital. A positive reward means a gain and a negative one a loss. Essentially, the agent is driven to learn how to maximize the initial capital which is initialized to 10 000 USD.

#### **4.8.2 Stabilization**

In RL, a crucial aspect is finding the right balance between taking new actions to gain knowledge (exploration) using the strategy already known to be successful (exploitation). To manage this, we adopted several techniques.

- Epsilon-Greedy Strategy: With a probability  $\epsilon$ , the agent chooses a random action (exploration); with probability  $1 - \epsilon$ , it uses the already learned policy.
- Replay Memory and KNN: We use a circular memory that stores past experiences (state, action, reward, new state). Instead of randomly sampling from this memory, we implemented a K-Nearest Neighbors (KNN) approach that selects historical experiences similar to the current observation. This stabilizes learning and makes decisions more consistent.

- **Target Network:** To address the instability typical of Q-learning, we introduced a delayed copy of the main neural network. The main network updates at each step, but the target network only periodically. This trick reduces the oscillation of Q values, significantly improving convergence.

### 4.8.3 Results

To compute the performance of our Multi-Expert Trading System, we compared the results of both the individual experts and the combined system (the Master Agent) with a benchmark. We chose the simple annual buy-and-hold strategy. In practice, this strategy consists of buying Bitcoin at the beginning of the year and holding it in the portfolio until the end, without making any trades in between.

Can the multi expert system perform better than the buy-and-hold strategy? In order to answer to this question we've done a backtest of the system over the following period '2022-09-01' - '2025-01-01'.

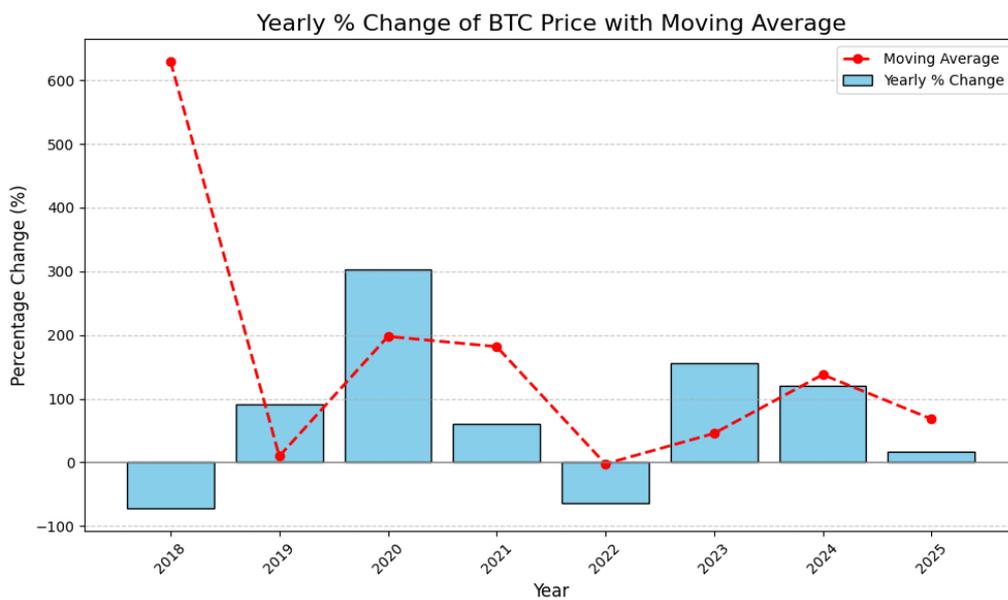


Figure 4.10: Benchmark

As we can see from the benchmark chart, 2023 and 2024 have been very profitable. Buy-and-hold returned  $-64\%$  in 2022, in 2023 were  $155\%$  and in 2024 were  $120\%$ . This is a very high benchmark considering the significant growth, in order to verify performance, we backtested both single models and the multi-expert system and obtained the following results.



Figure 4.11: Bull-expert actions Results

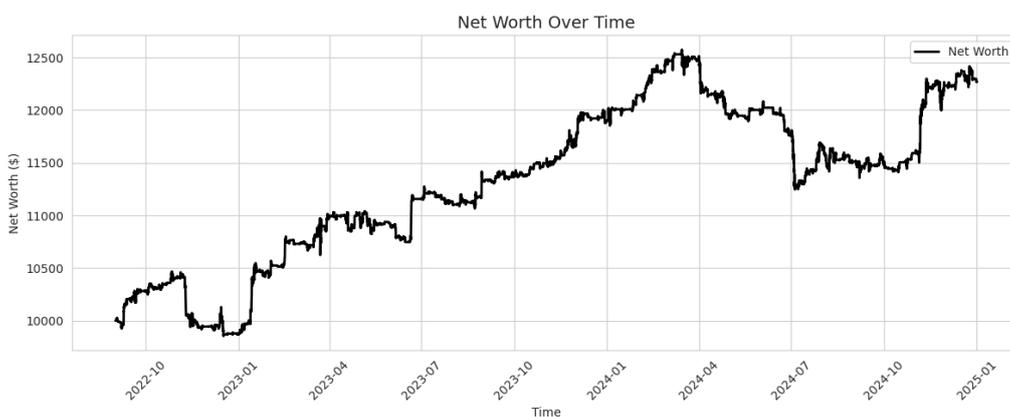


Figure 4.12: Bull-expert capital Results



Figure 4.13: Bear-expert actions Results

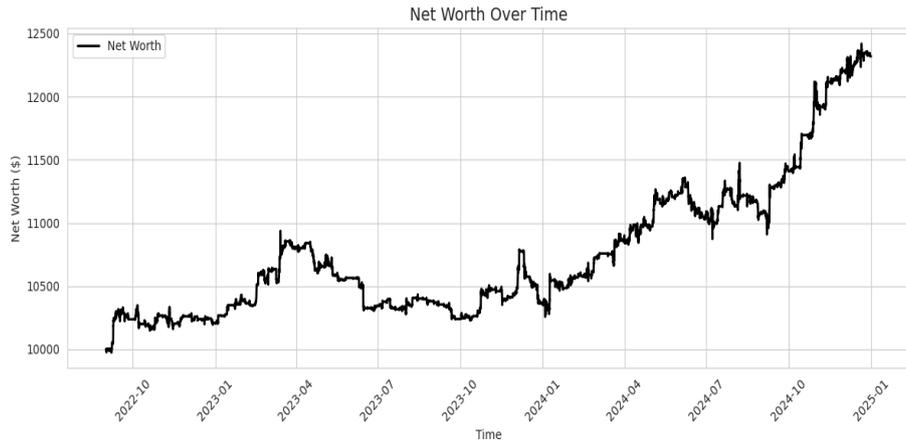


Figure 4.14: Bear-expert capital Results

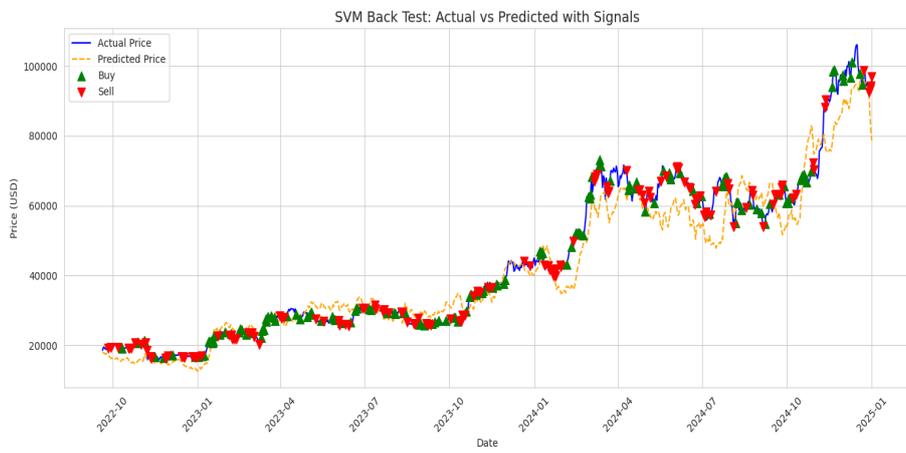


Figure 4.15: SVR action Result

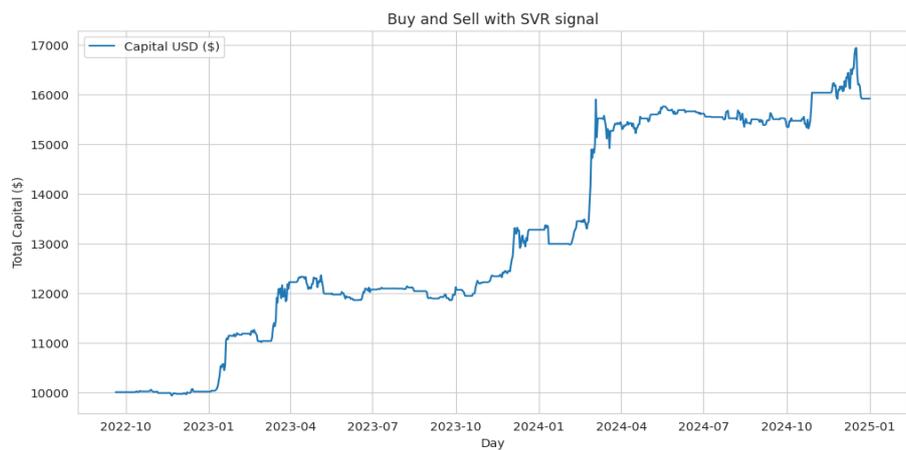


Figure 4.16: SVR capital Result



Figure 4.17: Multi-expert system Results



Figure 4.18: Multi-expert system Capital

As we can see from the images above, all models achieved a capital gain during the testing period. Although the results are far from the predetermined benchmark, we can see that, just as the Bull Expert gained approximately 12 – 12.5% of its capital, it was subject to greater losses when the bitcoin market was declining, while at the same time experiencing greater gains during periods of growth. The Bear Expert also achieved a capital gain of 12 – 12.5% over the three years, however, its capital fluctuations were smaller than the Bull Agent during periods of decline. The most profitable model during this period was the SVR, which achieved a capital gain of 16 – 17% . We can also immediately see that the number of trades executed by the SVR Agent is much lower than those executed by the Bull and Bear Agents, especially during periods of sharp decline or decline. Finally, we can see how the multi-expert system, a combination of the previous models, managed to combine the benefits of the previous models. In fact, it achieved a 15 – 16% gain with fewer orders than the individual Bull and Bear experts. Although the results are far from the target benchmark, this approach shows promise and opens the door to further improvements.

## Conclusion

The analysis conducted in this thesis clearly demonstrates how the cryptocurrency market, and Bitcoin in particular, is an incredibly complex field of study. Starting from the basics and understanding of how blockchain works and its unique characteristics, we were able to introduce and apply a set of on-chain metrics that proved essential for a purely quantitative analysis. The next goal was to combine these metrics with various machine learning methodologies to create a true multi-expert system capable of making operational decisions in the context of automated trading. One of the most fascinating aspects that emerged is the transparent and public nature of information on blockchain. Unlike traditional financial markets, where crucial data may be incomplete or accessible only to insiders, blockchain makes a vast amount of data on transactions, addresses, blocks, and capital movements available to anyone. This allowed us to identify and integrate them into our predictive models. The crypto market in general offers an inexhaustible resource for quantitative research and the application of artificial intelligence. The second key element was the choice and combination of machine learning models. Practice shows that no single algorithm seems perfect in every context. Therefore, we chose to adopt a multi-expert strategy, in which different models supervised, unsupervised, and reinforcement contribute complementary perspectives. Supervised models, such as Support Vector Regression (SVR), have proven highly effective in forecasting short-term actions. Reinforcement learning offered an extremely flexible framework, treating trading as a sequential decision-making process, almost like a strategic game, in which the agent progressively learns the optimal policy based on a reward system. Integrating these experts into a master model that unifies their forecasts generated more robust trading signals than using a single, isolated model. This architectural choice highlights that the complexity of the crypto market requires a variety of tools and perspectives. Much like in medicine, where multiple specialists are often consulted before a definitive diagnosis, in finance the synthesis of algorithmic forecasts can dramatically increase the overall reliability of the system. Despite the encouraging results, it is crucial to be honest about the study's limitations. First, Bitcoin's extreme volatility makes it difficult to build models with consistent performance. What works today may not work tomorrow due to sudden changes stemming from macroeconomic factors, regulations, or simply new speculative dynamics. Furthermore, while on-chain metrics provide tremendous added value, they cannot explain all price dynamics. Exogenous variables such as political decisions, unexpected news, investor sentiment, and global financial market movements remain outside the model. Let's not forget the practical aspects: transaction costs and liquidity, which, in a real-world trading scenario, can significantly impact profitability. Another issue to address is computational complexity, particularly for models based on deep learning and reinforcement learning. Training deep neural networks requires significant resources in terms of time and computing power. Finding a balance between predictive accuracy and operational efficiency is crucial. Future developments in this field should focus on algorithm optimization and the use of more lightweight architectures. In summary, this research is hopefully significant step towards building more intelligent, adaptable, and reliable automated trading systems. While absolute price prediction remains impossible, the combined use of on-chain metrics and machine learning algorithms offers a competitive advantage

in interpreting data and creating operational strategies. Digital markets and traditional finance are converging, and tools like those developed in this thesis will help bridge the gap between academic analysis and practical applications. In conclusion, the work presented here is not a final destination, but rather an invitation to further explore the potential arising from the combination of blockchain, on-chain data, and artificial intelligence.

# Appendix

## 4.9 SVM Model

```
1 class SVRModel:
2
3     def __init__(self, dataset_map, dataset_name, kernel = "rbf", C=1.0, epsilon=0.1,
4                 gamma="scale"):
5
6         self.kernel = kernel
7         self.C = C
8         self.epsilon = epsilon
9         self.gamma = gamma
10        self.base_model = Pipeline([("svr", SVR(
11            kernel="rbf",
12            C=1.0,
13            epsilon=0.1,
14            gamma="scale"
15        ))])
16
17        self.model = MultiOutputRegressor(self.base_model, n_jobs=-1)
18
19        self.dataset_map = dataset_map
20        self.dataset_name = dataset_name
21
22        self.input_size = self.dataset_map[self.dataset_name]["x_train"].shape[1]
23        self.output_size = self.dataset_map[self.dataset_name]["y_train"].shape[1]
24
25    def train(self):
26
27        self.model.fit(self.dataset_map[self.dataset_name]["x_train_reshaped"], self.
28            dataset_map[self.dataset_name]["y_train"])
29
30    def predict(self, datasets, date_to_predict_str, hold_tolerante_threshold=0.01):
31
32        index = self.find_index_by_date_fast(datasets[self.dataset_name]["y_idx_test"],
33            date_to_predict_str)
34        if index == -1:
35            raise ValueError(f>Date {date_to_predict_str} not found in y_idx_test.")
36
37        x_test = datasets[self.dataset_name]["x_test"][index]
38        y_test = datasets[self.dataset_name]["y_test"][index]
39        idx_test = datasets[self.dataset_name]["idx_test"][index]
40        y_idx_test = datasets[self.dataset_name]["y_idx_test"][index]
41
42        if len(x_test.shape) == 2:
43            x_test_flat = x_test.reshape(1, -1) # single sequence
44        else:
45            x_test_flat = x_test.reshape(x_test.shape[0], -1) # multiple sequences
46
47        y_pred = self.model.predict(x_test_flat)
48
49        prices = np.exp(x_test[:, 0]) if x_test.ndim == 2 else np.exp(x_test[:, :, 0])
50        last_price = prices[-1] if prices.ndim == 1 else prices[:, -1]
51
52        y_test_flat = y_test.flatten()
53        y_pred_flat = y_pred.flatten()
54
55        dataset_results = pd.DataFrame({
56            "last_day_idx": y_idx_test,
57            "last_price": last_price,
58            "y_test": np.exp(y_test_flat),
59            "y_pred": np.exp(y_pred_flat)
60        })
61
62        dataset_results['last_pred_price'] = dataset_results['y_pred'].shift(1)
```

```

63     dataset_results['pct_diff'] = (dataset_results['y_pred'] - dataset_results['
        last_pred_price']) / dataset_results['last_pred_price']
64
65     dataset_results['signal'] = np.where(
66         dataset_results['pct_diff'] > hold_tolerante_threshold, 1,      # Buy
67         np.where(
68             dataset_results['pct_diff'] < -hold_tolerante_threshold, -1, # Sell
69             0 # Hold
70         )
71     )
72
73     return dataset_results
74
75 def create_signal(self, datasets, hold_tolerante_threshold = 0.01):
76
77     x_test = datasets[self.dataset_name]["x_test"]
78     print("x_test", x_test)
79     y_test = datasets[self.dataset_name]["y_test"]
80     idx_test = datasets[self.dataset_name]["idx_test"]
81     idy_test = datasets[self.dataset_name]["y_idx_test"]
82     print("idx_test", idx_test)
83     print("y_idx_test", idy_test)
84
85     y_pred = self.model.predict(datasets[self.dataset_name]["x_test_reshaped"])
86
87     prices = np.exp(x_test[:, :, 0])
88     day_idx = [seq_indices[-1] for seq_indices in idx_test]
89
90     y_test_flat = y_test.flatten()
91     y_pred_flat = y_pred.flatten()
92
93     last_price = prices[:, -1]
94
95
96     dataset_results = pd.DataFrame({
97         "last_day_idx": day_idx,
98         "last_price": last_price,
99         "y_test": np.exp(y_test_flat),
100        "y_pred": np.exp(y_pred_flat)
101    })
102    dataset_results['last_pred_price'] = dataset_results['y_pred'].shift(1)
103    dataset_results['pct_diff'] = (dataset_results['y_pred'] - dataset_results['
        last_pred_price']) / dataset_results['last_pred_price']
104
105    dataset_results['signal'] = np.where(dataset_results['pct_diff'] >
        hold_tolerante_threshold, 1,      # Buy
106        np.where(dataset_results['pct_diff'] < -
            hold_tolerante_threshold, -1, # Sell
107            0) # Hold
108
109    return dataset_results
110
111
112 def back_test(self, datasets, initial_capital = 10000, position_size = 0.15,
        hold_tolerante_threshold= 0.01):
113
114     dataset_results = self.create_signal(datasets, hold_tolerante_threshold)
115     dataset_results['position_size'] = 0.0
116     dataset_results['capital'] = initial_capital
117     dataset_results['total_cap'] = initial_capital
118
119
120     shares = 0
121     cash = initial_capital
122
123     for i in range(len(dataset_results)):
124         price = dataset_results['last_price'].iloc[i]
125         signal = dataset_results['signal'].iloc[i]
126
127         if signal == 1: # Buy

```

```

128         invest_amount = cash * position_size
129         shares_bought = invest_amount / price
130         shares += shares_bought
131         cash -= invest_amount
132     elif signal == -1: # Sell
133         cash += shares * price
134         shares = 0
135
136     dataset_results.at[i, 'position_size'] = shares
137     dataset_results.at[i, 'capital'] = cash
138     dataset_results.at[i, 'total_cap'] = cash + shares * price
139
140
141
142     return dataset_results

```

““

## 4.10 Stock To Flow Model

```

1
2 class StockToFlowModel:
3
4     def train(self, x_train, y_train):
5         linreg = LinearRegression().fit(x_train, y_train)
6         score = linreg.score(x_train, y_train)
7         self.linreg = linreg
8         self.score = score
9         return score
10
11     def predict(self, x_test):
12         if not hasattr(self, "linreg"):
13             raise Exception("Model not trained yet. Call train() first.")
14         return self.linreg.predict(x_test)
15
16     def get_action(self, s2f, actual_price):
17         if not hasattr(self, "linreg"):
18             raise Exception("Model not trained yet. Call train() first.")
19         if not hasattr(self, "kmeans"):
20             raise Exception("KMeans not trained yet. Call train_signal_predictor()
21                             first.")
22         if not hasattr(self, "scaler"):
23             raise Exception("Scaler not found. Did you forget to run
24                             train_signal_predictor()?")
25
26         s2f = np.array(s2f).reshape(1, -1)
27         predicted_price = self.linreg.predict(s2f)[0]
28
29         current_distance = actual_price - predicted_price
30
31         current_distance_scaled = self.scaler.transform(
32             pd.DataFrame({"distance": [current_distance]}))
33
34         cluster = self.kmeans.predict(current_distance_scaled)[0]
35
36         cluster_centers = self.scaler.inverse_transform(self.kmeans.cluster_centers_)
37         .flatten()
38         sorted_clusters = np.argsort(cluster_centers)
39
40         # Map clusters to signals
41         signal_map = {
42             sorted_clusters[0]: 1, # Buy
43             sorted_clusters[1]: 0, # Hold
44             sorted_clusters[2]: 2, # Sell
45         }
46         self.signal_map = signal_map

```

```

46     # Get the signal for this single input
47     signal = signal_map[cluster]
48     return signal
49
50
51 def backtest_strategy_stock_to_flow(dataset_results, initial_capital=10000,
52     position_size=0.15, plot=True):
53
54     df = dataset_results.copy()
55     df['signal'] = df['signal'].astype(str).str.strip().astype(int)
56     df['position_size'] = 0.0
57     df['capital'] = initial_capital
58     df['total_cap'] = initial_capital
59
60     shares = 0.0
61     cash = initial_capital
62
63     for i in range(len(df)):
64         price = df['price'].iloc[i]
65         signal = df['signal'].iloc[i]
66
67         if signal == 1: # Buy
68             invest_amount = cash * position_size
69             shares_bought = invest_amount / price
70             shares += shares_bought
71             cash -= invest_amount
72
73         elif signal == -1: # Sell
74             cash += shares * price
75             shares = 0
76
77         df.at[i, 'position_size'] = shares
78         df.at[i, 'capital'] = cash
79         df.at[i, 'total_cap'] = cash + shares * price
80         df.at[i, 'price'] = price
81         df.at[i, 'signal'] = signal
82         df.at[i, 'date_time'] = df.index[i]
83
84     return df
85
86 def train_signal_predictor(self, x_train, y_real, y_pred_train):
87     # Flatten training arrays
88     x_train = np.array(x_train).flatten()
89     y_real_train = np.array(y_real).flatten()
90     y_pred_train = np.array(y_pred_train).flatten()
91
92     # Compute distance
93     distance = y_real_train - y_pred_train
94
95     # Create DataFrame
96     df_train = pd.DataFrame({
97         "x": x_train,
98         "y_real": y_real_train,
99         "y_pred": y_pred_train,
100        "distance": distance
101    })
102
103     # --- Scale distances ---
104     scaler = StandardScaler()
105     distance_scaled = scaler.fit_transform(df_train[['distance']])
106
107     # --- Cluster distances ---
108     self.k = 3
109     kmeans = KMeans(
110         n_clusters=self.k,
111         init='k-means++',
112         n_init=100,
113         max_iter=1000,
114         random_state=71

```

```

115     )
116     self.kmeans = kmeans
117     self.scaler = scaler # store scaler for later use if needed
118
119     df_train['cluster'] = kmeans.fit_predict(distance_scaled)
120
121     cluster_centers = scaler.inverse_transform(kmeans.cluster_centers_).flatten()
122     sorted_clusters = np.argsort(cluster_centers)
123
124     signal_map = {
125         sorted_clusters[0]: 1, # Buy
126         sorted_clusters[1]: 0, # Hold
127         sorted_clusters[2]: 2 # Sell
128     }
129     self.signal_map = signal_map
130
131     df_train['signal'] = df_train['cluster'].map(signal_map)
132
133     return df_train

```

““

## 4.11 Multi Agent System

### 4.12 Single Agent Environment

```

1 class OneHourStockEnv:
2
3     def _get_observation(self):
4         return np.array(
5             [self.balance, self.positions, self.avg_buy_price]
6             + self.history.tolist()
7             + self.mv_z_score.tolist()
8             + self.hr_capitulation.tolist()
9             + self.hr_crossed.tolist()
10            + self.hr_ma30.tolist()
11            + self.hr_ma60.tolist()
12
13            # Address counts
14            + self.plankton_address_count.tolist()
15            + self.shrimp_address_count.tolist()
16            + self.crab_address_count.tolist()
17            + self.fish_address_count.tolist()
18            + self.shark_address_count.tolist()
19            + self.whale_address_count.tolist()
20
21            # Plankton diffs
22            + self.plankton_address_count_7d_diff.tolist()
23
24            # Shrimp diffs
25            + self.shrimp_address_count_7d_diff.tolist()
26
27            # Crab diffs
28            + self.crab_address_count_7d_diff.tolist()
29
30            # Fish diffs
31            + self.fish_address_count_7d_diff.tolist()
32
33            # Shark diffs
34            + self.shark_address_count_7d_diff.tolist()
35
36            # Whale diffs
37            + self.whale_address_count_7d_diff.tolist()
38
39            # Technical analysis metrics
40            + self.fast_ma.tolist()
41            + self.slow_ma.tolist()

```

```

42     + self.rsi.tolist()
43     + self.technical_analysis_signal.tolist()
44     + self.delta_cap.tolist()
45     + self.realized_cap.tolist(),
46     dtype=np.float32
47 )
48
49
50 def step(self, action):
51     """
52     Actions:
53     0 = Hold
54     1 = Buy
55     2 = Sell
56     """
57
58     current_row = self.data.iloc[self.t]
59     current_index = self.data.index[self.t]
60
61     # Price & on-chain metrics
62     current_price = current_row['price']
63     current_mv_z_score = current_row['mv_z_score']
64     current_hr_capitulation = current_row['hr_capitulation']
65     current_hr_buy = current_row['hr_buy']
66     current_hr_crossed = current_row['hr_crossed']
67     current_hr_ma30 = current_row['hr_ma30']
68     current_hr_ma60 = current_row['hr_ma60']
69
70     # Address counts
71     current_plankton_address_count = current_row['plankton_address_count']
72     current_shrimp_address_count = current_row['shrimp_address_count']
73     current_crab_address_count = current_row['crab_address_count']
74     current_fish_address_count = current_row['fish_address_count']
75     current_shark_address_count = current_row['shark_address_count']
76     current_whale_address_count = current_row['whale_address_count']
77
78     # Address diffs
79     current_plankton_address_count_7d_diff = current_row['
80         plankton_address_count_7d_diff']
81     current_shrimp_address_count_7d_diff = current_row['
82         shrimp_address_count_7d_diff']
83     current_crab_address_count_7d_diff = current_row['
84         crab_address_count_7d_diff']
85     current_fish_address_count_7d_diff = current_row['
86         fish_address_count_7d_diff']
87     current_shark_address_count_7d_diff = current_row['
88         shark_address_count_7d_diff']
89     current_whale_address_count_7d_diff = current_row['
90         whale_address_count_7d_diff']
91
92     # Technical analysis metrics
93     current_fast_ma = current_row['fast_ma']
94     current_slow_ma = current_row['slow_ma']
95     current_rsi = current_row['rsi']
96     current_technical_analysis_signal = current_row['technical_analysis_signal']
97 ]
98     current_delta_cap = current_row['delta_cap']
99     current_realized_cap = current_row['realized_cap']
100
101     prev_price = self.data.iloc[self.t - 1]['price'] if self.t > 0 else
102         current_price
103     reward = 0
104     num_shares = 0
105     cost = 0
106     proceeds = 0
107     profit = 0
108     unrealized_profit = 0
109
110     if action == 1:
111         # if self.balance >= current_price:

```

```

104         num_shares = self.balance / current_price
105         cost = num_shares * current_price
106         self.balance -= cost
107         self.avg_buy_price = (
108             (self.avg_buy_price * self.positions + cost) / (self.positions +
109                 num_shares)
110             if self.positions > 0 else current_price
111         )
112         self.positions += num_shares
113
114     elif action == 2:
115         if self.positions > 0:
116             proceeds = self.positions * current_price
117             profit = (current_price - self.avg_buy_price) * self.positions
118             self.balance += proceeds
119             self.total_profit += profit
120             self.positions = 0
121             self.avg_buy_price = 0
122
123     prev_portfolio_value = self.balance + self.positions * prev_price
124     current_portfolio_value = self.balance + (self.positions * current_price)
125     reward = ((current_portfolio_value - prev_portfolio_value) / self.
126               initial_balance)
127     log = {
128         "action": action,
129         "step": self.t,
130         "balance": self.balance,
131         "positions": self.positions,
132         "prev_price": prev_price,
133         "current_price": current_price,
134         "prev_portfolio_value": prev_portfolio_value,
135         "current_portfolio_value": current_portfolio_value,
136         "reward": reward
137     }
138
139     self.t += 1
140     if self.t >= len(self.data) - 1:
141         self.done = True
142
143     price_change = current_price - prev_price
144
145     # Price & on-chain metrics
146     self.history = np.roll(self.history, -1)
147     self.history[-1] = price_change
148
149     self.mv_z_score = np.roll(self.mv_z_score, -1)
150     self.mv_z_score[-1] = current_mv_z_score
151
152     self.hr_capitulation = np.roll(self.hr_capitulation, -1)
153     self.hr_capitulation[-1] = current_hr_capitulation
154
155     self.hr_crossed = np.roll(self.hr_crossed, -1)
156     self.hr_crossed[-1] = current_hr_crossed
157
158     self.hr_ma30 = np.roll(self.hr_ma30, -1)
159     self.hr_ma30[-1] = current_hr_ma30
160
161     self.hr_ma60 = np.roll(self.hr_ma60, -1)
162     self.hr_ma60[-1] = current_hr_ma60
163
164     self.hr_buy = np.roll(self.hr_buy, -1)
165     self.hr_buy[-1] = current_hr_buy
166
167     # Address counts
168     self.plankton_address_count = np.roll(self.plankton_address_count, -1)
169     self.plankton_address_count[-1] = current_plankton_address_count
170
171     self.shrimp_address_count = np.roll(self.shrimp_address_count, -1)

```

```

172     self.shrimp_address_count[-1] = current_shrimp_address_count
173
174     self.crab_address_count = np.roll(self.crab_address_count, -1)
175     self.crab_address_count[-1] = current_crab_address_count
176
177     self.fish_address_count = np.roll(self.fish_address_count, -1)
178     self.fish_address_count[-1] = current_fish_address_count
179
180     self.shark_address_count = np.roll(self.shark_address_count, -1)
181     self.shark_address_count[-1] = current_shark_address_count
182
183     self.whale_address_count = np.roll(self.whale_address_count, -1)
184     self.whale_address_count[-1] = current_whale_address_count
185
186     # Address diffs
187     self.plankton_address_count_7d_diff = np.roll(self.
188         plankton_address_count_7d_diff, -1)
189     self.plankton_address_count_7d_diff[-1] =
190         current_plankton_address_count_7d_diff
191
192     self.shrimp_address_count_7d_diff = np.roll(self.
193         shrimp_address_count_7d_diff, -1)
194     self.shrimp_address_count_7d_diff[-1] =
195         current_shrimp_address_count_7d_diff
196
197     self.crab_address_count_7d_diff = np.roll(self.crab_address_count_7d_diff,
198         -1)
199     self.crab_address_count_7d_diff[-1] = current_crab_address_count_7d_diff
200
201     self.fish_address_count_7d_diff = np.roll(self.fish_address_count_7d_diff,
202         -1)
203     self.fish_address_count_7d_diff[-1] = current_fish_address_count_7d_diff
204
205     self.shark_address_count_7d_diff = np.roll(self.shark_address_count_7d_diff
206         , -1)
207     self.shark_address_count_7d_diff[-1] = current_shark_address_count_7d_diff
208
209     self.whale_address_count_7d_diff = np.roll(self.whale_address_count_7d_diff
210         , -1)
211     self.whale_address_count_7d_diff[-1] = current_whale_address_count_7d_diff
212
213     # Technical analysis metrics
214     self.fast_ma = np.roll(self.fast_ma, -1)
215     self.fast_ma[-1] = current_fast_ma
216
217     self.slow_ma = np.roll(self.slow_ma, -1)
218     self.slow_ma[-1] = current_slow_ma
219
220     self.rsi = np.roll(self.rsi, -1)
221     self.rsi[-1] = current_rsi
222
223     self.technical_analysis_signal = np.roll(self.technical_analysis_signal,
224         -1)
225     self.technical_analysis_signal[-1] = current_technical_analysis_signal
226
227     self.delta_cap = np.roll(self.delta_cap, -1)
228     self.delta_cap[-1] = current_delta_cap
229
230     self.realized_cap = np.roll(self.realized_cap, -1)
231     self.realized_cap[-1] = current_realized_cap
232
233     return self._get_observation(), reward, self.done, current_price,
234         current_index

```

““

## 4.13 Agent

```

1 class Agent:
2
3     def __init__(self, name, input_size, hidden_size, n_actions, lr, gamma=0.90
4         target_update_freq=1000):
5
6         self.name = name
7         self.input_size = input_size
8         self.hidden_size = hidden_size
9         self.n_actions = n_actions
10        self.count_action_from_q = 0
11        self.count_action_from_memory = 0
12        self.lr = lr
13        self.gamma = gamma
14        self.action_space = [i for i in range(self.n_actions)]
15
16        #Policy Network
17        self.Q = CnnLstmNetwork(self.lr, self.n_actions, self.input_size, self.
18            hidden_size, lstm_layers=2)
19
20        #Target Network
21        self.Q_Target = CnnLstmNetwork(self.lr, self.n_actions, self.input_size,
22            self.hidden_size, lstm_layers=2)
23        self.Q_Target.load_state_dict(self.Q.state_dict())
24        self.target_update_freq = 200
25        self.steps = 0
26
27        self.memory = ReplayMemory(action_space = self.action_space, memo_size
28            =5000, train_step=5000, size=input_size)
29
30
31     def choose_action(self, observation, env, prev_action_day, current_action_day):
32
33        state = T.tensor(observation, dtype=T.float32).unsqueeze(0).to(self.Q.
34            device)
35        log = {}
36
37        if np.random.random() > 1-self.gamma:
38            # print("observation", observation)
39            actions = self.Q.forward(state)
40            #print("actions:", actions)
41
42            action_mask = env.get_action_mask(prev_action_day, current_action_day)
43            action_mask_t = torch.tensor(action_mask, dtype=torch.bool, device=
44                actions.device)
45            #print("action mask:", action_mask_t)
46
47            # Replace invalid logits with very negative number
48            masked_actions = actions.masked_fill(action_mask_t == 0, float('-inf'))
49
50            # Get probabilities
51            action_probs = F.softmax(masked_actions, dim=-1)
52
53            action = torch.argmax(action_probs).item()
54
55            # store in log
56            log["mode"] = "exploitation"
57            log["actions_logits"] = actions.detach().cpu().numpy()
58            log["action_mask"] = action_mask
59            log["action_probs"] = action_probs.detach().cpu().numpy()
60            log["chosen_action"] = action
61            log['position'] = env.positions
62            log['balance'] = env.balance
63            log['initial_balance'] = env.initial_balance
64            log['total_profit'] = env.total_profit
65            log['avg_buy_price'] = env.avg_buy_price
66
67            # print("ChooseAction Log:", log)
68            self.count_action_from_q+=1
69        else:

```

```

65
66     action = self.memory.sample(observation)
67     #print("Choosing action based on the memory and randomness: ", action)
68
69     if isinstance(action, (list, tuple, np.ndarray)):
70         action = action[0]
71
72     log["mode"] = "exploration"
73     log["chosen_action"] = action
74     self.count_action_from_memory+=1
75
76
77     # print("ChooseAction Log:", log)
78
79     return action
80
81 def learn(self, state, action, reward, state_):
82
83     self.Q.optimizer.zero_grad()
84
85     states = T.tensor(state, dtype=T.float32).to(self.Q.device).unsqueeze(0) #
86         (1, input_size)
87     states_ = T.tensor(state_, dtype=T.float32).to(self.Q.device).unsqueeze(0)
88         # (1, input_size)
89     actions = T.tensor(action, dtype=T.long).to(self.Q.device).unsqueeze(0)
90     rewards = T.tensor(reward, dtype=T.float32).to(self.Q.device)
91
92     # predicted Q-values for current state
93     q_pred = self.Q.forward(states).gather(1, actions.unsqueeze(1))
94
95     # maximum Q-value for the next state (usando la rete Target)
96     q_next = self.Q_Target.forward(states_).max(1)[0].detach()
97
98     # Q-target (Bellman equation)
99     q_target = rewards + self.gamma * q_next
100
101     #print("q_pred:", q_pred.squeeze(), "Target:", q_target.squeeze())
102
103     # Calculate loss MSE
104     loss = self.Q.loss(q_pred.squeeze(), q_target.squeeze())
105
106     loss.backward()
107     self.Q.optimizer.step()
108
109     self.steps+=1
110     if self.steps % self.target_update_freq == 0:
111         self.Q_Target.load_state_dict(self.Q.state_dict())

```

““

#### 4.14 CnnLstmNetworkNetwork

```

1 class CnnLstmNetworkNetworkV2(nn.Module):
2
3     def __init__(self, lr, n_actions, input_size, hidden_size, lstm_size,
4         lstm_layers=3, input_channels=1):
5         super(CnnLstmNetworkNetworkV2, self).__init__()
6
7         self.input_size = input_size
8         self.hidden_size = hidden_size
9         self.lstm_layers = lstm_layers
10        self.n_actions = n_actions
11        self.input_channels = input_channels
12        self.lr = lr
13        self.lstm_size = lstm_size
14
15        ## CNN layers
16        self.conv1d_l1 = nn.Conv1d(

```

```

16         in_channels=self.input_channels,
17         out_channels=self.hidden_size,
18         kernel_size=4,
19         stride=2
20     )
21     self.relu_conv1d_l1 = nn.ReLU()
22     self.max_pooling_l1 = nn.MaxPool1d(kernel_size=4, stride=2)
23     self.dropout_conv_max_pooling_l1 = nn.Dropout(0.5)
24
25     self.conv1d_l2 = nn.Conv1d(
26         in_channels=self.hidden_size,
27         out_channels=self.hidden_size,
28         kernel_size=3,
29         stride=2
30     )
31     self.relu_conv1d_l2 = nn.ReLU()
32     self.max_pooling_l2 = nn.MaxPool1d(kernel_size=3, stride=2)
33
34     self.conv1d_l3 = nn.Conv1d(
35         in_channels=self.hidden_size,
36         out_channels=self.hidden_size,
37         kernel_size=2,
38         stride=2
39     )
40     self.max_pooling_l3 = nn.MaxPool1d(kernel_size=2, stride=2)
41
42     self.proj = None
43     self.lstm = None
44     self.relu = nn.ReLU()
45
46     # fully connected layers
47     self.fc1 = nn.Linear(self.hidden_size, 128)
48     self.fc2 = nn.Linear(128, n_actions)
49
50     self.optimizer = optim.Adam(self.parameters(), lr=lr, betas=(0.9, 0.999))
51     self.loss = nn.MSELoss()
52
53     self.device = T.device("cuda:0" if T.cuda.is_available() else "cpu")
54     self.to(self.device)
55
56     def forward(self, x):
57         if x.dim() == 2:
58             x = x.unsqueeze(1) # (batch, channels, seq_len)
59
60         # CNN layers
61         out = self.conv1d_l1(x)
62         out = self.relu_conv1d_l1(out)
63         out = self.max_pooling_l1(out)
64         out = self.dropout_conv_max_pooling_l1(out)
65
66         out = self.conv1d_l2(out)
67         out = self.relu_conv1d_l2(out)
68         out = self.max_pooling_l2(out)
69
70         out = self.conv1d_l3(out)
71         out = self.max_pooling_l3(out)
72
73         # flatten for LSTM
74         out = out.reshape(out.size(0), 1, -1) # (batch, seq_len=1, features)
75         if self.proj is None:
76             in_size = out.size(-1)
77             self.proj = nn.Linear(in_size, self.hidden_size).to(out.device)
78             self.lstm = nn.LSTM(self.hidden_size, self.hidden_size,
79                                 self.lstm_layers, batch_first=True).to(out.device)
80
81         out = self.proj(out)
82
83         h0 = torch.zeros(self.lstm_layers, out.size(0), self.hidden_size).to(out.device)

```

```

84     c0 = torch.zeros(self.lstm_layers, out.size(0), self.hidden_size).to(out.
      device)
85
86     # LSTM
87     out, _ = self.lstm(out, (h0, c0))
88     out = self.relu(out)
89
90     # fully connected layers
91     out = F.elu(self.fc1(out[:, -1, :]))
92     out = F.elu(self.fc2(out))
93     actions = F.softmax(out, dim=-1)
94
95     return actions

```

““

## 4.15 ReplayMemory

```

1
2
3 class ReplayMemory:
4
5     def __init__(self, action_space, size, memo_size=5000, train_step=5000, k=5):
6
7         self.memo_size = memo_size
8         self.k = k
9         self.appended_rows = 0
10        self.train_step = train_step
11        self.memory = pd.DataFrame()
12        self.predictor = Pipeline([
13            ("imputer", SimpleImputer(strategy="mean")),
14            ("knn", KNeighborsClassifier(n_neighbors=k))
15        ])
16        self.trained = False
17        self.action_space = action_space
18        self.times_train_predictor = 0
19        self.size = size
20
21
22
23    def add_experience(self, state, action, reward):
24
25        #if reward <= 0:
26            # return
27        # Ensure state is flattened and has a consistent structure
28        state = state[:self.size]
29        state = state.flatten()
30        #print("state_ shape:", state.shape)
31        state_dict = {f"state_{i}": v for i, v in enumerate(state)}
32        #print("state:", state, "state_dict:", state_dict)
33
34        state_dict.update({
35            "action": action,
36            "reward": reward
37        })
38
39        new_row = pd.DataFrame([state_dict])
40        self.memory = pd.concat([self.memory, new_row], ignore_index=True)
41
42
43        if len(self.memory) > self.memo_size:
44            self.memory = self.memory.iloc[1:].reset_index(drop=True)
45
46        self.appended_rows += 1
47
48
49        if self.appended_rows > 1000 and not self.trained:
50            self.times_train_predictor += 1

```

```

51     print("Train memory predictor times= ", self.times_train_predictor)
52
53     df_memo = self.memory.copy()
54     y = df_memo["action"]
55     X = df_memo.drop(columns=["action", "reward"])
56
57     # Train
58     self.predictor.fit(X, y)
59     self.trained = True
60
61     if self.appended_rows % self.train_step == 0:
62         self.times_train_predictor += 1
63         print("Train memory predictor times= ", self.times_train_predictor)
64
65         df_memo = self.memory.copy()
66         y = df_memo["action"]
67         X = df_memo.drop(columns=["action", "reward"])
68         X = np.nan_to_num(X, nan=0.0, posinf=1e6, neginf=-1e6)
69
70         # Retrain periodically
71         self.predictor.fit(X, y)
72
73     def sample(self, state):
74
75         state = state[:self.size]
76
77         if np.random.random() > 0.10:
78
79             if(self.trained == False):
80                 return np.random.choice(self.action_space)
81             else:
82                 row = {f"state_{i}": v for i, v in enumerate(state)}
83                 new_row = pd.DataFrame([row])
84                 return self.predictor.predict(new_row);
85             else:
86                 return np.random.choice(self.action_space)

```

““

## 4.16 Multi Agent System

```

1 class MultiAgentSystem:
2
3     def train_agent(self, end_date=None, n_games=1):
4         if n_games <= 0:
5             print("No training to perform (n_games <= 0).")
6             return self.agent, self.env, pd.DataFrame()
7
8         scores = []
9         results = []
10
11         traders = pd.DataFrame(columns=["Time", "Price", "Action", "Reward", "Game "
12             ])
13
14         isFirstAction = True
15         current_day = 1
16         prev_day = 1
17         print("Training the agent...")
18
19         for i in tqdm(range(n_games), desc="Training Progress"):
20             score = 0.0
21             done = False
22             obs = self.env.reset()
23
24             self.bull_expert.env.reset()
25             self.bear_expert.env.reset()
26
27             while not done:

```

```

27
28     expert_obs = obs[:579]
29
30     current_day = self.env.getCurrentDay()
31     current_date = self.env.getCurrentDate()
32     #print("current_date", current_date)
33
34     if end_date is not None:
35         if pd.to_datetime(current_date) >= pd.to_datetime(end_date):
36             print(f"Reached end_date {end_date}, stopping episode {i+1}")
37             done = True
38             break
39
40     if( isFirstAction == True):
41         prev_day = self.env.getCurrentDay()
42
43     bull_expert_action = self.bull_expert.predictAction(expert_obs,
44 self.bull_expert.env, current_day, prev_day)
45     bull_obs_, bull_reward, bull_done_, bull_price, bull_time = self.
46     bull_expert.step(bull_expert_action)
47     self.bull_expert.add_experience(expert_obs, bull_expert_action,
48     bull_reward)
49
50     bear_expert_action = self.bear_expert.predictAction(expert_obs,
51 self.bear_expert.env, current_day, prev_day)
52     bear_obs_, bear_reward, bear_done_, bear_price, bear_time = self.
53     bear_expert.step(bear_expert_action)
54     self.bear_expert.add_experience(expert_obs, bear_expert_action,
55     bear_reward)
56
57     if bull_done_ is True or bear_done_ is True:
58         print(f"Bull or Bear done, stopping episode {i+1}")
59         done = True
60         break
61
62     #current s2f of the current date
63     target_date = pd.to_datetime(current_date)
64
65     s2f_value = self.s2f_dataset.loc[target_date, "sf_ratio_log"]
66
67     current_price = bull_price or bear_price
68
69     # print("current_price:", current_price)
70
71     if current_price is not None:
72         # Get action from Stock-to-Flow model
73         s2f_model_action = self.s2f_model.get_action(s2f_value,
74     current_price)
75
76         # Get action from SVR model
77
78     else:
79         s2f_model_action = 0
80
81     svr_result = self.svr_model.predict_train(self.svr_dataset,
82     date_to_predict_str=current_date)
83     svr_model_action = svr_result.get('signal', 0) # default to 0 if '
84     signal' missing
85
86     # print("svr_model_action", svr_model_action)
87
88     agent_action = self.agent.choose_action(obs, self.env, current_day,
89     prev_day)
90     obs_, reward, done, price, time = self.env.step(
91     agent_action, bull_reward, bull_expert_action, bear_reward,
92     bear_expert_action, svr_model_action, s2f_model_action
93 )

```

```

85
86         self.agent.add_experience(obs, agent_action, reward)
87
88         prev_day = current_day
89         current_day = self.getCurrentDay()
90
91         score += reward
92
93         self.agent.learn(obs, agent_action, reward, obs_)
94         obs = obs_
95
96         new_order = pd.DataFrame([{"Time": time, "Price": price, "Action":
97             agent_action, "Reward":reward, "Game": i}])
98         traders = pd.concat([traders, new_order], ignore_index=True)
99
100        scores.append(score)
101        avg_score = np.mean(scores[max(0, i - 100): i + 1])
102        print(f"Game {i}: Score {score:.6f}, Avg Score {avg_score:.6f}")
103
104        results.append({"Game": i, "Score": score, "Avg_Score": avg_score})
105        self.agent.saveGame(dataset_path, i)
106
107        results_df_copy = pd.DataFrame(results)
108        results_df_copy.to_csv("training_results_master_agent_last_results.csv"
109            , index=False)
110
111        return self.agent, self.env, trader

```

““

## Bibliography

- [1] Ethem Alpaydm. *Introduction to Machine Learning*. MIT Press, 2010. ISBN: 978-0-262-01243-0.
- [2] Debasish Basak, Srimanta Pal, and Dipak C. Patranabis. “Support Vector Regression”. In: *Neural Information Processing: Letters & Reviews* 11.10 (2007). E-mail: deba65@yahoo.com, srimanta@isical.ac.in, dcp@iee.jusl.ac.in, pp. 203–224. DOI: .
- [3] Imran Bashir. *Mastering Blockchain*. Packt Publishing, 2018. ISBN: 978-1-78883-904-4.
- [4] Massimo Bergamini and Anna Trifone. *Matematica.blu 5 (Volume 5)*. Zanichelli, 2017. ISBN: 978-88-08-50002-1.
- [5] Marco Boella. *Probabilità e Statistica per l’Ingegneria e la Scienza*. Pearson, 2011. ISBN: 978-88-7192-618-6.
- [6] Andrea Caliciotti, Marco Corazza, and Giovanni Fasano. “From regression models to Machine Learning approaches for long term Bitcoin price forecast”. In: *Annals of Operations Research* 336 (2024), pp. 359–381. ISSN: 1572-9338. DOI: 10.1007/s10479-023-05444-w.
- [7] Bruno Casella and Lorenzo Paletto. “Predicting Cryptocurrencies Market Phases through On-Chain Data Long-Term Forecasting”. In: (2023), pp. 1–4. DOI: 10.1109/ICBC56567.2023.10174989.
- [8] Alexander Elder. *The New Trading for a Living*. John Wiley & Sons, Hoboken, New Jersey, 2014. ISBN: 978-1-118-44392-7.
- [9] Giovanni Fasano and Andrea Pontiggia. *Data Analytics and Machine Learning paradigm to gauge performances combining classification, ranking and sorting for system analysis*. Rapporto di Ricerca , Working Paper, 05-2021, Venice School of Management. 2021.
- [10] Laura Graesser and Wah Loon Keng. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Boston, MA: Addison-Wesley Professional, 2019. ISBN: 978-0135172384.
- [11] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2012. ISBN: 978-0-12-381479-1.
- [12] Chien-Yi Huang. “Financial Trading as a Game: A Deep Reinforcement Learning Approach”. In: *CoRR* abs/1807.02787 (2018). Preprint, Department of Applied Mathematics, National Chiao Tung University, Hsinchu, Taiwan; E-mail: cyhuang.am03g@nctu.edu.tw. DOI: 10.48550/arXiv.1807.02787.
- [13] Yan Li and Wei Dai. “Bitcoin price forecasting method based on CNN-LSTM hybrid neural network model”. In: 2020.15 (2020). Journal of Engineering, Presented at the 3rd Asian Conference on Artificial Intelligence Technology (ACAIT 2019), pp. 407–411. ISSN: 2051-3305. DOI: 10.1049/joe.2019.1203.
- [14] Jojo Moolayil. *Learn Keras for Deep Neural Networks*. Apress, 2019. ISBN: 978-1-4842-4239-1.
- [15] Andreas Müller and Sarah Guido. *Introduction to Machine Learning with Python*. O’Reilly, 2016. ISBN: 978-1-449-36941-5.
- [16] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Rapporto Tecnico/White Paper. Disponibile su: <http://www.bitcoin.org/bitcoin.pdf>. Oct. 2008. URL: <http://www.bitcoin.org>.
- [17] Brett N. Steenbarger. *Trading Psychology 2.0*. Wiley, 2017. ISBN: 978-1-118-93683-2.
- [18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2014.
- [19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. A Bradford Book. Cambridge, MA: MIT Press, 2018. ISBN: 978-0262039246.
- [20] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005. ISBN: 0-321-42052-7.