Trading Strategy Discovery in the Cryptocurrency Market using a Genetic Algorithm

March 2022

Contents

List of Figures

List of Algorithms

Listings

Abstract

This project aims to find novel trading strategies for trading the Bitcoin cryptocurrency asset in the 15 minute trading window. A genetic algorithm (GA) is used to generate and evolve trading strategies from composable building blocks of technical indicators and logical conjunctions. Using technical indicators emulates human approaches to trading and preserves the domain knowledge that they represent. This is in contrast to recent studies which have looked to put more onus on machine and deep learning techniques. However as examined in the literature review, these techniques may suffer due to the complex and chaotic nature of cryptocurrency markets, and consequently hit problems regarding generalisability, overfitting and explainability.

The concept of a trading strategy is used to denote a signal to execute an order (either buy/sell/hold) based upon multiple TI indicators. Indicators are pooled from four categories, these being: Trend based, Volume based, Oscillators, and Volatility indicators. An abstract data model is presented to handle the encoding of arbitrarily complex trading strategies so that they can be randomly generated in code and further evolved through the genetic mutation process. As GA is an inherently computationally heavily workload a high degree of parallelism through multi-processing was built into the system. Both horizontal and vertical scaling were considered as means of further improving the run time performance of the algorithm. The system was containerised using Docker and deployed to a cloud server, in accordance with modern software engineering practices. Therefore this project presents a fully portable container artefact which could be turned into a production system with further scaling.

The final experiment used an evolutionary process of 50 generations with a population size of 50 strategies per generation. Each of these strategies was tested against historical data to evaluate its relative success. The use of historical rather than live data ensures impartiality during the process as there are no financial implications to the execution of trades.

The results are supportive to the theory that GA is a suitable candidate research area for algorithmic trading using traditional human measures, and therefore also support the notion that there is still value in non-DNN led approaches to algorithmic trading. Further research is required to optimise the GA.

This work contributes to a growing body of research into the potential for AI-driven techniques to outperform traditional technical analysis-based approaches in 'hard-to-value' markets, such as cryptocurrency.

Chapter 1

Introduction

This section provides further background to core components of the project, these being: the cryptocurrency market; technical analysis approaches to trading systems; machine learning and deep learning approaches to building trading systems; genetic algorithms, and software engineering principles and their applications. Further, it details the motivations and contributions of the project, and outlines the scope and objectives of the project.

1.1 Algorithmic Trading

Algorithmic trading is the application of 'computerized executions of financial instruments' in a given market for a given asset or asset class [28]. Broadly speaking an algorithm can be thought of as the execution of a set of mathematical instructions that translate to investors' trading goals. [37] provides a useful definition that algorithmic trades are rule-based and can be viewed as a 'programmed policy which is either deterministic or stochastic, which outputs a trading action according to the information available to the trading agent i at time step t.' At a fundamental level, algorithmic trading intends to automate the work carried out by human traders in a given market to increase speed and efficiency, remove bias and provide consistency. Since the early 2000s, the application of algorithmic trading has exploded with up to 92% of foreign exchange (Forex) trades executed programmatically rather than by human traders in 2019 [28]. The drivers for algorithmic trading over human-initiated trades can be broadly categorised as follows:

1. An algorithm is non-subjective in its execution (i.e. an algorithm only considers the current market data and its predefined operations, without influence from external human factors such as fear or hope). [1]

2. The use of an algorithm cuts out the need for a 'middleman' (e.g. a broker) in the process of placing buy/sell orders.

3. An algorithm can be executed at high speed and repetitively upon trigger conditions being hit. This opens the door for high-frequency trading (HFT) with each trade only aiming for marginal gains (aka 'scalping').

[1] Not withstanding algorithms which deliberately factor in human emotion for example those which use social media feeds such as Twitter as an input to the algorithm's understanding of the market state.

4. An algorithm can be expected to exhibit consistent behaviour in a given market state; providing that it has been trained on similar market states.

There has also been a rise in the popularity of algorithmic trading amongst retail investors, with motivations including: a reduction in management or brokerage fees when using algorithmic trading; reduced transaction costs; wide market access, and transparency (for rules-based algorithms which behave predictably) [28]. Conversely, the potential downsides of using algorithmic trading include complacency and placing excessive or undue trust in algorithm performance. Limitations to algorithm performance include the inability to adapt to unplanned events or unforeseen market conditions (humans may often utilise or rely upon an innate 'judgement' or 'expertise', whereas algorithms are limited to doing as they have been explicitly programmed or stochastically trained). Additionally, multiple algorithms may be required for different market conditions, i.e. one would not expect to be able to trade in the same manner in a bear (that is, a market where selling pressure is outperforming buying thus pushing prices down) vs a bull market (where high buying activity creates upward pressure on prices). Consequently, 'black-box' type algorithms with a reliance on deep learning trading decisions can lack transparency and explainability [28].

There are broadly speaking three categories of algorithmic trading strategies, these being:

- Execution algorithms
- Profit-seeking algorithms
- High-frequency trades

The respective objectives of which can be thought of as: portfolio selection/optimisation (e.g. of a given stock); executing buy and sell signals over given assets to trade at a profit, and performing profit-seeking at a very high frequency (i.e. at low time frame intervals) to make incremental marginal gains on an account. This third category of algorithmic trading strategy is demonstrative of the real-world impact of algorithmic trading on market activity; effectively giving rise to an entirely novel class of market participants. The 'high-frequency trader' (HFT) did not exist in the year 2000, but by 2010 HFTs accounted for 33% of the total trading volume, with a concurrent decrease in proportional contribution from asset managers and hedge funds [28].

These algorithms in each of these classes can be fine-tuned to match the risk appetite of the investor/fund on an aggressive-passive continuum (i.e. from seeking growth aggressively at a high risk, or choosing a low/moderate risk with slower growth potential [36]). An investor might use an established metric portfolio evaluation metric such as the Sharpe ratio, which gives an indication of the risk-adjusted gain on an account to tailor an algorithm's performance to parallel their own appetite for risk.

1.2 Cryptocurrency

Since 2009 the total market capitalisation of all cryptocurrency assets (that is the total price of all assets * the total number of assets in circulation) has grown year on year. It stands at over $2.5 Trillion USD at the time of writing according to market analytics site CoinGecko.

Cryptocurrencies are typically traded through an exchange in which retail and professional investors buy/sell or swap digital assets. Exchanges can be categorised as either centralised (with

the market leaders being Binance and CoinBase) or decentralised (e.g. SushiSwap, UniSwap). Centralised exchanges are most similar to traditional trading platforms for the stocks and Forex markets in that a centralised authority owns and maintains the platform and trades. Conversely, decentralised exchanges (DExs) offer a peer-to-peer means of directly swapping one cryptocurrency for another via automated market maker (AMM) smart contract algorithms [23]. In such algorithms, assets are paired together to facilitate dynamic price discovery based on the proportional representation of the asset in the pool. In both models, the exchange makes income via transaction fees which are applied to trade, but the models differ in that in the centralised model all of the transaction fees provide revenue for the exchange whereas in the DEx model a portion of the fee is redistributed to liquidity providers in the network. Network participants who provide liquidity to the platform in the form of the cryptocurrency coins/tokens that they offer to be traded on (incentivising 'liquidity providers' to participate in the exchange long term as a passive income generating activity)[23].

1.3 Algorithmic Trading in Cryptocurrency

The rise of algorithmic trading in traditional markets has been prominent, therefore one might assume the same mechanics might also apply to cryptocurrency. Indeed, in many ways algorithmic trading is a more natural choice in the cryptocurrency market due to idiosyncrasies present in cryptocurrency vs traditional financial markets, these being:

- The market never sleeps. The cryptocurrency market trades 24 hours a day, i.e. there is no market open or close time as is the case with the stock exchanges. For serious and professional traders there is a great risk of missing periods of high price volatility at unsociable hours. This problem is immediately solved by the use of algorithmic trading as clearly the prospect of running an algorithm with a good degree of reliability, consistency and precision around the clock is much better than that of a human trader. An algorithmic trading system does of course have other complex runtime requirements such as CPU and RAM constraints in the system, internet connection speed, scalability etc. but a requirement for sleep is out of the equation.

- The cryptocurrency market is in general highly volatile, offering greater risk at a greater reward in a shorter time period than traditional markets. New crypto millionaires (even billionaires) are created regularly as low market cap assets see huge price surges meaning that a modest initial holding can become incredibly valuable overnight (aka a 'moonshot'). One of the most notorious examples is that of a $8000 USD holding of the Shiba Inu token growing to a value of 5.7B USD in under a year [1]. It is easy for human investors to become influenced by hope and fear (specifically 'fear of missing out' or 'FOMO') and thus make poor choices when trading [27]. Algorithmic trading overcomes this fragility by acting entirely impartially at the point of trade as it has no vested interest in the outcome of the trade ('no skin in the game' so to speak).

- The cryptocurrency market is both highly volatile and also still quite nascent, because of this judging good/bad trades can be complex for individuals vs an algorithmic approach that is better placed to learn from data and implement impartiality and consistency of decision-making. Algorithms can be skewed by biased or unbalanced data. Whilst this can lead to misclassification of an output, any such error would at least be consistent and thus correctable

upon further analysis. Humans are liable to make not just the same mistakes again, but also make different mistakes at different times.

- Even the most experienced human trader is unable to process as many data points as an algorithm in a given time frame, by orders of magnitude. Even expert, successful trading experts stand to benefit from the insight that ever-improving algorithmic strategies may provide.

1.4 Technical Analysis

Technical Analysis (TA) is a form of analysis that has been implemented historically by traders to analyse the market (or performance of a given asset in the market) to inform trading decisions [28]. TA is often used to counteract some of the flaws in human-led trading (such as bias, fear, optimism/pessimism, or feelings involving luck) to add a systematic approach to running trades and improve consistency in decision-making. TA introduces a systematic approach for traders to use when evaluating trades, akin to algorithmic trading by hand.

Fundamental analysis looks to use data about a given asset's fundamental value (such as the company's end-of-year financial statements, the state of the industry in which a company operates, and market forces such as external regulatory factors etc.) as part of the fair market hypothesis. Comparatively, technical analysis casts a closer net on the performance of the asset in a given time window to provide metrics such as RSI (Relative Strength Indicator), MA (Moving Average), EMA (Exponential Moving Average), BB (Bollinger Bands) to name but a few. These indicators are driven by real-time and historic market activity data (such as price, volume data or buy/sell orders currently awaiting execution in an exchange) and provide lagging (or in some cases leading) indicators towards an asset's price movement [8].

As an example, let us consider the RSI indicator. The RSI is used to give an indication of the strength of a price change in an asset. It is classified as a momentum oscillator as it is in the 0-100 range (other indicators such as averages will be tied to an asset's price and therefore have no fixed bounds) and denotes the momentum of price change in the market. When the RSI is over 70 the asset is generally considered to be in an 'overbought' state meaning that it is a good time to look to sell, the inverse is true when the RSI is below 30 [16].

Armed with these indicators trading strategies can be constructed. That is, one can combine multiple indicators and thresholds to act as a signal and execute a trade once the signal fires. A trivial example might be an RSI-based strategy which states:

$$SELL \iff RSI(indicator) \geq 70(threshold) \vee BUY \iff RSI(indicator) \leq 20(threshold). \tag{1.1}$$

This strategy is deemed trivial as it contains only one indicator. Majoritively, professional traders would consider such strategies of little value to trade as there are too many complex variables in the market affecting price to base financial decisions on a single data point.

There is a trade-off here in that the more sophisticated a strategy (i.e. the more indicators and thresholds combined) the more accurate the strategy is likely to be, however, it will have fewer opportunities to trade due to the constraints on wider applicability imposed by increasing quantities of parameters. There is little point in having a strategy with a 99% win rate if it is only applicable to execute once per year. Conversely, a frequently firing strategy with a poor success rate

is guaranteed to lose money. Balancing the level of sophistication with the trading opportunities that a strategy presents is a challenge for retail and institutional traders. This challenge highlights a point of difference offered with algorithmic trading, as strategy metadata such as: the number of generated trades in a timeframe; the percentage of wins/losses, and the precision/recall of trading decisions (that is the false positive and false negative rates) can be evaluated to optimise the strategy accordingly. Furthermore, after re-optimising the strategy it can be expected in a consistently a given set of inputs (even if the algorithm is non-deterministic, given the same set of inputs we can reasonably expect similar decision points or classifications), the same cannot be said of a human trader so may still be influenced by psychology and factors such as 'gut' or 'intuition' despite looking to trade in a systematic manner.

[20] argue that it is beneficial to incorporate domain knowledge into a trading system (i.e. using established TIs preserves the conventional wisdom of traders vs a blackbox deep learning approach). There is an implicit assumption here that this domain knowledge is correct, relevant and useful. Recent developments in reinforcement learning such as alphaGo/alphaGo Zero and the subsequent alphaGo chess engine challenge this assumption, showing that embedding domain knowledge in a model is not always the optimum solution. In the alphaGo context, Deep Reinforcement Learning (DRL) networks greatly outperformed those which included conventional wisdom (in the case of the game playing engines this is the difference between giving the engine the rules of the game and allowing it to learn via reinforcement or training it on data from top-level human played games). Any system using TI as input is implicitly preserving the domain knowledge of the traders and market analysts from whom the indicators originated. It is out of the scope of this project to attempt to definitively answer whether knowledge-preserving ML can outperform deep learning AI in the domain of cryptocurrency algorithmic trading, however, in choosing to use TIs as inputs this study deploys an approach which at least partially preserves conventional wisdom. As such this study does contribute to the question of whether such approaches are at all applicable to the domain.

1.5 Genetic Algorithms

Genetic algorithms (GA, AKA genetic programming (GP) are a non-deterministic approach to finding reasonable approximate maxima given a problem with a large search space. The approach was originally introduced by [21] in 1992 as a way of mimicking the natural evolutionary process to find novel solutions to complex problems. GP belongs to a class of algorithms called evolutionary programming which aim to 'evolve' a solution by mimicking traits observed in nature and evolution, such as natural selection, inheritance and diveristy. GP specifically looks to mimic the evolutionary reproductive process, wherein genes are inherited from parents and mutated in subsequent generations. GA been used for a wide range of applications, perhaps most famously training robotic autonomous agents to walk and fight for food [13].

It should be noted that there is no guarantee a genetic solution will be optimal, but GP offers a means to find useful approximate solutions to problems with large search spaces [10]. Indeed a key challenge of GA is to avoid solutions becoming stuck in a local optima, it is for this reason that mutation techniques are used in population generation to aim to increase diversity and led to an eventual break out of local optima towards a globally optimal solution [21].

1.5.1 Core functional components of genetic programming

The following is a high-level overview of the core components of a GP algorithm [25]:

- Genetic data representation of a solution aka. chromosome (i.e. encoding).
- Fitness function to evaluation each solution against a goal.
- Selection function to select the best solutions from a generation.
- Crossover function to crossover some stands of the genome of a chromosome.
- Mutation function to randomly mutate some bits of the genome - aiming to prevent the solution from getting stuck in a local maxima by providing population diversity.

There are many choices to be made for a specific implementation of GP, such as: selection of the fitness function, selection of the crossover function, application of mutation (what type of mutations and on what portion of the population), and choices around data representation and encoding. A high level overview of these practical choices is shown in figure 1.1[25], which categorises these choices as Encoding, Selection, Crossover and Mutation (see figure 3.3).

Input:
Population Size, n
Maximum number of iterations, MAX
Output:
Global best solution, Y_{bt}

begin
Generate initial population of n chromosomes $Y_i \ (i = 1, 2, ..., n)$
Set iteration counter $t = 0$
Compute the fitness value of each chromosomes
while $(t < MAX)$
Select a pair of chromosomes from initial population based on fitness
Apply crossover operation on selected pair with crossover probability
Apply mutation on the offspring with mutation probability
Replace old population with newly generated population
Increment the current iteration t by 1.
end while
return the best solution, Y_{bt}
end

Figure 1.1: Genetic programming concept, source [25]

A property of GA is that domain knowledge of the problem is typically required for successful implementation, due to the fact that every fitness function uniquely maps the specific problem at hand. Further, each of the decisions around algorthmic structure (such as what type of crossover and mutation operations to use) are directed by factors of the data such volume and cardinality. Whilst this is also true for hyperparameter selection in DNN models such as learning rate, decay and activation function selection; in these applications these choices are generally more generic and as such best practices (or at least 'rules of thumbs') have been established both in academia

and in industry. The same is not true for GA and as such it offers less of a generic black box solution. This means that GA offers inherent explainability, and could be classified as a 'human-in-the-loop' ML method, due requirement of the fitness function to map to a desired solution to the problem statement. The trade off here is that GA naturally lacks the sophistication of multi-layer DNN architectures. There is a great importance on the implementation of the fitness function (e.g. human error and biases are not overcome by the use of GA, as they can be occurring in the formulation of the fitness function). Further research into the offspring of these approaches to offer a hybrid network solution [2] would be an interesting direction.

1.6 Software engineering principles

Farley [15] defines software engineering as 'the application of an empirical, scientific approach to finding efficient, economic solutions to practical problems in software.' In the domain of this research, the financial context has a distinct meaning as the system being designed has a direct financial output. This is an ethical factor that must always be considered when developing financial instruments. It is therefore of utmost importance that such systems are designed with rigour, well tested, and evaluated thoroughly before being trusted to trade.

A further consideration lies in managing complexity as the system grows. 'Complexity' here is an umbrella term encompassing features such as: system maintainability; ease of deployment; robustness; managing environments, and developer/release toil. Key to achieving our goals of maintainable, minimally complex software are tools such as automated testing and test-driven development (TDD), continuous integration and continuous deployment (CI/CD), system monitoring and observability and system modularity and service design [15]. In recognition of the utility of these well-defined criteria, this project will consider many of these principles in its design and implementation, and where relevant discuss their application in the methodology and evaluation sections of the report.

1.7 Aims and Objectives

This project aims to use TI-based methods in combination with GA in order to find novel combinations of technical indicators and indeed parameterisations thereof. The approach is to programmatically build and test trading strategies by combining indicators and analysing the relative performance of each.

This study will automatically generate traditional TI-based trading strategies and backtest these against historic Bitcoin data to assess the viability of non deep-learning based trading approaches and the success of the strategies. This problem can be expressed as one of large search-space permutation, and as such a genetic algorithm approach shall be adopted for strategy generation and evolution.

1.7.1 Objectives

1. Explore if technical indicator-based trading is a viable option for algorithmic trading of Bitcoin.

[2] Perhaps using multiple fitness functions encoded as hidden layers in a neural network, which takes input from a GA

2. Determine if genetic algorithms offer a means to the discovery of novel trading strategies.
3. Produce a fully functioning end-to-end genetic algorithm software system for this specific context.

The hypothesis is that TI-based algorithmic trading strategies are discoverable over an evolutionary process using GA, meaning that in general the TI-based method is appropriate for trading cryptocurrencies.

1.7.2 Value and Contributions

This research aims to contribute in the following areas:

- To aid in uncovering novel trading strategies that may lead to improved transferability vs DNN-originated methods.
- More generally, to speak to the question of whether TI trading methods are still valid given recent technological advances.
- Provide a python software package that can be reused by the community for further testing (if open sourced).
- Provide a containerised application that can be trivially deployed to allow ease of implementation.
- To provide insight into some of the specifics involved in creating genetic algorithms for trading purposes (such as data representation, use of fitness function, and mutation methods employed).

1.8 Summary

This section has provided an introduction to the field of algorithmic trading and discussed its implication in both the traditional financial and the nascent cryptocurrency market. We have looked at the factors that make cryptocurrency a suitable candidate for algorithmic trading and a poor candidate for human-instigated trades. More generally we have looked at the advantages offered by algorithmic trading techniques vs human-led trades such as the ability to trade at high frequency with complete consistency of decision and the lack of psychological interference. Technical analysis through the use of technical indicators was discussed as a means for human traders to provide a systematic approach to making trading decisions. However, even the use of technical indicator-led systematic trading does not abate these concerns. We discussed software engineering principles and their application to this project, and specifically the ethical considerations that have to be given to designing software with a direct financial outcome associated. Finally, we discussed the aims, objectives and contributions of the project.

Chapter 2

Literature Review

This chapter provides a review of the current and state-of-the-art as well as historic research of particular significance in shaping the landscape for algorithmic trading. The chapter has two areas of concern: machine and deep learning techniques and their application to algorithmic trading, and the genetic algorithmic approach to trading. The reasoning behind focusing on genetic algorithms is made apparent during the course of the chapter. As discussed in the introduction, algorithmic trading is a nebulous term with many subdomains, for the avoidance of doubt, the research analysed in this section focuses specifically on profit-seeking algorithms. Cryptocurrency-specific research is analysed where available, but in some cases, relevant studies are focusing on stocks and shares or Forex, and were particularly illuminating these have also been considered.

2.0.1 Machine and Deep Learning Algorithmic Trading

Commonly algorithmic trading is viewed as a classification task aiming to classify out-of-sample data points as a buy/hold/sell signal. An alternative approach is to view it as a regression problem in which we are tasked with predicting the next price given a sequence of previous price data. Pieces of research taking both or either of these approaches have been sought and discussed in this section.

A key challenge for both human and algorithmic participants in the market is that the signal generated from trading activities creates a continuous feedback loop. We can view the trading decisions made at a given timestep T as isomorphic to the mathematical factorial definition - the current state of the market at T encapsulates all information about the market at $T-1$ and so on; any trading decisions made do not exist in a vacuum, but rather, continuously influence the state of the market. In this way a virtuous/vicious[1] cycle is created, as trading decisions compound other trading decisions ad infinitum [37]. To make this more concrete consider the 'flash crashes' of the stock market in 2010 and the UK pound in 2016, both of which have been largely attributed to disparate algorithmic trading agents all responding to the same signals and creating a mass sell-off. The same trading agents then reacted to this selling pressure created by selling more, which led to a steep drop in the price of the respective assets [18]. The fact that high-frequency trading (HFT) agents are so pervasive means that in combination (and at scale) they have the power to influence the direction of the market, this makes traditional trading methods somewhat fraught as naturally human traders will be slower to respond to flash changes than HFTs [18].

[1] Depending on your perspective and trading goals.

Moreover, ML/DNN-based approaches may be responding to signals which are contradictory (or at least not supportive) of traditional technical indicator data consumed by human traders. In such situations human traders are in danger of being on the wrong end of a seismic shift (or a 'self-fulfilling prophecy') in the market instigated by machine participants [38]; highlighting one of the ethical challenges of HFT and algorithmic trading in general. To put it another way, each trader is not just a passive participant but rather has an active role in the future price of an asset. The role of any single trader is insignificant[2], but collectively can be influential on price direction as they react en masse to the same market signals[3]. Note that this has an interesting effect on assessing the 'correctness' of price predictions/trading decision-making at the algorithmic level, as the self-perpetuating classifications contribute towards artificially inflated/deflated prices. Interestingly it can be argued either way where the network effects of algorithmic participation in trading lead to a strengthening or weakening of the 'efficient market hypothesis'[4], as an asset's price at a time step T reflects its perception in the market through the lens of all market participants and therefore it represents its fair value, however to human traders the fact that an algorithmic flash cycle has begun will not be apparent until the price has already significantly moved and so this information is inaccessible to the underlying value of the asset at point of making a trade [18]. By extension of this logic, one could argue that there is far greater financial risk present in being a human trader than an algorithmic trader due to the fact that humans will always lag behind the ensemble of HFTs in the network driving prices in a certain direction.

The networks effects described by [18] are countered by the fact that in cryptocurrency markets prices are highly volatile (even 'chaotic' in nature [30] [35]) and as such hard to generalise (both for stochastic and deterministic, i.e. with rule-based, models) [37][29]. [29] implement a deep feed-forward neural network (DFFNN) to predict the price of Bitcoin at a given time and with remarkable accuracy. The DFFNN consisted of 5 input layers, and 3 hidden layers each of which mapped to 20 neurons, and was trained for 50 epochs [29]. Using the root mean squared error (RMSE) method of assessing accuracy, the study showed a peak model performance of RMSE=1.4406. The data was trained on data from 2016-2018, and during this time the price of Bitcoin was in the \$500 - \$700 USD price range, meaning that the RMSE on average erred by 0.0024% of the actual price of Bitcoin. These results are clearly very promising, however as previously noted the price of Bitcoin during this phase was relatively stable without large peaks or troughs that were observed in later market cycles. The authors did not extend the study to analyse out-of-sample data from later years (this would be an interesting further research piece to confirm the findings). These results highlight the inherent problem described by [37] that it is relatively easy (or at least possible) to create a model which performs well in a given market cycle, but proving that the model has not overfit is highly problematic [40]. The problem of overfitting and more generally, model stability, is prevalent in financial time series data, and particularly cryptocurrency market data due to the low 'signal to noise' ratio and the non-stationary nature of price data [40].

[37] counter this challenge in their approach by building a model based on the Deep Q-Network

[2]Unless they hold such a high proportion of an asset's total value that a sell-off would trigger a price drop, i.e. a 'whale' investor [31]

[3]Incidentally, a similar effect can be observed in recent times with so-called 'meme stock' trading where in a vast number of human retail traders are able to pool forces to influence the price of an asset artificially. This infamously occurred with the GameStop and Bed Bath and Beyond stocks. This type of market force has only now become powerful enough to influence price through the presence of internet-scale retail trading platforms, allowing millions of market participants to force a stock's price through collective buy/sell activities.

[4]The 'efficient market hypothesis' states that an asset's price represents all of the information available about an asset and therefore is a fair reflection of its intrinsic value. The implication of this is that there is no such thing as an under/overvalued asset as all asset prices are in fact a representation of all available data.

reinforcement learning approach, which has been highly successful in autonomous agent game playing. The authors assert that a deep reinforcement learning (DRL) approach is applicable to the domain of trading as DRL by nature encompasses the sequential interactions between agent and environment, and is therefore well suited to handle the trading decision/price feedback loop as discussed above [37]. Furthermore, as reinforcement learning is a reward-based approach to learning, it intuitively seems a logical fit for algorithmic trading as there is no need to create an artificial reward structure (i.e. the results of the trade/gain of account can be used).

The research acknowledges another key challenge present in algorithmic trading in general - that is the poor observability of the environment. Referring back to the efficient market hypothesis, consider that an asset's price at a time step is reflective of all data available about that asset. It is practically infeasible to encode all of this data to a trading agent, without creating a vast model pulling in both quantitative and qualitative data sources (e.g. news feeds, Twitter feeds etc.). Therefore trading decisions are at best made on partially complete information about an asset's state at a given time [37].

The study focused on stock market trading but contended that the results should generalise to other asset classes. The results were again encouraging but did not indicate that the DRL approach could consistently outperform the simple 'buy and hold' strategy over the same time period, and in some instances (e.g. for certain stocks), performed significantly worse than buy and hold. These results highlight the aforementioned difficulty in generalising stochastic-based approaches across multiple assets, even with the use of sophisticated neural networks. When considering the additional volatility present in cryptocurrency markets it is safe to assume that any such inconsistencies would be exaggerated.

Another study to use DRL for algorithmic trading is [31]. The authors follow a similar method as [37] (using the DQN algorithm as the DRL agent) and address the issue of noisy financial data in the market with the use of stacked denoising autoencoders (SDAEs) - an autoencoder which looks to remove noise by reducing the dimensionality of the data. The output layer of the network is a fully connected long-short-term memory (LSTM) layer - a recurrent neural network (RNN) with a temporal dimension and sequential awareness [9], which is able to assess recent and historical observations to provide classifications. The use of the LSTM layer is an intuitive choice as due to the cyclical nature of the trading to asset price interplay previously discussed, a model which is able to remember historic states, as well as recent inputs, appears somewhat isomorphic to the price evolution process for an asset. Additionally, RNNs have become the de-facto choice for many time series problems such as text generation and grammar correction problems in the NLP domain [9], and thus it presents a natural candidate choice for this time series problem. The study used stock market data from 2008-18 to train the model, and the results showed a consistent improved annual returns performance for agent-led trading vs the buy-and-hold strategy, suggesting that the LSTM and SDAE components of the network were impactful when combined with the DQN agent.

[24] evaluates the performance of DNN, LSTM, CNN (convolutional neural networks), SVM (support vector machines), ResNet, CRNN (convolutional recurrent neural network), and Ensemble models for predicting Bitcoin price. Using the mean absolute percentage error (MAPE) method for assessing the models, the authors found that DNNs performed the best when given a smaller number of previous windows as inputs for prediction ($m = 5$, i.e. five previous periods of data were used to make the prediction), whereas LSTMs performed well when $m \exists [10..20]$ and SVM performed best when looking back further over $m \exists [50..100]$ periods. For all models, the shorter time frame prediction ($m = 5$) performed best. This result highlights the inherent volatility in pricing Bitcoin data. Generally speaking one would expect greater accuracy of prediction when including more data

points to predict from, however in this case all models performed better with a shorter prediction window to consider. The authors offer the explanation that across the data set the ratio of interday price shifts of $\geq 5\%$ is roughly 17%, meaning that on any single day is there a greater chance of prediction accuracy when considering only the most recent temporal price data, as generally, large jumps in price were rare. This conclusion may hold valid for this study, but it again underlines the challenges previously discussed regarding non-stationary price data drastically affecting model performance and transferability [40]. Further research should be conducted to establish whether this finding generalises to different time periods and market conditions. Another intuitive for these finding is that when training on a long time period market volatility and will tend to average out and smooth when viewed holistically, and the dynamics of market cycles may be removed. Training multiple short time frame models in an ensemble is an interesting area for potential further research.

[9] also use an LSTM based approach to predict bitcoin prices. They analysed Bitcoin price data from 2016-21. This is a positive of this study as the data encompasses multiple bull/bear cycles and highly volatile periods of price movement, rather than producing a model which lacks transferability to different market conditions. The results were striking as during the 2016 market cycle when the price of bitcoin was relatively stable the LTSM performed consistently well with minimal loss (measured by the mean squared error rate), however as the market reached later cycles the predictions varied wildly and become increasingly more spiking in nature (i.e. price prediction could jump or fall drastically between periods rather than staying within the 5% of the previous period offered by [24]), and thus the error rate also increased. This is very informative as it presents perfectly the challenge of transferability of models which have performed demonstrably well in certain market conditions in unpredictable time series financial data.

[36] use an auto-encoder Deep Belief Network (DBN) to predict Bitcoin prices. The research provides a method for predicting a 'look ahead' price over a set number of discrete time steps, and comparing this with a 'look back' price over the same number of steps (starting from the same base). The delta between these points then gives an indication and weight for a buy/sell decision (e.g. a negative value would indicate a downward trend and therefore to sell, and the magnitude would map to how much to buy/sell). The authors propose a method of retraining the system on recent data on daily intervals to provide a more adaptive model, however, the research presented is only theoretical and no experimental results are provided, and thus we cannot judge the applicability of the suggested implementation.

The Chaotic Neuro-oscillatory multi-agent financial prediction and trading system (COSMOS) proposed by [30] is the first of the papers analysed to use a supervised learning approach as part of its architecture. A Feedforward Back-Propagation Neural Network (FBPNN) based supervised network is used in combination with the 'Lee-oscillator' as the input layer to provide a time series price prediction model. The architecture then uses a DRL model layer in its next phase, with a separation of concerns between the layers such that the FBPNN provides a price prediction and buy/sell classification, and the unsupervised DRL layer provides a weighting for how much to buy/sell given this input (i.e. it provides some strategy optimisation). The model was tested on a range of financial assets from cryptocurrency, Forex and traditional stocks and shares.

The authors choose to compare their model performance to simpler DNN strategies (isolated SVN, FFBPN, and PCA-based architectures), the buy and hold benchmark, and the top 15% and 5% of human day traders active on the 'Quantum Finance Forecast Center' website (a community of professional traders and researchers [30]). The method consistently outperformed all but the top 5% percent of traders according to the experimental results. However, these results should be taken with a pinch of salt, as a) the buy-and-hold data was presented as a monthly figure, meaning

that cumulative gains for holding an asset for multiple months were negated in the assessment (e.g. there was an assumption of rebuying the asset at its current price at the start of each period). Additionally, the comparison of the performance to the pool of traders is a fairly arbitrary comparison. The authors state that the QFFC website has 1000 active trading members, therefore the top 5 is a comparison between strategies of the top 50 traders (and indeed this is just a relative comparison, there is no indication as to whether these traders outperform the market). The comparison of performance vs human traders is a potentially interesting data point, however, to be a meaningful result the comparison pool would have to be much greater (e.g. by comparing to users of a much larger trading platform such as eToro, CoinBase or Binance). The availability of such data may be problematic, however, without it the results cannot be considered significant. In general, this critique speaks to a point raised in [37] that there is no conformity/consensus among researchers in the field as to how results should be evaluated, as they surmise:

> 'making a fair comparison between trading strategies is a challenging task, due to the lack of a common, well-established framework to properly evaluate their performance. Instead, the authors generally define their own framework with their evident bias.' [37]

Indeed, seeking to establish such an academic framework for the assessment of algorithmic trading would be a valuable contribution to future research.

[40] addresses the problem of feature selection in ML models with a double ensemble learning method. The authors express the problem that financial data models can contain in the order of thousands of features as inputs, and eliminating the noise from the signal is a complex challenge due to the volatility of the market. The proposed 'DoubleEnsemble' method looks to combat this by providing numerous sub-models to optimise feature selection to be used in the actual predictive model. The approach is to first train K sub-models using the input data for an asset, for each sub-model the loss curve and loss values are calculated; and used to produce learning trajectory-based weights from the losses; which in turn are fed into a feature selection function. Having iterated all of the K sub-models, the model thus achieves automated noise reduction and feature selection. The results are stated in terms of gain, Sharpe ratio, and using the statistical methods of precision, recall and the F1 measure. No comparison to other networks or trading strategies is offered, but the results show a peak F1 score of 77%. F1 is used as it provides the harmonic mean between precision and recall, that is it equally accounts for the precision (false negative rate) and recall (true positive rate). This is important in our domain as it encodes the earlier discussion in section 1 on striking an equilibrium between trading accuracy and frequency of trades.

A fascinating study was presented in [35] as it acknowledged all of the challenges discussed of noisy/chaotic data leading to highly unpredictably sequential time series and draws the conclusions that whilst ostensibly deep learning approaches may be best suited to predicting cryptocurrency prices, ultimately deep learning alone is not sufficient to solve the problem, due to the complexity of the problem. They assert that their results 'provide significant evidence that deep learning models are not able to solve this problem efficiently and effectively' [35]. The study uses CNN, LTSM and BiLSTM (a bi-directional LSTM model with both forward and backwards-looking LSTM memory cells) based architecture to predict the price of various cryptocurrencies over an 18-month period from 2018-19. The DNN-based approaches were evaluated against less sophisticated ML techniques and evaluated using RMSE and F1 as a measure of accuracy. The authors' hypothesis that the temporal RNNs models should outperform the simpler ML approaches was shown to be incorrect (at least against the data analysed) as both sets of models' performances were roughly equal, and the 3NN ML model actually achieved the best F1 score. Furthermore, a test for 'autocorrelation

in the residuals was performed (a test which looks at the difference between the predicted and actual results, with correlation indicating that the model failed to capture all available information to make a prediction). Based on these results the authors concluded that the DNN models were 'insufficient and unreliable' for cryptocurrency price prediction. This viewpoint was corroborated by [31] which surmises that many deep learning methods reviewed 'are not robust to the dynamic real market and can not be directly applied to algorithmic trading'.

The authors postulate that the reason for this surprising result could be that cryptocurrency prices follow a 'random walk' and as such and temporal-based models are in fact best to only consider recent samples (corroborating the findings of [24]). The authors' further assert that based on other research there is a strong possibility that technical indicator (TI) based strategies may be able to outperform DNN [22](cited in [35]). One interpretation of this conclusion is that traditional TIs naturally focus more on localised time windows as part of their and thus are less susceptible to overfitting to the noise. This lends credence to the continued use of technical indicators as part of an advanced trading strategy.

As previously mentioned, [22] takes a TI-based approach to predict the variance in the price of Bitcoin. The authors reframe the problem of predicting an exact price of the asset, to looking to predict the movement (i.e. percent change) in asset value. 124 TIs are included in the study and categorised as overlap study indicators, momentum indicators, cycle indicators, volatility indicators, and pattern recognition indicators. Using these indicators as input a decision tree (DT) model is built to classify the expected price movement into one of 21 discrete bands (e.g. $P\exists[-100\%..-11\%]$ being the band of sharpest price drop and $P \geq 11\%$ being the band of largest increase). The sample period used data between 2012-2017, and as such contained many boom and bust cycles. The constructed DT was shown to outperform the buy and hold strategy and thus suggest the continued relevance of Ti-based approaches to trading.

2.0.2 Technical Indicator and Genetic Programming approaches to algorithmic trading

This section reviews technical indicator-led approaches to algorithmic trading and the application of genetic algorithms to this domain. Investopedia defines technical indicators as

> 'heuristic or mathematical calculations based on the price, volume, or open interest of a security or contract used by traders who follow technical analysis'[7].

Using technical indicators human traders are able to define and backtest and execute trading strategies based on technical indicators crossing certain thresholds during a trading window. Technical indicators are built from data originating in a market, for example price data; volatility data (e.g. rate and direction of change); volume data (how many market participants and trading a given asset at a time step, and in what direction (buy/sell).

[14] was identified as a key study as it offered a comprehensive survey of cryptocurrency trading. Their research included papers covering a range of systems including trading platforms, systematic trading (including trading signals), trading strategy research and risk management. The authors note that there is a wealth of research into similar systems focused on stock markets and Forex trading, but that cryptocurrencies are entirely distinct from these asset classes both in nature (i.e. an entirely digital asset class) and market behaviour, and so established methods which have shown historical success may not apply to cryptocurrency. The authors identified technical analysis as a type of systematic trading, drawing a further distinction between real-time (i.e. high-frequency

trading) and other systems such as portfolio selection, pairs trading and arbitrage trading. Such systems are out of the scope of this investigation, but it is worth noting that there has been a considerable effort in these areas to date. In their review, the authors identify the presence of AI, ML and technical indicator-based approaches to trading. This indicates that although there is a strong move toward AI-based approaches, traditional techniques are still garnering academic attention; therefore consensus has not been reached on the methodological conceptualising of algorithmic trading strategies in cryptocurrency (recall the discussion at the end of the previous section with highlights that this is not the case in other subdomains of AI research such as NLP and imagine recognition).

[19] analyse the performance of using traditional technical indicators in bitcoin trading, and found that the so-called 'range breakout' rule consistently outperformed the buy and hold strategy (based on the Sharpe ratio - a measure of risk-weighted reward). This was the best performing strategy analysed as others were not able to beat buy and hold. The authors assess strategies from seven indicator categories. The study analysis 3084 daily price between 2010-2018. The authors' assessed the strength of strategies using the Sharpe ratio of executing the strategy and compared this to the buy and hold strategy. The results showed that the 'range breakout' strategy did outperform the buy and hold strategy across the full data set (p=0.06 for support/resistance bands based on 150 previous trading periods, to p=0.003 for bands based on 200 previous trading periods). This provides a relevant insight that technical indicators can perform better across long time periods (as they are less skewed by outliers when only using data from a shorter window). This is contrary to indentified research using DNNs, in which shorter timeframes generally performed better for prediction. The authors claim that the strategy also outperforms buy and hold in a subset of the data which was classified as trending, however analysis of the results shows varying degrees of statistical significance in different years (ranging from p=0.03 to 0.56). The study is relevant as it uses technical indicators traditionally used in stocks and Forex markets in the new asset class of cryptocurrency trading (specifically Bitcoin).

The study could be improved by combining multiple indicators to make more complex strategies. As the authors tested the Sharpe ratio resulting from trading just one indicator at a time, the results were not representative of a complex trading system. Whether such compound strategies would offer better results is unexplored by the authors, but represents a far more accurate representation of how traders (indeed both human and algorithmic) operate. This study aims to contribute to this question by exploring complex(multi-signal trading strategies).

[32] used multi-agent ensemble trained on technical indicators (TI) to trade Forex. Although this study focuses on Forex, it was analysed as the approach was to use TIs as input features to an ML model which is evolved via genetic programming. Specifically, a 'buy-stay-sell' Decision Tree (DT) was created using multiple technical indicators. The authors note that the population DTs created in the study create different trading signals not just based on the input TIs selected, but also with different parameterisations of the TIs and the trading windows to which they are applied. Highlighting the issue of strategy transferability across time frames. trading is 'non-stationary' and thus a successful strategy at time step x has no guarantee for success and time step y. Furthermore, parameter permutation can lead to vastly different strategies even from the same TIs. This highlights the challenge of generalisation of strategies for any trading scenario, heightened by the highly volatile cryptocurrency markets. The authors note that this challenge has led to further research such as retraining and incremental evolution.

The study highlights three specific challenges of GP to trading strategy construction these being: computational overhead, the potential for over/underfitting due to the non-stationary nature of

the data discussed above, and the challenge of population management and evolution of abstract trading strategies (that is what does it mean for a given strategy to be slightly mutated?). These are key considerations that are considered to during *this study*. The first of these considerations are addressed by reducing the functionality in TI and DT to counterbalance the increased computational overhead of GP, i.e. a so-called 'weak learner' approach. The second challenge is handled by re-triggering training once certain criteria are met. The authors use a voting mechanism to combine solutions from each population and avoid keeping a large number of results from different genetic populations (thus reducing CPU and memory overhead at runtime).

The genetic program (GP) uses Value, Moving average (MA), and Weighted Moving Average (WMA) as the TIs in its base population from which it evolves strategies. The output of the run for a given timeframe is a buy/hold/sell signal for each strategy. This signal is defined as:

$$IF(a \geq y)THEN(buy)ELSEIF(a \leq -y)THEN(sell)ELSE(hold) \tag{2.1}$$

where Y is a predefined constant parameterised with the model and a recommended action. The measure for how strongly the action recommended (i.e. a - Y) is a data point used in the GA's fitness function. This leads to the conclusion that although GP is inherently non-deterministic (i.e. there is no guarantee that subsequent runs to achieve the same result with the same population), the starting value for Y is of fundamental importance for this algorithm (which speaks to the more general challenge of parameter selection in ML).

The results showed that the best performing strategy had 82% profitable runs. The authors give the performance of the strategies in terms of a percentage point increase, rather than as an absolute profit or loss, therefore evaluating the applicability of the results is limited. The simplified training (i.e. the 'weak learner') didn't affect performance (either positively or negatively) but it did reduce training time, and so this was a successful approach aid to improving performance.

[26] is an older study from 2010 which focuses on evolutionary learning and TI for stock trading strategies. The study was selected as although it is somewhat dated in the field, it was considered very relevant for its application of TI and GP. In discussing the motivation for using an algorithmic trading system, the author acknowledges the effect of the interference of human psychology (i.e. FOMO). The author notes that historically TI or AI-based approaches have been used for defining trading strategies and discusses the history of GP in trading systems, which dates back to the early 2000s with reported success in stock and Forex trading.

In [32], a methodology of strategy generation from TI inputs using GP is presented to form strategies from 'linear combinations of classical technical indicators'. The author asserts that a benefit of such an approach vs AI-based approaches is that TI-based strategies are inherently more comprehensible to humans than AI-driven approaches which tend to be a black box. This is an interesting reflection as at the time of writing AI was still relatively nascent, however the concern of explainability was still prevalent. When considering that outside of academic discovery, the goal of a trading system is to profit in a capital market, the explainability of results is of pragmatic importance.

[32] discusses the challenge of non-static market conditions for generalised trading strategy formation, [26] also acknowledges this challenge by classifying the market state at the relevant time step as either up, down, or sideways and asses the generated strategies in each of these conditions.

The strategies were formed from the indicators ROC, SMA, WMA, EMA, MACD, Dynamic support/resistance indicator, stochastic momentum indicator, on balance volume indicator, and ease of movement value to generate buy/hold/sell signals. The author notes that the problem of investment signal discovery can be viewed as finding the best combination of finite-state machines

from a population of solutions enhanced by a genetic algorithm'. This is slightly contentious as by nature genetic algorithms incorporate some amount of randomness in evolving populations, and indeed the approach is non-deterministic (i.e. the same inputs can lead to entirely different results depending on the random mutations of the population that occur during the process). For this reason, we cannot conclude in absolute terms that a result obtained from a genetic algorithm is definitively the 'best combination', rather the most we can hope for is that is a well-performing solution that escapes local optima. This critique aside, the framing of the problem as a finite state machine is valuable insight.

The evolutionary algorithm generates a random population of strategies and assigns a rank across the population based on its Pareto dominance (that is whether the strategy performs at least as well as other strategies in the population) and sorts the solutions. Next, for some number of generations generate a new population and execute trades (based on the binary classification of the strategies) and then mutate the strategies. At the end of the generations the Pareto-optimal solutions are returned (that is solutions which are not Pareto dominated by any other solutions).

The results identified market momentum-based indicators with fixed percentage bands as the most informative signals by the GA during bear markets. The performance is evaluated against a buy and hold strategy for the same period and the GA achieved a loss of -6% against a buy and hold loss of -35%. In bull markets the GA had a trade direction prediction success of 60%, however the gain on the account was lower than the buy and hold strategy when considering transaction fees (12% vs 11% gain). This highlights the challenge of finding strategies which are successful not just in academic terms but also in practice, and indeed the underlying problem of market volatility for producing trading algorithms which produce generalisation strategies.

the research analysed in this section highlights that genetic algorithm approaches perform comparably to the AI-driven methods previously explored. This supports the analysis that DNN-based approaches may struggle with the chaotic nature of market data to find consistently tradable rules. That GA should be a suitable candidate for algorithmic trading seems intuitive as due to the vast number of parametrisations of numerous primary and secondary technical indicators across varying market conditions, strategy discovery in algorithmic trading can be framed as a combinatorial problem with a vast search space (similar to other problems in which it has been shown interesting results such as the travelling salesmen problem). This point is discussed further in the following sections.

Summary

This short literature review has provided a small cross-section of relevant research using cutting-edge deep learning and machine learning approaches to cryptocurrency trading and price prediction. The research analysed showed at times promising results, but in doing some also highlighted some of the key challenges around the aim of producing transferable prediction models in this domain; namely the unpredictable, highly volatile and chaotic pricing movements making any such generalisation highly ambitious given the data. Indeed, [35] asserted that any such attempts using DNN-based approaches are likely to be unreliable, and as such more complex ensemble methods should be sought. An interesting finding of the review is that whilst there are a high proportion of RNN-based approaches such as LTSM models(and indeed this model selection intuitively seems reasonable due to its applicability in other time series prediction domains), 'when comparing different ML models there is no clear and unambiguous winner' [9]. That this is the case adds fuel to the argument of [35] that DNNs alone are not able to solve this problem due to the complex and random nature of the

data. If DNNs were able to provide a clear solution one would expect to see academic convergence around a particular type of network architecture; as we have done in both the image classification (with ResNet and CNNs) and natural language processing domains (with LTSM and CNNs).

There were numerous pieces of corroborating research found that highlighted the fact that shorter windows seemingly outperform long time frames for price prediction, with the 'random walk' theory offered as an explanation. This speaks to the opinion that TI-supported approaches still have plenty to offer in the field of profit-seeking algorithms as they naturally tend to operate on shorter windows. This means that based on the literature review we cannot eliminate TIs as a valid means of constructing trading strategies and conversely DNN approaches offer no clear silver bullet. Secondarily we considered genetic technical indicator-based approaches to trading which used genetic algorithms in their conception of strategies. The research reviewed highlights that both traditional and AI-based approaches to trading have been implemented successfully, but that there has been no suggestion that using indicator market-derived data is becoming obsolete.

Chapter 3

Design and Methodology

This section discusses the design and implementation of the genetic trading algorithm in python as well as descriptions of the data that will be used to evaluate the trades.

3.1 Trading Strategy Composition

A trading strategy can be thought of as a collection of indicators and parameters which map to buy/hold/sell signals. For example, the Relative Strength Indicator (RSI) is an oscillator which provides an indicator towards the measure of the buying/selling pressures in the market[1]. $RSI \exists [0..10]$, and established wisdom suggests that an RSI value ≥ 70 is a sell signal. Formally we can define a strategy as a non-empty set of indicators, parameters and logical conjunctions:

$$\{I_i, P_i, L_i ... L_{n-1}, I_n, P_n\} \tag{3.1}$$

Where I represents a given indicator, P its parameterisation, and L a conjunction. For example, $BUY \iff RSI \leq 20 \wedge EMA[t] \geq EMA[t-1]$ is a strategy to buy only when RSI crosses a given threshold, and the exponential moving average for the time period hasn't dropped below the previous period (denoted by t, t-1). We shall expand on this definition later, but for now note that we have the building blocks of programmatically generating arbitrary trading strategies. Armed with these strategies we can test them against historical market data to simulate and asses what trades would have been made.

We can introduce some constraints to the formulation of the problem to reduce the search space, such as introducing a minimum/maximum number of indicators permitted, for e.g. a valid strategy is be any combination of indicators/thresholds/conjunctions so long as it has at least one conjunction and no more than three conjunctions, expressed using the Z notation as [39]:

$$strategies = \{i : Indicators, c : Conjunctions \mid 2 \leq \#i \geq 4 \wedge \#c = \#i - 1\} \tag{3.2}$$

Where 'Indicators' and 'Conjunctions' are the relevant sets thereof. The implicit logic here is that indicators necessarily require conjunction to be joined in a strategy, therefore the set of strategies generated by this comprehension contains a minimum of two and a maximum of four

[1] For brevity, and as they are well established in the literature, formal definitions of indicators are omitted from project

technical indicators (these parameters are arbitrary, there is nothing theoretically preventing us making intently large strategies). This preserves conventional wisdom as seasoned traders would consider trading a solitary indicator too crude a measure (lacking the sophistication to reliably generate profits in complex market conditions), and strategies with a large number of indicators will naturally present far fewer opportunities to trade.

There is a subtlety of strategy generation which has been so far ignored. That is our current model assumes that each conjunction is joining an indicator to another indicator (e.g. joining RSI to EMA via some logical operator). A more nuanced approach is to say that conjunctions join strategies to strategies. In this domain, conjunctions become far more powerful. For example, we could now generate strategies such as:

$$RSI \geq x \wedge (EMA \geq y \vee WMA \geq z) \tag{3.3}$$

In the above contains the following valid strategies:

- $RSI \geq x$
- $EMA \geq y$
- $WMA \geq z$
- $EMA \geq y \vee WMA \geq z$
- $RSI \geq x \wedge (EMA \geq y \vee WMA \geq z)$

Viewing strategies in this way creates a recursive type for the strategy similar to a LinkedList as each element contains a valid strategy at its head and is joined to another valid strategy at the tail (NB in the LinkedList datatype the join is a pointer to a reference of the next data location, whereas in our strategy object of the join is a logical conjunction). Such a definition enables arbitrarily complex strategy generation.

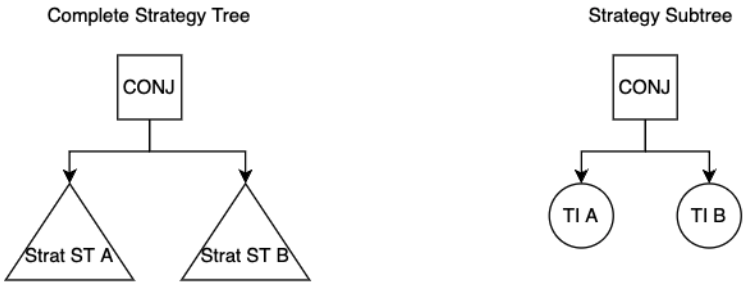

Figure 3.1: Strategy tree and subtree definition - on the left we see a representation of a complete strategy, that is two sub-strategies joined by a conjunction. On the right is a representation of a sub-strategy, that is two indicators joined by a conjunction.

3.2 Brute Force approach

Tasked with building and testing trading strategies, one approach might be a brute force method to make sets of strategies using in the above definition. naively generating strategies from a set of indicators, operators, thresholds and conjunctions.

The issue with this approach is the search space is vast even when limiting the number of indicators in the input set. It is worth considering well-known NP-hard problems with large search spaces such as the 'knapsack problem' or the 'Travelling Salesman'. When N is a very small number (e.g. ≤ 10) these problems are often solvable with some simple calculations. However, as N grows the problem becomes intractable using the brute force approach as the search space explodes.

In our case we are interested in generating strategies in order to test them against data to evaluate performance, and there is no constraint at the generation phase to rule certain strategies out (such a constraint does not exist as it is impossible to know if a strategy is useful for predicting B/H/S signals backtesting). Therefore, even when limiting the search space by enforcing the strategy formation constraints in the strategy definition, the brute force approach of strategy generation will yield an intractable number of strategies to test.

This is compounded further when we consider the choices for logical conjunctions (note that switching the 'AND' for an 'OR' in the example RSI/EMA strategy outlined above drastically changes the strategy).

3.3 Genetic Algorithms

As discussed the brute force method leads us into intractable territory, therefore the genetic algorithmic approach was selected to facilitate novel strategy generation. GA was considered a good choice due to the factors highlighted in the literature review, namely: similar studies had shown promise using the technique for trading and the hypothesis that DNN approaches may not be the best handle equipped to handle the volatility of the data[2].

Using the core components of GA indentified in section 1, the genetic algorithm for this specific project will :

1. Create Generation 0 - a random population of solutions (in our case well formed trading strategies).
2. Compute fitness for the solutions by back-testing the data against historic market data.
3. Assign probabilities for keeping a solution based on its fitness.
4. Select the best two solutions and perform the Crossover operation (creating two new solutions).
5. Move the new solutions to Generation 1.
6. Repeat until Generation 1 is full (minus n spaces)
7. Using 'elitism', fill the remaining spaces with the top n solutions from generation 0 (this prevents mutating out the strongest solutions).

[2]There is no reason to seek to preserve domain knowledge as dogma are other approaches offer demonstrable improvements, however in this domain and following the literature this was not felt evident.

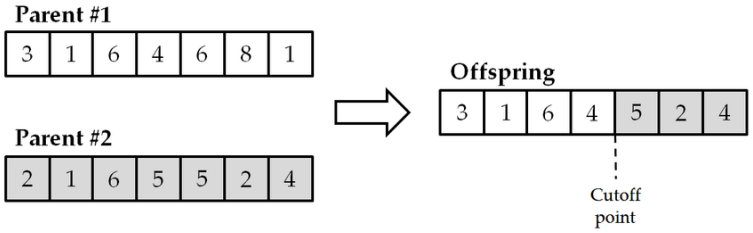

Figure 3.2: Single point crossover, source [17]

8. Mutation - Now that generation 1 is full, randomly switch some bits in the genomes to mutate the solutions.

9. Loop for a maximum number of generations, or until some precondition is met (e.g. fitness $\geq K$).

A prerequisite of this using this algorithm in our context is that we need the means to create well-formed strategies using our strategy definition. We cannot randomly generate a population of strategies if we do not have the ability to create a sub-strategy. In this regard, a brute force approach for initial strategy creation can be used to generate the initial population of strategies. Once the initial population is full, the genetic program should hone in well-performing strategies and poor strategies will be selected away from subsequent generations. The design and implementation of the genetic algorithm used are detailed in section 3.4.

Technical Indicators

There are many open source libraries in python for evaluating technical indicators given some raw price and volume data. This project used the open source 'TA' python library [34]. Other libraries were found with more indicators and with a high degree of optimisation, but the indicators implemented were already classified into 'oscillator', 'trend', 'volume' and 'volatility' meaning that the library would lend itself very well for adoption in our codebase. The library also provides a simple interface and good documentation meaning that there was a low barrier to entry to get it to a functional state within the code.

In total 39 indicators were used from the TA library as inputs to strategy generation. With 9 Volume indicators, 5 Volatility indicators, 14 Trend indicators, and 11 Momentum indicators. A complete list of the indicators available can be seen on the project's GitHub repository [34].

3.3.1 Data Representation

In order to be able to pass strategies into the GA and mutate bits of the strategy, we need to have a componentised representation of a strategy to form the basis of the mutations. Consider the single-point cross over operation, in which the head of one member of the population is split with the tail of another (see figure 3.2[17]).

In order for this to be achievable, a strategy (or rather our representation of a strategy in code) needs to support composition from subdivisions. Generally, this in GAs is achieved by creating a sparse matrix of features of the population [21], where the columns represent the population

f_1	f_2	f_3	...	f_n
1	1	0	...	0
0	1	1	...	1

Table 3.1: Encoded Population Representation

features and the rows the population members. This is demonstrated in table 3.1, in which the columns represent the genes and the rows the chromosomes of the population as a whole.

In our case, such an encoding is not appropriate as the population members have no fixed feature set (recall that strategies can have a variable number of indicators and conjunctions). Therefore a more sophisticated data model is required. To make this concrete, let us reuse our earlier example of a strategy:

$$BUY \iff RSI \leq 20 \wedge EMA[t] \geq EMA[t-1] \tag{3.4}$$

Unpacking the components we have the first indicator (RSI), its comparator ($\leq$) and its threshold (20), a conjunction (AND), a second indicator (EMA), its comparator ($\geq$), and its threshold (EMA[t-1]). If we split the strategy at the conjunction, both of the remaining parts are independently valid strategies (per our recursive definition of a strategy).

An additional complexity is that indicators do not necessarily operate on the same data type. Further, some use absolute values whereas others require relative context (e.g. the RSI strategy described uses the absolute threshold value 20 as RSI recall that RSI $\exists$ [0..100]; for EMA an absolute value would be nonsense as the average is by definition relative to the price of the asset. A desiderata is that the system should be able to create compound strategies from absolute and relative signals (indeed as a human trade might). Therefore we need a data representation for the strategy that is generic enough to handle either of these, and logic to handle both cases when dynamically creating the strategies at runtime. The following data schema was conceived as a means of doing this:

```
{
    "name": String,
    "absolute": Boolean,
    "op": String,
    "abs_value": Int,
    "rel_value": String
}
```

This is flexible enough to support both categories of indicators. Absolute value indicators are trivial to implement as they are just a mapping of an operator to a value. For example, we can represent $RSI \geq 70$ as follows:

```
{
    "name": "RSI",
    "absolute": true,
    "op": ">=",
    "abs_value": 70,
    "rel_value": null
}
```

The 'absolute' boolean signals whether the strategy should use the 'abs_value' or 'rel_value' value when assessing a trade. In order to handle relative value trades, additional logic is required to load the relative value at the point of backtesting. In this way the approach is analogous to lazy computation and 'just in time compilation': a strategy is loaded with a relative value indicator but not until the backtest is about to run (i.e. trades are made using the strategy for entry and exit signals) is the value mapping to the indicator known. This leads to a 'design by contract' API design wherein strategies can be formulated using the JSON schema described and the 'rel_value' is a enum mapping to a specific action server side to yield the value (hence design by contract, as the 'rel_value' string is not arbitrary). To make this more concrete, consider the following representation of $MA \geq price$:

```
{
    "name": "MA",
    "absolute": false,
    "op": ">=",
    "abs_value": null,
    "rel_value": "PRICE"
}
```

Here the 'PRICE' flag is used to instruct the server running the code that a price loading function has to be called. When the strategy is backtested the callback function would yield the moving average price for the assess the current trading window in scope.

The schema defined so far only handles sub-strategies, for the case of complex strategies we need to wrap this object inside another as follows:

```
{
    "indicators": [
        {
            "name": "MA",
            "absolute": false,
            "op": ">=",
            "abs_value": null,
            "rel_value": "PRICE"
        }
    ],
    "conjunctions": []
}
```

This schema allows us to represent arbitrarily complex strategies (as per our definition), providing that we observe the invariant that the $\#conjunctions = \#strategies - 1$, and that given an index i, $conjunctions[i]$ corresponds to $indicators[i], indicators[i+1]$. To provide a concrete strategy codified in this schema, consider the RSI/EMA strategy previously described in equation 3.4, which can be expressed as:

```
{
    "indicators": [
        {
            "name": "RSI",
            "absolute": true,
            "op": "<=",
```

```
            "abs_value": 20,
            "rel_value": null
        },
        {
            "name": "EMA",
            "absolute": false,
            "op": ">=",
            "abs_value": null,
            "rel_value": "PREVIOUS_PERIOD"
        }
    ],
    "conjunctions": ["AND"]
}
```

At runtime, the strategy construction engine would read the flag 'PREVIOUS_PERIOD' and use this to load the value of the EMA indicator for the previous window.

Using algorithm 1 and the above data representation enabled the generation of strategies to form an initial population. An example of such a random generated strategy is shown in 3.1:

Listing 3.1: Example generated strategy

```
{
        'indicators': [{
                'indicator': 'trend_vortex_ind_diff',
                'type': 'trend',
                'name': 'trend_vortex_ind_diff_gt',
                'absolute': False,
                'op': '<',
                'abs_value': None,
                'rel_value': 'MA'
        }, {
                'indicator': 'momentum_rsi',
                'type': 'momentum',
                'name': 'momentum_rsi_gt',
                'absolute': True,
                'op': '>=',
                'abs_value': 80,
                'rel_value': None
        }],
        'conjunctions': ['and']
}
```

Base indicator JSON data

The schema described above in section **??** was implemented as two JSON files containing representations of all of the 39 technical indicators and absolute and relative permutations of these (see: **??** and **??**. The absolute signals were manually created (as indicators have distinct value ranges[3]), and the relative indicators were generated programmatically with the python script 'gen-

[3]Illustrating point made in section 1 that GA often requires seeking and encoding additional domain knowledge.

erate_blank_indicators.py'. From these base indicators, complex strategy generation can occur by implementing the algorithm as described by algorithm 1.

3.3.2 Strategy generation

To handle our requirement to fill our initial population with well-formed strategies, algorithm 1 was designed. The algorithm somewhat naively combines indicators and conjunctions to produce valid strategies. The term 'somewhat naively' is relevant as during the design of the algorithm the following constraints were added to our definition of a strategy:

1. Each indicator can appear only once per strategy.
2. There is a limit to the number of indicators per indicator category for each strategy (oscillator, trend, volume, volatility).

The rationale for each of these choices is that without constraint 1 there is a possibility of complex strategies which are either tautologies or contradictions, e.g consider the following:

a. $RSI \geq 70 \vee RSI \leq 70$

b. $RSI \geq 70 \wedge RSI \leq 70$

a. is always True and b. is always False; there is no value in using either strategy in the population (theoretically they should be selected out by the genetic process, but the constraint helps the algorithm find meaningful results with fewer iterations).

The second constraint is more nuanced but also aims to reduce production (and backtesting) of provably redundant strategies. The reason to limit the number of strategies per category is that without this constraint we could create very narrowly focused complex strategies leading to contradictions. Some indicators are secondary technical indicators (i.e. built from other indicators), e.g RSI is derived from price data and Stochastic RSI is derived from RSI. Without constraint 2 the following would be a valid strategy:

$$BUY \iff RSI \geq 70 \wedge StochRSI \leq 0.2 \tag{3.5}$$

(n.b. that $0 \geq StochRSI \leq 1$). This is a valid strategy by our definition, however, it is a contradiction due to the fact that StochRSI is derived from RSI and as such will always be approximate $= RSI/100$. This is problematic as there is no straightforward logical constraint we can apply to guard against this (i.e. even greater domain knowledge outside of the scope of the algorithm is required). One possible solution is to create a map of compatible technical indicators and use this as the set from which to generate strategies. This would greatly increase the program complexity as at generation time each randomly selected indicator would have to be compatibility checked against all other indicators already generated in the strategy up to the current iteration of the generative process. To circumvent this complexity, (and indeed to place some of the computation back to the algorithm from the human)it was prudent to introduce the more rudimentary constraint of limiting the number of same class indicators and allowing the evolutionary process to eliminate any strategies which led to contradictions. There is a great trade-off here in that by introducing this constraint at the cost of removing false positives at generation time we have also eliminated the generation of any true positive strategies which violate this constraint. This speaks to the challenge

of balance type 1 and 2 errors in system design and whether the system has a natural preference for precision over recall (or vice-versa). Generally, for a trading system there is no cost of false negatives when backtesting and there is an opportunity cost[4] for false positives. A non-academic application of the system may benefit from removing this constraint, however in our case, it was a pragmatic choice to narrow the search space of strategy generation.

Strategy generation - Algothimic Design

The algorithm was codified in 'generate_strategy.py' and tested with TDD to ensure the correctness of the generated strategies. Implements the above constraints to ensure that no invalid strategies are generated by storing state including the number of indicators of each class present at each iteration of generation.

[4] i.e. the loss of potential gains that could have been made trading had the strategy been discovered

Algorithm 1 Strategy generation

Require: $MaxInds \geq 1$ ▷ Max indicators per strategy
  function CHOSEINDICTOR(indicators, M, S)
    $Ind \leftarrow random\_choice(indicators)$ ▷ Randomly select next indicator
    if $S[Ind.class] + 1 \leq M$ **then**
      return Ind
    else
      return CHOSEINDICTOR$(indicators, M, S)$
    end if
  end function
  ▷ Initial State
  $MaxClassInds \leftarrow 4$ ▷ maxSameClassIndicators
  $i \leftarrow 0$
  $IndTypeCounts \leftarrow Map < K, V >$ ▷ indicatorTypeCounts
  $Indicators \leftarrow lookup\_indicators()$ ▷ Retrieve stored indicators
  $Conjunctions \leftarrow lookup\_conjunctions()$ ▷ Retrieve stored conjunctions
  $N \leftarrow random\_choice(1, MaxInds)$ ▷ Randomly select the # of indicators
  $Strategy \leftarrow []$
  $BaseInd \leftarrow random\_choice(Indicators)$ ▷ Randomly select a base indicator
  $Indicators.pop(BaseInd)$ ▷ Remove selected indicator from set
  $Strategy \leftarrow Strategy :: [I]$ ▷ Append indicator to strategy
  $IndTypeCounts[BaseInd.type] \leftarrow IndTypeCounts[BaseInd.type] + 1$
  ▷ Generation
  while $i \leq MaxInds$ **do**
    if $i \leq N - 1$ **then**
      $Strategy \leftarrow Strategy :: [random\_choice(Conjunctions)]$ ▷ Append N-1 conjunctions
    end if
    $NextInd \leftarrow$ CHOSEINDICTATOR$(indicators, MaxClassInds, IndTypeCounts)$
    $Strategy \leftarrow Strategy :: [NextInd]$
    $Indicators.pop(NextInd)$ ▷ Remove selected indicator from set
    $IndTypeCounts[NextInd.type] \leftarrow IndTypeCounts[NextInd.type] + 1$
    $i \leftarrow i + 1$
  end while

3.3.3 Backtesting strategies

Backtesting is the process of testing a trading strategy using historical data to assess its performance. In this study the generated strategies were backtested against data downloaded from the centralised cryptocurrency trading platform, Binance. The data ranged from '2018-01-01' up to '2022-08-03', and all trades were made in the 15-minute time frame, meaning a total of 127182 records were used.

There are multiple open source backtesting libraries available in python which provide a means of creating custom strategies and are heavily optimised to allow for testing strategies in parallel (most notably pandas_TA and vectorBT libraries), however both of these options are not generic enough for the purposes of this study. Each of these has been designed with the assumption that the programmer will know at write time the strategy that is being tested, (i.e. both APIs do not easily support testing dynamically generated strategies as part of an iterative process as the

strategy components have to be hard coded). For this reason, the backtesting phase of the program has been manually created. The trading setup is listed below.

- All trades are using the 15-minute time window against Bitcoin (BTC).
- All trades buy $1000 with a target profit of 1.5% and risk/reward ratio of 2:1. Meaning that each trade stands to gain $15 and lose $7.5 USD.
- The account is started with an initial $10000 USD.
- Only 'long' trading positions (i.e. sells) are considered, short-selling is out of scope. So too are other sophisticated trading mechanisms such as trading leverage and margin trading.

Trading logic

There facilitate backtesting is a requirement to translate the raw trading data into an entry/exit signal given a strategy. Currently, the strategies generated are represented as JSON which we need to map against the CSV trading data. The open source Pandas library[5] was used to store representations of the trading data, using the following method:

1. Load pandas.DataFrame from CSV
2. Translate strategy to a parsed string representation to enable pandas filtering - i.e. map the strategy predicate every row in the DataFrame.
3. Find all rows where the strategy predicate is true - these are the strategy entry points across the full data set.

In our approach we can use the 'DataFrame.query' method to query an arbitrary string across the whole data set to subset the data, e.g. we can construct a query from the JSON representation of the strategy (see: section **??**) and use this to form the basis of a filter to query the data. This was implemented in the 'load_strategy' module. The function 'query_strategy' is the main entry point used to subset a 'df'(a Pandas DataFrame instance) of trading data where a given 'strategy' evaluates to True. We can iterate all strategies generated and use each as a separate predicate in this way to find all entry points for all strategies in the population. We can therefore define the entry points using set comprehension as:

$$entrypoints = \{s : S, d : D \mid s(d).d\} \tag{3.6}$$

Where S denotes the set of all strategies and D is the set of all trading data to be evaluated.

Evaluating Trades

Next, we need to calculate the profit/loss of a trade. Typically in 'real' trading scenarios, traders would establish exit signals for the trade as well as the entry predicate. These exit signals are either 'stop loss' signals which act to sell an asset if the trade is going in the wrong direction (to minimise losses), or 'take profit' signals. For example, a systematic trader may set their profit/stop loss

[5]Pandas is one of the most well know python libraries for data science applications and is established as an industry standard for processing columnar data in memory).

targets as ±2% of the buy price. These rules have been implemented in the 'genetic.py' module, setting a risk:reward ratio of 2:1, and a target profit per target of 1.5% percent. In practical terms this means that each trade is looking to gain 1.5% increase and if this target is hit the trade will be a successful exit. If the price fails to reach the target and is instead 'stopped out' when the stop loss price is hit the trade is unsuccessful.

Each trade is given a 24-hour window to execute the strategy (calculated from the timestamp of the entry point). From this window, we then find all exit points (that is points where the data breaches the stop loss or profit target thresholds). To assess whether the trade would have been successful, we then evaluate if the target was hit before the stop loss.

Once all strategies have been executed we can then evaluate the relative success of the strategy (using a fitness function), and continue with the genetic process using the best performing strategies in the population.

One drawback of this approach is that targets can be hit as part of an upward price movement, meaning that a trade may be complete before the asset hits its peak potential price. An alternative approach is to give each trade a 24-hour window to execute and assess the high and low price point hit and then evaluate whether on average the signal would create reasonable trades. In taking the more situational approach to trade execution it should be noted that we are essentially setting the risk appetite of the algorithm. In reality, the targets for gain and loss may well be adjusted to suit the trader. It should be acknowledged here that this decision is somewhat arbitrary; a supporting piece of further research to investigate target setting for a given strategy would be beneficial. The fact that an arbitrary choice has to be made is evocative of the critique raised in [37] that the field of algorithmic trading lacks academic standardisation for how the success of such systems should be evaluated.

Algorithm 2 shows the pseudocode for this algorithm.

Algorithm 2 Trade selection

```
function queryStrategy(df, Strategy)
    subset ← []
    for row in df do
        if Strategy(row) then                          ▷ Evaluate strategy predicate
            subset ← subset :: [row]                   ▷ Append rows which fulfill predicate
        end if
    end for
    return subset
end function
Strategies ← generateStrategies                        ▷ Previously generated strategy population
df ← loadData                                          ▷ All raw trading data
for Strategy in Strategies do
    subset ← queryStrategy(df, Strategy))
    for S in subset do
        window ← S.start : S.Start + 24hours           ▷ Select 24 hour window to trade in
        target ← df.open ∗ 1.015                       ▷ Set profit target to 1.5%
        stopLoss ← df.open ∗ (1 − 0.0075)              ▷ Set stop loss to - 0.75%
        targetHit ← queryStrategy(df, 'row.open >= target').   ▷ Find where target hit
        stoppedOut ← queryStrategy(df, 'row.open <= stopLoss')  ▷ Find where stopped out
        success ← targetHit[0] < stoppedOut[0]         ▷ Assess which occurred first - profit or loss
    end for
end for
```

3.4 Genetic Algorithm Design and Implemention

Algorithms 1 and 2 have detailed subprocesses of the overall genetic programming algorithm; that is so far we've identified how to naively generate a strategy and how to backtest that strategy against historic data to identify trades. The missing component is the genetic program itself. Given a population of strategies we wish to evolve the population over some number of generations to find an approximate optimal solution (there is no guarantee that the solution is optimal due to the inherent randomness of population mutation, and the fact that we will not exhaust the entire search space [10]). Figure 1.1 [25] depicts a generic genetic algorithm. The genetic algorithm implemented in this work shall replicate this algorithm where possible, with further specificity encoded were required.

The components of the GA that require context-specific are outlined in the general sense in section 1; figure 3.4 presents specific operators that can be used to implement a GA[6]. Each selection of these operators in this study and the rationale behind these choices is discussed in the section.

Encoding

The encoding is the primitive representation of the chromosomes which make up the population. Generally in GA this will be a sparse matrix/ one hot encoding type data representation in which

[6] An interesting piece of further research could be a meta-study into GA to produce novel genetic algorithms based on these operators for a given input and fitness function.

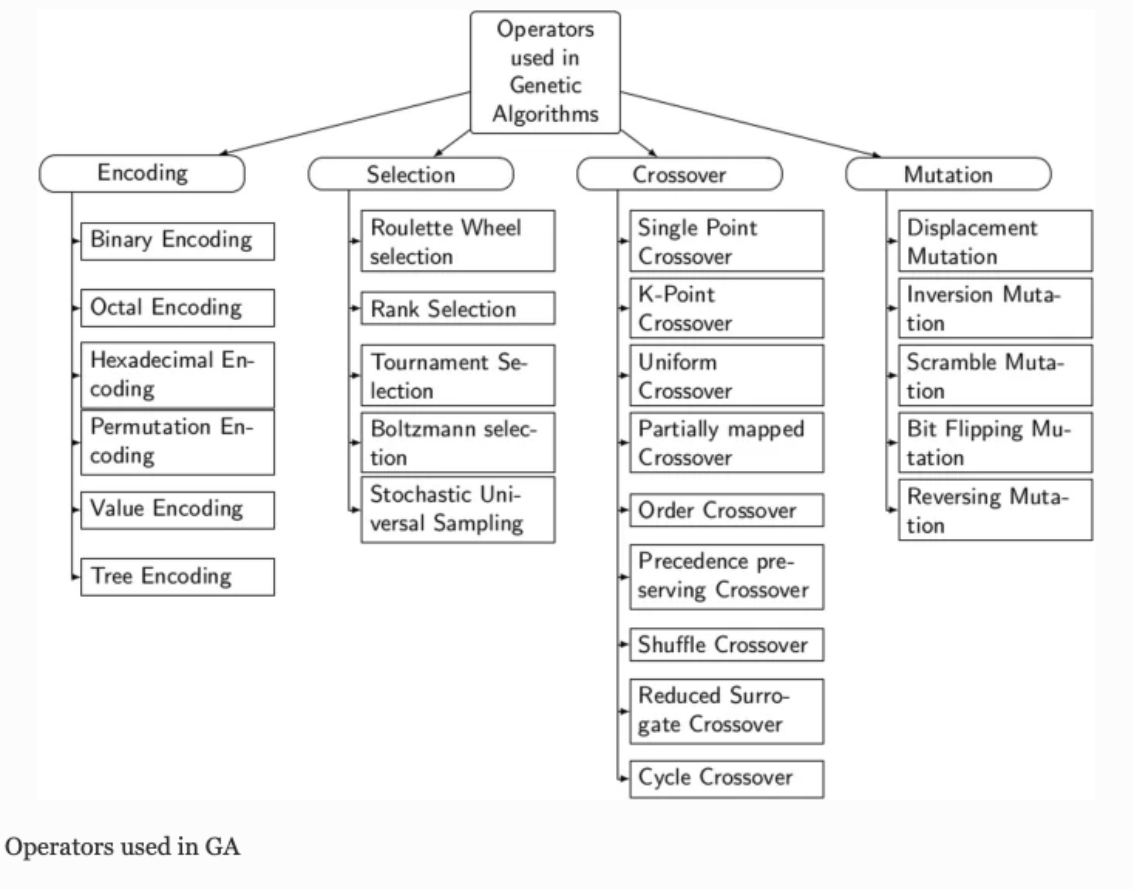

Figure 3.3: Genetic programming operators, source [25]

i_1	i_2	i_3	i_n	c_i	c_2	c_{n-1}
1	2	4	7	a	b	a
1	9	0	0	c	0	0

Table 3.2: Strategy Encoding Representation

bit maps to a specific gene and each non-zero bit maps to a value for that gene (refer to table 3.1 for a visual representation).

As described in section 3.3.1 a standard binary encoding to represent the strategies and their parametrisations is impractical in the context of this study due to the arbitrary (and non-uniform) structures of strategies. Some consideration was given to using a standardised encoding method such as binary or tree encoding as described by [25], however it was felt that this would be too rigid to represent the arbitrary nature of the strategies proposed (see discussion in section 3.3.1). One standardised encoding considered was to store an id for every indicator, and then strategies could be represented as show in table 3.2. In 3.2 the columns $i_1..i_n$ map to the n possible indicator that make up the max permitted indicators for a validate strategy (a constraint we previously set to 4 per algorithm 1). The columns $c_i..c_{n-1}$ map to the $n-1$ conjunctions required to make a validate strategy. In the first row we see an instance of a strategy with N indicators, and in the second row we see a simple strategy with two indicators. Note that in this data represent each column is a placeholder for a potential indicator/conjunction, a non-zero bit means that this place is filled whereas a zero bit effectively ends the strategy tree at its current depth.

Whilst this is a valid representation of the strategy which does support variable length chromosomes, it was decided that there was no substantial benefit to implementing this encoding as the strategies would have to convert back to a format that Pandas compatible format in order to enable querying the data (see the discussion on strategy predicts per algorithm 2). Generally, the purpose of such a strategy representation is to facilitate the mutation operations as part of the GA, however this was not deemed a requirement in this study (for reasons discussed below).

Selection

The purpose of selection in the GA is to choose chromosomes from the population which are most suited to the task based on a fitness function [25]. The selected chromosomes will be carried on to the next generated population either through elitism (meaning that the chromosomes are passed into the next round as an exact copy of their current state) or via mutation. There exist numerous selection methods, the simplest of which is the 'roulette wheel' selection method, wherein chromosomes are selected based on their fitness. The higher the fitness the higher the probability that the chromosome will be selected. To illustrate the point, picture spinning a wheel which has been divided into segments representing the population, and whose areas are proportional to the fitness, the segment that the spinner lands on is the selected chromosome. Better-performing solutions will be selected at a higher frequency when spinning the wheel as they occupy a larger surface area.

The issue with Roulette wheel selection is that that it can lead to an increase in solutions not escaping local optima across multiple generations [25]. In figure 3.4 'solution a' has a $\geq 50\%$) of being selected, and the top two solutions make up $> 86\%$). Whilst it is desirable that the best solutions should be picked most frequently, a criticism of this method is that the middle-of-the-pack solutions are all extremely unlikely to be selected, leading to limited diversity and an increased

Solution	Fitness
a	100
b	65
c	20
d	2
e	1
f	1
g	0.5
h	0.3
i	0.2
j	0.01

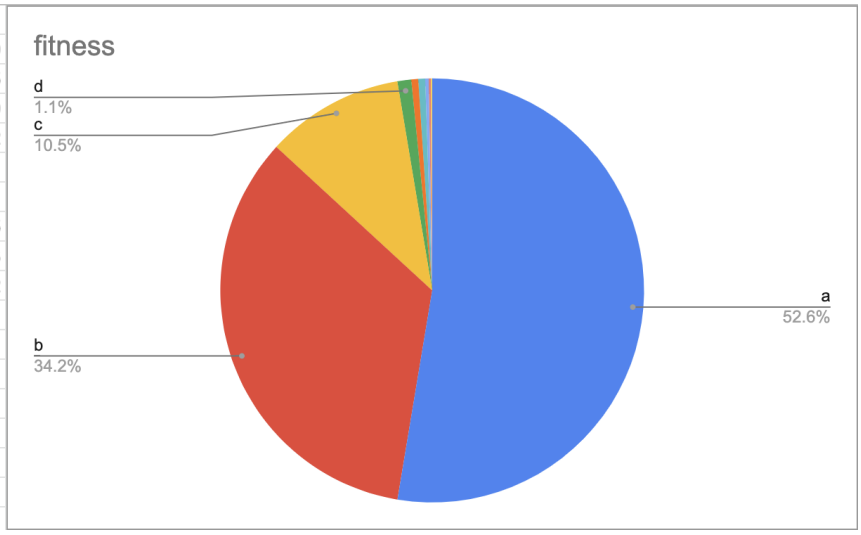

Figure 3.4: Example of Roulette Wheel selection

probability of not escaping local optima across the generations.

An alternative selection method is rank-based selection, where each chromosome is ranked based on its fitness and then assigned a selection probability $= N/r$ where N is the total number of chromosomes in the population. Figure 3.5 uses the same example data set as before but this time applies the ranking formula. Note that now a random selection has an increased probability of selecting a solution which isn't the dominant chromosome of the round. This leads to better diversity propagating through subsequent generations and hence a diminished probability of not escaping local optima across generations.

By contrasting figures 3.4 and 3.5, we can see the that probability of selecting a chromosome outside of the top two solutions based on fitness goes from $< 14\%$ to $> 50\%$, which underlines the benefit of ranked selection.

[25] highlight other possible selection techniques such as tournament, Boltzmann, and stochastic universal sampling, however ranked selection is generally accepted as a fine choice for most applications due to both its simplicity to implement and computational efficiency (n.b. there is a sorting operation required to rank the solutions based on their fitness, but this is still far lesser a computational overhead than some other selection-based methods which are $O(N * M)$ operations). For these reasons, it ranking selection was used in this project. This is implemented in the function 'apply_ranking' in the module 'genetic.py'.

Fitness Function

The purpose of the fitness function is to evaluate the suitability of a chromosome for a given task. In the Darwinian evolutionary model, it is the 'survival of the fittest' component of the process. [20] define a fitness function based on the 'attractiveness' of a given strategy where attractiveness is the profitability and statistical frequency of the strategy. A similar approach was taken in this study, whereby fitness was defined as $a(w * t)$, where a is the average % gain across all trades for

Solution	Fitness	Rank	Ranked fitness
a	100	1	9
b	65	2	4.5
c	20	3	3
d	2	4	2.25
e	1	5	1.8
f	1	5	1.8
g	0.5	6	1.5
h	0.3	7	1.285714286
i	0.2	8	1.125
j	0.01	9	1

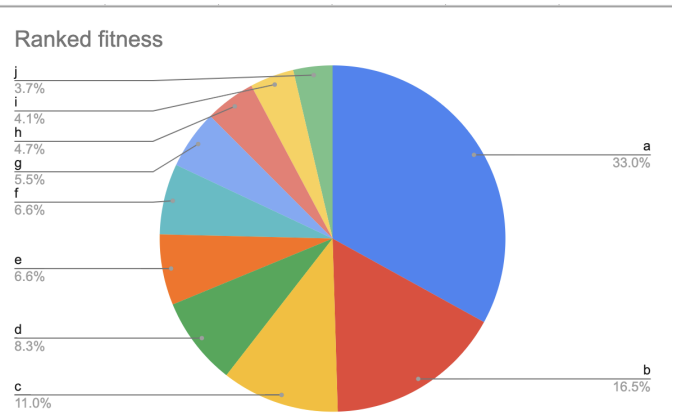

Figure 3.5: Example of Ranked selection

the strategy, w the win % (that is the number of successful trades the total number of trades) and t the total number of trades. The intention of the final term t was to reward strategies which traded more frequently. It is possible that a very complex strategy may only ever execute a single trade, but that this trade was profitable. If the fitness function was only considering win % then this strategy would the best strategy available (as it was 100% successful). This speaks to the point of balancing strategy complexity with the trading opportunities generated.

Selection

Once the strategies have been ranked, we can randomly select parents based on the ranking from which to create the offspring for the next generation. The first two chromosomes of the next generation will be the highest ranked two chromosomes from the current generation, and these will be placed in the population unchanged. This is the aptly named process of 'elitism', which ensures that we don't accidentally mutate away the key genes that made the top-ranked solutions so performant. That leaves $N - 2$ spaces to fill (where N is the population size). As depicted in 1.1, the next step for filling the population is to randomly select the parents and generate offspring from them until the population is full.

This simply samples at random from the current generation based on the ranking (see figure 3.5). An interesting design choice is whether to allow the same parental combinations to be selected more than once (i.e. for multiple offspring to be children of the same parents). It was felt that doing so was more likely to lead to the algorithm getting trapped in local optima, therefore steps were taken to ensure that each random sampling was from a distinct parent set. By distinct parent set, we mean that the selected parents have no offspring together (but either parent may have offspring with another parent), e.g. in the example shown in fig 3.6 given parent set A, parent set B is valid but parent set C is invalid (and thus would be dropped and re-selected).

The code for this is implemented in the functions 'generate_population' and 'select_parents' in the module 'genetic.py'. The implementation is fairly trivial, but one interesting feature to highlight is the use of a functional closure in 'select_parents' in order to track which parent have already been selected. Using a closure means that we can avoid passing in the parents as a variable to the

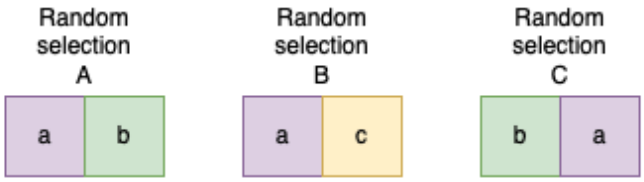

Figure 3.6: Random parent selection

function which enables us to keep the implementation functionally pure, i.e. we avoid having to modify a local variable to keep track of the parent state, as the state is handled by the persistent scope of the functional closure. This is shown in listing 3.3. The functional closure is implemented simply by setting the default value for 'parents' as an empty Dictionary in the function declaration. Due to the internals of functional implementation in python, this creates a closure which enables the function to remain pure.

Listing 3.2: Select parents function

```
def select_parents(
    ranked: dfa.DataFrame, _weights: Dict, parents: Dict = {}
) -> Tuple[dfa.DataFrame, dfa.DataFrame]:
    sampled = ranked.sample(2, weights=_weights.values())

    x, y = sampled.iloc[0], sampled.iloc[1]
    if (x.id, y.id) in parents:
        return select_parents(ranked, _weights, parents)
    parents[(x.id, y.id)] = 1

    return x, y
```

Crossover

Crossover is the process of taking a subset of the genes from multiple chromosomes and swapping them with one another to create a new solution from the parent chromosomes (i.e. an offspring). The simplest form of crossover operation is the single point crossover, wherein a random point is selected and anything to the right of that point is swapped from both chromosomes in scope (refer to figure 3.2). More complex solutions exist involving multiple crossover points or cycles [25].

There is an implementation challenge here in that as we have already established the strategies represented as chromosomes are less rigid than in a traditional GA (as they are of variable length). This means that the two parent chromosomes selected for crossover may be of entirely different lengths (i.e. have a different number of indicators/conjunctions in the strategy). For example, given the strategies A and B shown below the minimum length of these is two. Therefore any random integer between 1-2 would be safe to crossover, but any number > 2 would be invalid.

$$A = [a, b, c],\ B = [d, e] \tag{3.7}$$

To overcome this challenge, a version of 'precedence preservative crossover' (PPX) was implemented. PPX works by randomly generating a binary sequence of bits with each bit mapping to a parent chromosome, e.g a 0 might map to parent 0, and 1 map to parent 1. From this binary mapping, a child chromosome can be created by iterating the mapping and inserting the gene from the mapped parent at the relevant index [12][25]. If a gene already appears in the child, then the gene is pulled from the other parent instead. This is illustrated below in figure 3.7[12], where M represents the randomly generated binary mapping, F1, F2 the parent chromosomes and O the generated offspring.

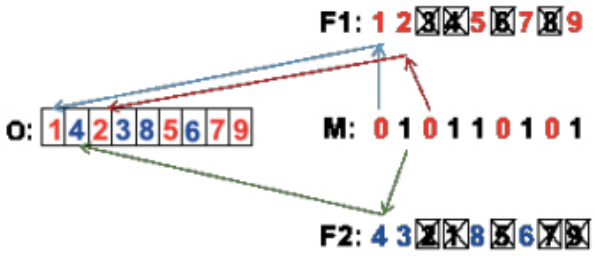

Figure 3.7: PPX selection, source [12]

The advantages of PPX are that it is computationally light (we only need to iterate the random string once, therefore $O(N)$), and it enables for purer offspring generation as the ordering of the parent elements are preserved before mutation is applied [25]. Both of these are relevant considerations for this project, however the main benefit offered was that it enabled mutation of variable length parent chromosomes. PPX handles this case well as we can simply choose the max length of the two parents and generate a child of this length, if the generated mapping doesn't exist in the selected parent then it logically must appear in the other parent. This is illustrated in figure 3.8 and was implemented in the function 'cross_over_ppx' in 'genetic.py'.[7] The python implementation is show in listing 3.3.

[7] 'cross_over_pmx' also appears in 'genetic.py'. Initially 'Partially matched crossover'[25] was intended to be used as the crossover operation, however after deeper exploration, it was decided that PPX was more flexible and better suited to the non-uniform nature of the chromosomes.

Listing 3.3: Select parents function

```
def cross_over_ppx(strat_x: Dict, strat_y: Dict) -> Dict:
    x_ind = strat_x["indicators"]
    y_ind = strat_y["indicators"]

    child_len = max(len(x_ind), len(y_ind))
    mapping = [random.choice([0, 1]) for _ in range(child_len)]
    mapped_inds = {0: (x_ind, y_ind), 1: (y_ind, x_ind)}

    offspring = []

    for ix, chromo in enumerate(mapping):
        primary, secondary = mapped_inds[chromo]
        try:
            p_ind = primary[ix]
        except IndexError:
            p_ind = secondary[ix]
        finally:
            if p_ind not in offspring:
                offspring.append(p_ind)
    return make_strategy_from_indicators(offspring)
```

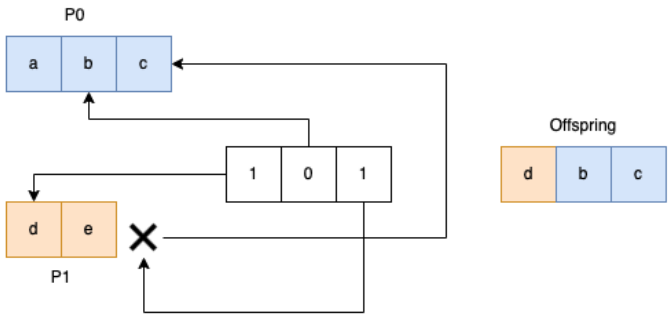

Figure 3.8: Custom PPX selection with uneven chromosomes

Mutation

Mutation is the process of randomly altering bits of a selected chromosome to create a new chromosome and exists as a way to further introduce diversity into population generation. A challenge of implementing mutation in the context of this domain is that there is no simple method for flipping bits in the chromosome representation in order to produce subtle variations [32]. For example, if

Listing 3.4: Mutation function

```
def mutate(strategy: List):
    _strat = deepcopy(strategy)
    abs_strats = [x for x in _strat["indicators"] if x["absolute"]]

    if abs_strats:
        mutate_ix = random.choice(range(len(abs_strats)))
        mutate_ind = abs_strats[mutate_ix]
        mutate_percent = random.choice(range(-20, 20)) / 100
        mutate_ind["abs_value"] = mutate_ind["abs_value"] * (1 + mutate_percent)

    return _strat
```

the binary encoding depicted in table 3.2 was used, then a mutation of a bit might have the effect of flipping a conjunction (e.g. swapping an 'AND' for an 'OR') or swapping the indicators that make up a strategy. These mutations fundamentally change the nature of the strategy. Whilst this may be an interesting experiment, it was decided prudent to avoid such complications and instead opt for a mutation strategy which mutates just the indicator thresholds by a randomly generated percentage in the range ±20%. The method for this is shown in algorithm 3. Due to the nature of relative indicator strategies not storing a value (i.e. the signal is relative to another strategy's signal the value of which unknown generation time) this approach is only applicable to absolute strategies.

Algorithm 3 Mutation

$strategy \leftarrow$ GENERATESTRATEGY ▷ base strategy
$\_strategy \leftarrow$ DEEPCOPY($strategy$)
$abs\_strategies \leftarrow$ FILTER($\_strategy, mutateStrat[absolute]$) ▷ Filter out relative strategies
$mutateIx \leftarrow$ RANDOM_CHOICE($len(abs\_strategies)$) ▷ Randomly select the strategy to mutate
$mutateStrat \leftarrow abs\_strategies[indicators][mutateIx]$ ▷ select the indicator to mutate
$mutatePercent \leftarrow$ RANDOM_CHOICE($range(-20, 20))/100$ ▷ Select a random in range +-20
$mutateStrat[abs\_value] \leftarrow mutateStrat[abs\_value] * 1 + mutatePercent$ ▷ absolute indicator

A nuanced but important step of algorithm 3 is the deep copy operation. Due to python's internal handling of data by reference, modifying an element of data structure (e.g. the item 'mutateStrat') updates the item in the original collection [4]. This is not desirable here as we may need the original strategy to be passed into the next generation of chromosomes unaltered (i.e. through elitism). By making a deep copy of the original, we are able to both mutate the indicator in the copied strategy and keep the original value. Note, that a deep, as opposed to a shallow copy, is important here as the item being copied is a python Dictionary (i.e. a key, value store similar to a HashMap) within a list, therefore a recursive copy of the full data structure is required. This incurs some extra time and space complexity costs, but it is in this case unavoidable. This algorithm is implemented in the function 'mutate' in the module 'genetic.py' and is shown in listing 3.4.

Confirmation of Genetic Process

The evolutionary process of the GA was manually tested over several runs of the algorithm parameterised with a smaller population size and a number of generations. Figure 3.9 shows a snippet of a pandas DataFrame of results. This process that generated this data had a population size of 6, therefore we should expect to see a general improvement of strategies every 6 runs.

```
In [5]: df.iloc[:12][['id', 'fitness']]
Out[5]:
                                     id  fitness
0   7575f321-3c05-4e69-b8ff-8038a517e642  17460.0
1   d9d3aa03-79ce-4d8b-8cfe-46288a73d3df   9097.5
2   15527c77-39ea-46e4-9d04-9d55e7865353     60.0
3   0d887190-acdc-410b-a02c-b1d1d366e467      0.0
4   8a2823fc-8e16-4f8d-91b2-8e170bafa1c5      0.0
5   d6e1ac47-cf9e-4ace-a80a-6e81dc49d989    -22.5
6   4f5c0291-0ddd-463b-ac84-71cf54e7ee3a  19357.5
7   7575f321-3c05-4e69-b8ff-8038a517e642  17460.0
8   d9d3aa03-79ce-4d8b-8cfe-46288a73d3df   9097.5
9   0a825795-eec7-4caf-9100-1c8e4e8fcf6e   7245.0
10  f0bdbda1-c13d-44bb-982d-6ba1ab70b9c1      0.0
11  3b321602-d3ee-4829-bafe-32fb24b1b92b      0.0
```

Figure 3.9: Sample evolutionary data. The blue line indicates two separate populations generated by the GA. The red boxed strategies are the best performing strategies in the first generation and therefore carried through to the second generation via elitism.

Summary

This section has provided an overview of the structure of a genetic algorithm in the general sense and detailed its application to our specific domain. Design choices for each of the core components of the algorithm have been discussed and justified, and where specific implementation details where required these have been provided.

3.5 Multiprocessing

An issue with genetic programming is that it can have a long run time given the requirement for a large number of iterations to evolve a solution. Multiprocessing can be used as a means of parallelising the backtesting of strategies. Due to the evolutionary nature of the GA, not all of the components of the algorithm are parallelizable; specifically, the ranking and regeneration of solutions can only be executed centrally[8] as by definition all chromosomes are required to be present to be ranked. This ranking enables selection and mutation and therefore these components of the algorithm are also not parallelizable. This approach can be characterise as the 'population parallel' [33]. Figure 3.10 demonstrates how parallelism was introduced to the GA.

Multiprocessing in python

When implementing multiprocessing in python, special attention needs to be paid to the 'Global Lock Interpreter' (GIL). The GIL is a central mutex in the C-python interpreter which provides

[8] An interesting study could use a map/reduce type architecture to distribute the entire algorithm across multiple nodes. In this distributed model local ranking would apply at the node level before the solutions were recombined in a shuffle operation (akin to a merge sort). Implementing this architecture is out of the scope of this project.

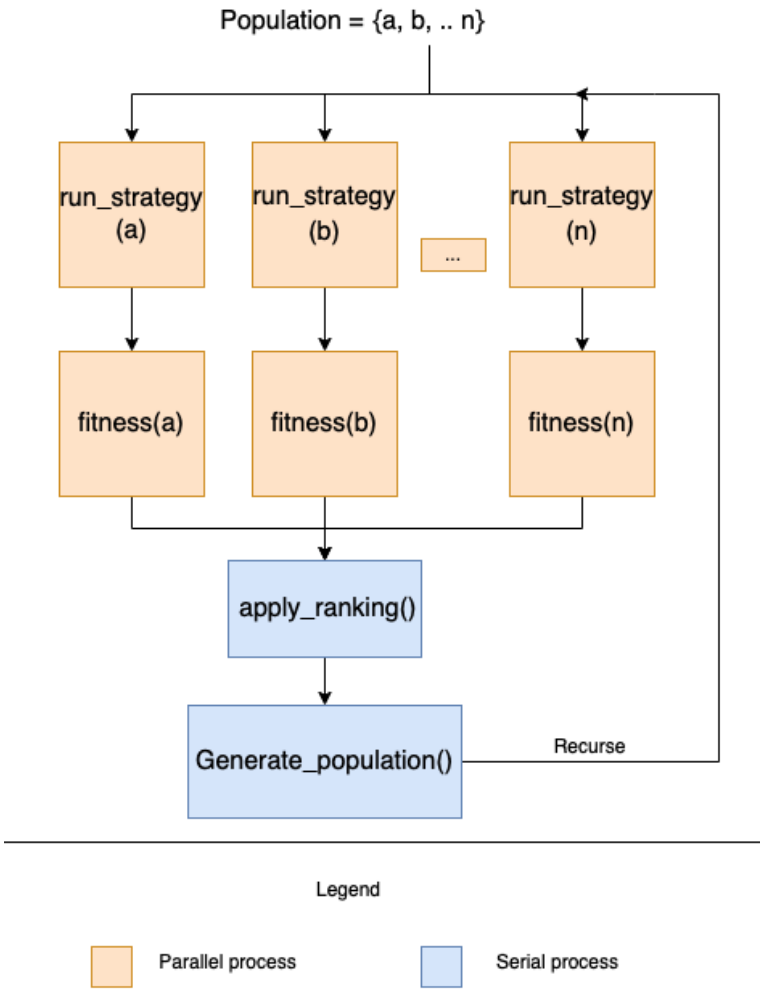

Figure 3.10: Multiprocessing workflow

locking to ensure thread safety at run time [6]. The low-level specifics of the GIL are out of the scope of this study, but the practical implication is that python is only able to execute one thread at a time. This means that multi-threading is only a viable option for programs which are I/O intensive rather than CPU intensive, as the mutex will block until the first CPU-intensive task releases it for the next thread to continue execution. To overcome this, multiprocessing is preferred for our solution as the backtesting component of the algorithm is CPU rather I/O intensive.

An alternative approach would have been to implement an API server which accepts requests containing strategies to backtest (as JSON blobs) (as per figure **??**). As sending multiple API requests is an I/O bound activity the code could implement multi-threading to parrallise the send of the requests, to the API service. This could increase the complexity of the system as of course the API requests to backtest each strategy would be long running asynchronous processes. A webhook architecture could be implemented in order to return the results. The client would have to keep track of all strategies sent and returned in order to know when to start the sequential stage of the process. This architecture is depicted in figure 3.11. Implementing such an architecture was out of the scope for this project.

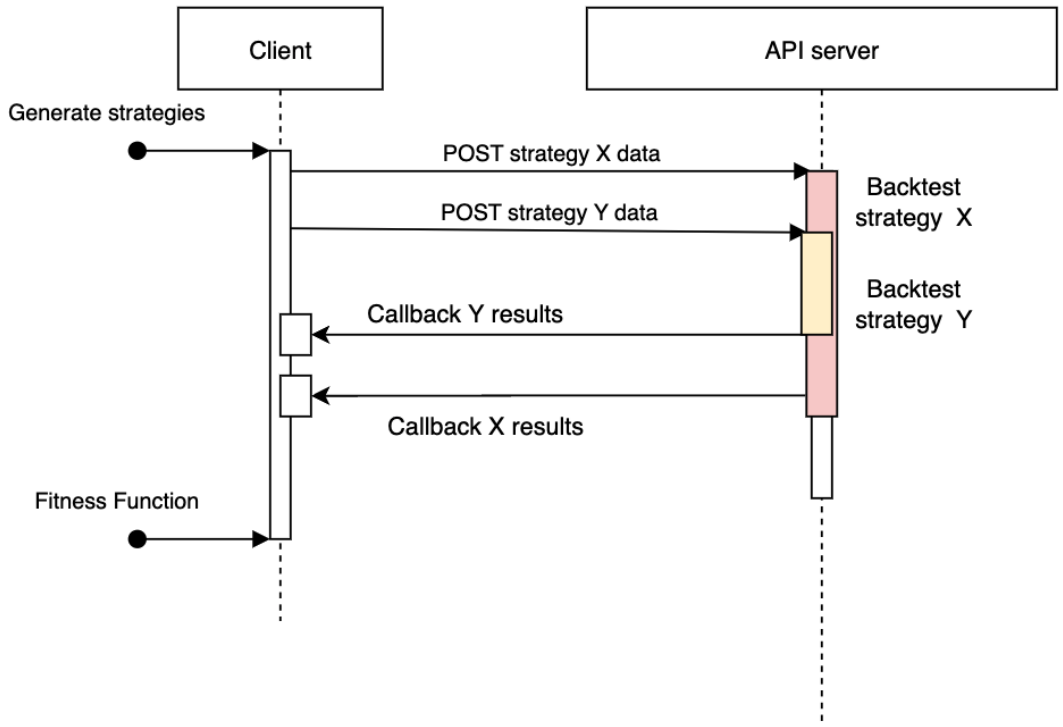

Figure 3.11: Sequence diagram for remote API processing

The multiprocessing capability was implemented in the module 'async_caller.py'. The module provides a thin wrapper around the built-in 'concurrent.futures' module (included in the python

standard library since version 3.2 [3]). The module provides an interface to map tasks to multiple processes/threads via the 'Executor.map' which maps the items of an iterable to a given callable [3]. Using this we might implement a multiprocessing add operation as follows in listing 3.5:

Listing 3.5: Trivial multiprocessing map example

```
from concurrent.futures import ProcessPoolExecutor

def add(a, b):
    return a + b

with ProcessPoolExecutor() as executor:
    result = executor.map(add, [[1,2], [2,3], [3,4]])
```

This is fine for many use cases, however in our case, we need to pass in the strategies to backtest along with the full set of trading data to trade evaluate the strategy predicate. This requires passing in multiple arguments to be mapped to the callable. One solution is to construct a list of iterables containing the strategy to be tested and the trading data, however due to the fact that the trading data is represented as a pandas DataFrame (which is itself an iterable object), this solution would not work in our case as the map method would attempt to map both iterables as the function arguments. This is illustrated in figure 3.12.

In section A of figure 3.12 we see the actual behaviour of passing in multiple iterables to a mapped executor, that is each item for both collections will be mapped to the function element-wise (where the notation k[i] represents the *ith* element for the collection k). In section B of the figure we see the desired behaviour, that is mapping the strategies to the 'run_strategy' function elementwise, but for each function call passing in a reference to the full trading data DataFrame. To achieve this behaviour, the function 'future_caller' constructs a 'partial function' - a metaprogramming construct through which a partially applied function mapping is created so that future calls to the function can provide the full argument list at runtime [5]. This enables us to achieve the desired behaviour illustrated in figure 3.12 B, by constructing a partial function with the trading_data as the partial argument and mapping the strategy data to this partial function. An abridged version of the implementation in code is shown in listing 3.6

Listing 3.6: Partial function application in multiprocessing map

```
# module: genetic.py
# function: main()
results = process_future_caller(
            run_strategy, strategies, trading_data)

# module: async_caller.py
from concurrent.futures import ProcessPoolExecutor

def process_future_caller(func, iterable, *constants):
    return future_caller(func, iterable, ProcessPoolExecutor, *constants)

def future_caller(func, iterable, Executor, *constants):
```

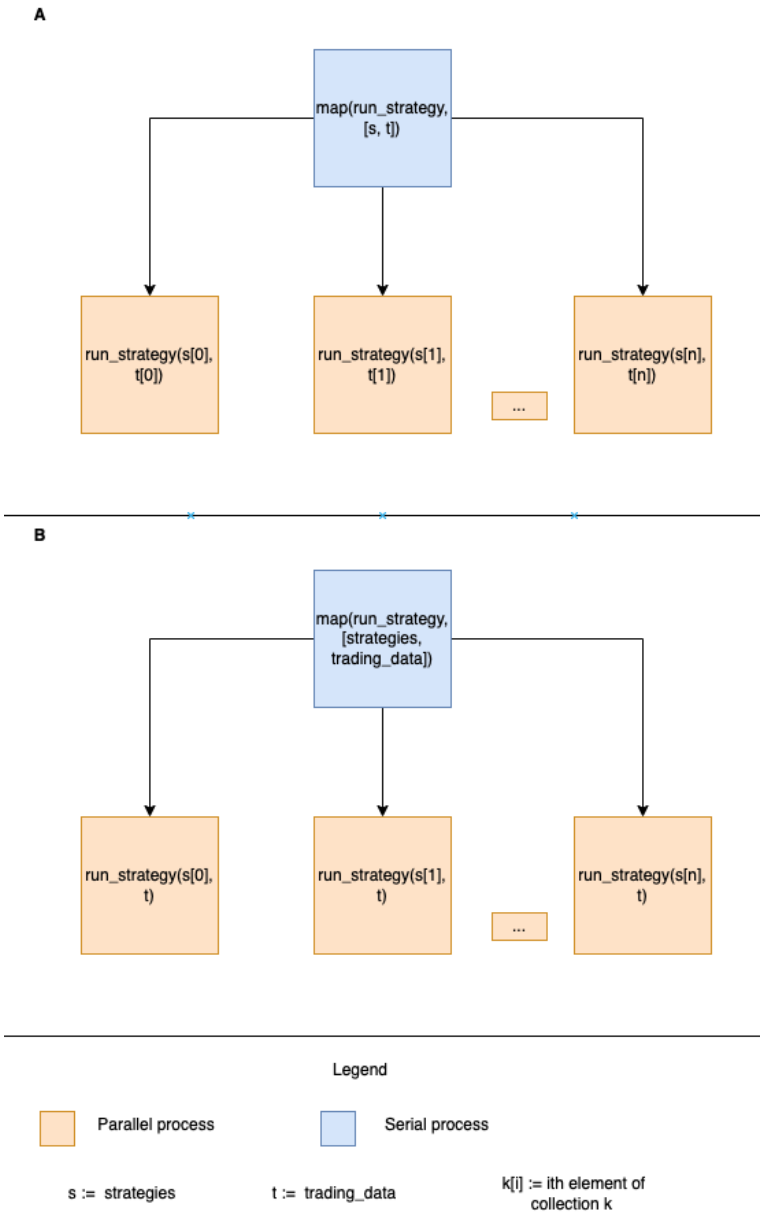

Figure 3.12: Multiprocessing map requirements

```
with Executor() as executor:
    fun = partial(func, *constants)
    results = executor.map(fun, iterable)
return tuple(results
```

Addressing point 1 is trivial as we can impose an additional constraint that disjunctive operators can only be applied for strategies with more > 2 indicators. Addressing point 2 is less trivial as the placement of the parenthesis can lead to a recurrence of point 1 if not well handled. Consider the a strategy of the $a \vee b \wedge c$. If we introduce a rule that any strategy including an *or* operator must be parenthesised, we could come up with the following parenthesis placement:

$$(a \vee b) \wedge c \tag{3.8}$$

$$a \vee (b \wedge c) \tag{3.9}$$

Note that eq. 3.9 is still invalid as it is logically equivalent to producing a single indicator strategy 'a' (as when 'a' is true the value of the second branch is not considered). Eq. 3.8 is valid as it is logically equivalent to producing two double indicator strategies $a \wedge c$, $b \wedge c$. Therefore when implementing parenthesis generation, a constraint is required that a disjunctive operator requires encapsulating in parenthesis. Note that this constraint holds on more complex strategies such as $a \vee b \wedge c \vee d$, $a \vee b \wedge c \wedge d$ etc.

Algorithm 4 Improved Strategy Generation

while $i \leq MaxIndicators$ **do**
  if $i \leq MaxIndicators - 1$ **then**
    if LENGTH(Indicators) == 2 **then**
      $Strategy \leftarrow Strategy :: AND$ ▷ Avoid OR operator dominance
    else
      $Strategy \leftarrow Strategy :: [random\_choice(Conjunctions)]$
    end if
  end if
  $NextInd \leftarrow$ CHOSEINDICTATOR$(indicators, MaxClassInd, IndictorTypeCounts)$
  $Strategy \leftarrow Strategy :: [L]$
  $Indicators.pop(NextInd)$ ▷ Remove selected indicator from set
  $IndictorTypeCounts[NextInd.type] \leftarrow ICT[NextInd.type] + 1$
  $i \leftarrow i + 1$
end while

Algorithm 4 shows the updated part of the generation algorithm; the initial setup of is unchanged from algorithm 1. The parenthesis generating component occurs after the strategies are generated during backtesting (recall the lazy computation used to assess trades described in section **??**). The logic finds instances of disjunctions in the strategy representation and performance string manipulation to add encapsulating parenthesis.

3.6 Summary

This section has provided a detailed view of many of the fundament components involved in designing and implementing the system. A precision definition of a trading strategy was provided to allow arbitrary strategy generation (with constraints). The strategy object was implemented in code as using JSON to allow dynamic generation of a population of strategies in the genetic algorithm. The components of the GA were discussed and the specific challenges of data encoding and mutation and how to overcome these were outlined. Three algorithms were presented for strategy generation, backtesting and the genetic process respectively. Finally challenges around the run time environment were evaluated and the measure taken to facilitated runtime parallelism through multi-processing in python were introduced.

Chapter 4

Evaluation of results

This chapter provides a presentation of experimental results and a discussion of their meaning, significance and whether they support or reject the hypothesis that TI-based trading is well suited to algorithmic trading of cryptocurrency. This chapter consists of two sections: preliminary experimentation and main experimentation.

4.0.1 Preliminary Experimentation

Having implemented the algorithms and multiprocessing patterns, an initial run of the GA was conducted to evaluate performance and identify areas for improvement. The code was run locally on an Apple Macbook 2021, M1 chip, with 32 GB RAM and a 10-core CPU. The GA was initialised with a population size of 10 and was left to run for 100 generations. In total, the run time was 21037 seconds (350 minutes).

Looking at the win percent and average gain of the strategies across the generations gives a good indication as to whether the GA is able to converge. Table 4.1 indicates that the GA failed to evolve strategies which lead to improved performance. We can see this by examining the standard deviation for both the win % and average gain %. For both of these, we see a very low standard deviation, indicating that there is insignificant movement from the average across the 100 generations. The 41% win rate and 0.25% average gain per trade show that the strategies generated do not perform notably well and there is no indication of any evolutionary progress. This is illustrated by the histograms in figures 4.2 - 4.3. In each histogram, we see a data point with between roughly 600 and 750 samples (of the total 1000 generations), falling at that point and very little distribution.

In an ideal situation, a greater standard deviation and distribution would indicate that over the course of the generations the evolutionary process led to improved performance (i.e. by starting wide and narrowing or honing in on a better result). The initial experimentation only ran for 100 generations, so we should not expect too much from these results in terms of absolute performance, but the fact that there was no significant improvement over the generations is enough to indicate that there are some improvements required to the GA.

mean win %	win % std dev	mean avg gain %	avg gain % std dev	mean fitness func	std dev fitness func
41.61	4.63	0.26	0.03	11098.30	3888.72

Table 4.1: Initial experimental results

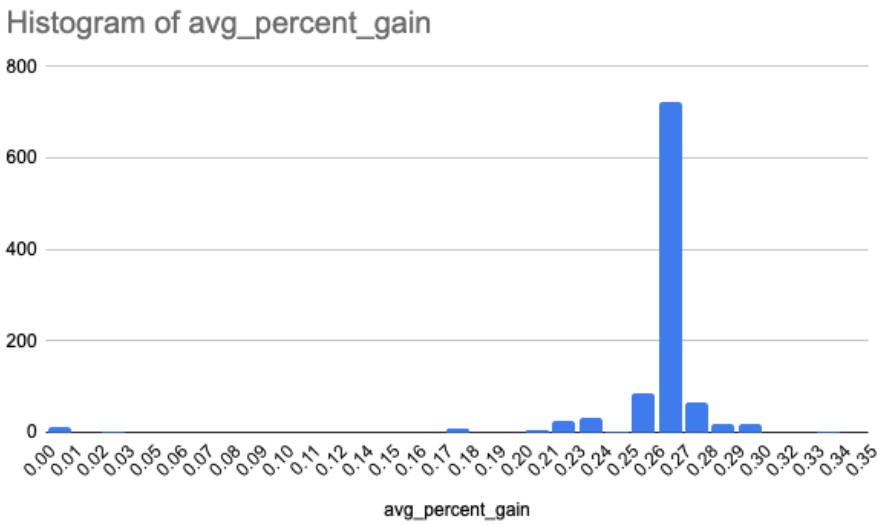

Figure 4.1: Average percent gain

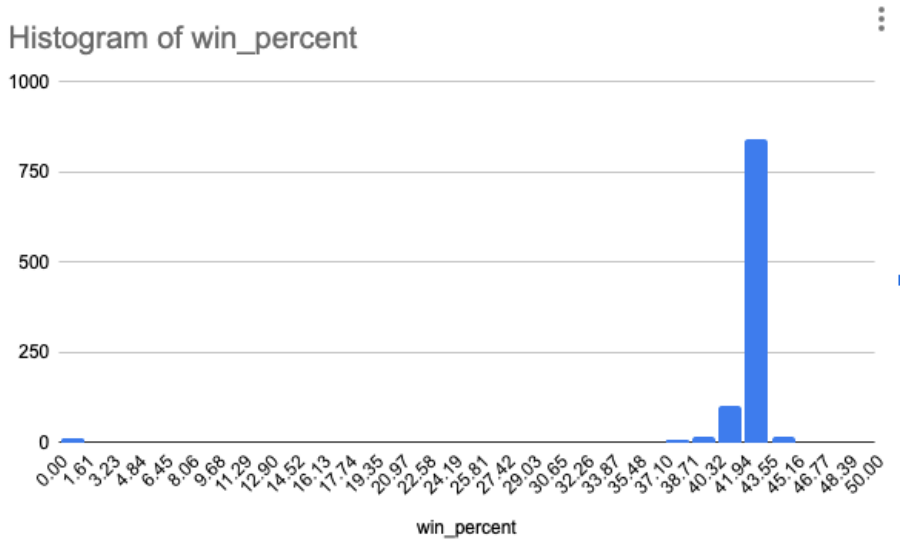

Figure 4.2: Win percent

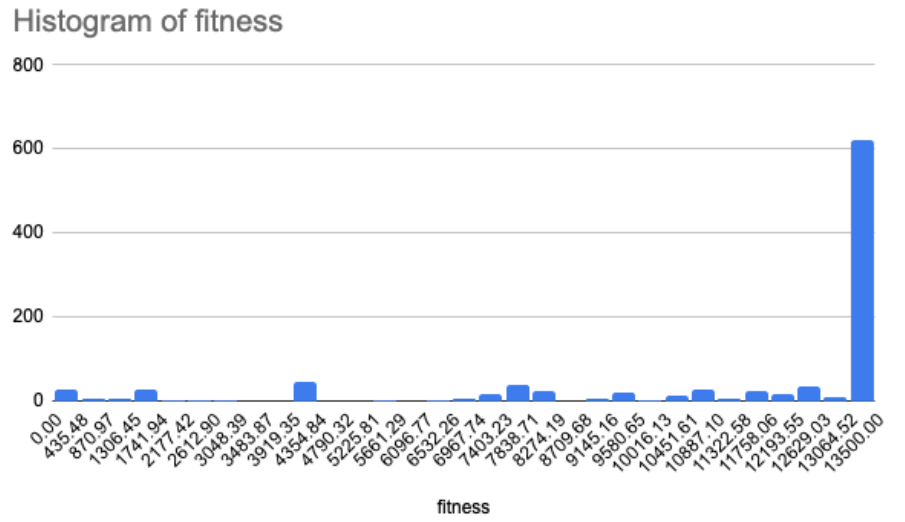

Figure 4.3: Fitness function

4.0.2 Improvements following initial results

From the analysis of the initial experimental results, the fitness function and strategy generation components of the GA were intentified for improvement.

Fitness Function

The fitness function is a key component of any GA as it acts as the natural selection operator in the Darwinian adaptive evolutionary model. The fact that no significant steps towards convergence occurred after 100 generations indicate that the fitness function was too loosely defined for the task. To address this the literature was reassessed to evaluate how similar studies had implemented fitness functions. In our study, the fitness function relied on percentage change (i.e. not accounting for the absolute price) whereas [20] used the asset price at a given timestep.

$$g(r)_{i,j} = \log \frac{p_c(i,\, j+k)}{p_c(i,\, j)}$$

Figure 4.4: Ha & Moon - Fitness function [20]

4.4 presents the fitness function used in [20], where r represents the strategy, i the asset, j the given time step at time step, k the number of trading steps, and $pc(i, j)$ the closing price for the asset at the time step. Using these inputs, the logarithmic gain is calculated $(g(r)i, j)$. One of the benefits of this approach is that it naturally rewards the strategies which trade more frequently as it includes the number of trading steps. This was the intention with the original fitness function design (hence using the %win rate * the number of trades), but the version implemented by [20] achieves this with greater elegance by taking the log of the result to prevent the dwarfing of well-performing chromosomes against lesser solutions. Therefore it was decided to re-implement the algorithm using a version of this fitness function.

A third fitness function was implemented which considered simply the profitability of the solution. This was calculated as:

$$W * win(x) + L * loss(x) \tag{4.1}$$

where W is the number of winning trades, L the number of losing trades $win(x)$ a function which returns $x * 1.015$ and $loss(x)$ returns $x * (1 - 0.0075)$ (i.e. a 2:1 profit loss ratio with a profit target of 1.5% per trade), where x is a value for the trade amount. In our case x was set to 1000, meaning that each trade stood to gain/lose 15/7.5 units respectively (note the currency is not important for this calculation).

this method doesn't take into account any compounding, i.e. each trade is given a new value for x every time, rather than using the outcome of the previous trade as the balance. A recursive compounding function of the profit/loss calculation was defined, however on inspection it was apparent that this approach was unsuitable as if several consecutive losses were hit the amount left to trade would reach a number immeasurably close to zero at an exponential rate. Conversely, following several successful trades, the balance would grow at an exponential rate, essentially becoming a whale trader whose participation would be strong enough to cause movements in the market in a real scenario. Therefore the simple version of the profit calculation was preferred.

Strategy Generation

Figure 4.5 highlights the issue with strategy generation. Here we can see that a 60% majority of the trading strategies traded > 121000 times (recall that there are 127182 rows in total in the data set).

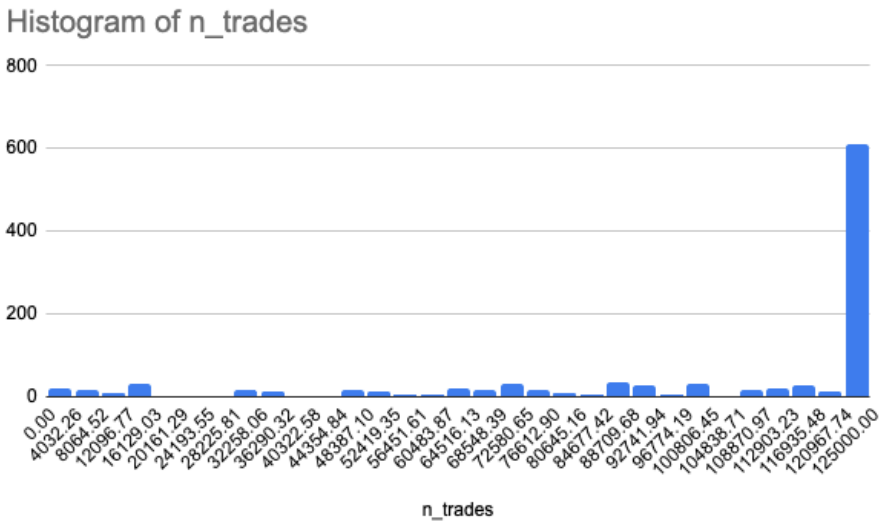

Figure 4.5: Number of trades per strategy

This indicates that the strategies generated cast too wide a net in which there was a very high probability that any given row would present an entry signal. Essentially, the strategies lacked the sharpness to discern between data points. This led to reviewing the data structure defined in 3.3.1, and the following areas for improvement were found:

1. Strategies with two indicators and a disjunctive effectively represent two single indicator strategies.
2. Strategies with multiple indicators were not chained with parenthesis, leading to sub-strategy dominance.
3. Removal of negated conjunctions.
4. Wide chromosome, shallow generations.

Point 1. is plain as given the proposition $a \vee b$ we know that if either a or b is True then the proposition is True. Therefore this form of strategy is invalid per our previous definition in which we stated that a strategy should have at least two conjunctions (this form of proposition effectively creates two strategies with no conjunctions). To rectify this, the strategy generation logic was updated to only use an 'OR' based conjunction if the number of randomly generated indicators was > 2. This is shown pictorially in tree representation in figure 4.6.

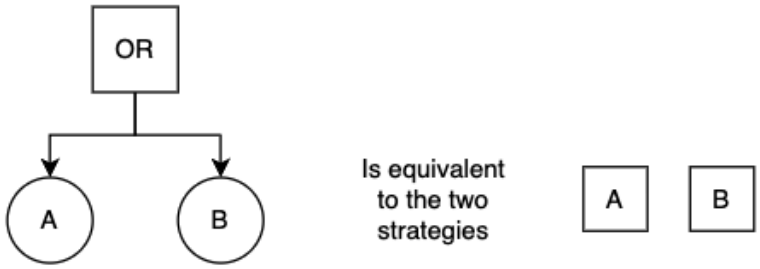

Figure 4.6: Logical strategy equivalence

Point 2 leads to a simplification of strategies which creates disjunctive dominance of the strategy as a whole. Consider a strategy of the form $a \wedge b \vee c$; upon querying the strategy in pandas using the 'DataFrame.query' method (see section 3.3.3) the final sub-strategy (in this case 'c') dominates the entire proposition. That is if 'c' is true then the entire proposition is true. Effectively, this complex proposition is equivalent to the two single propositions that arise as a result of point 1. This is illustrated below in 4.8, where a trivial example is constructed in the python interactive interpreter to highlight the effect of adding parenthesis to the query. This is shown pictorially in tree representation in figure 4.7.

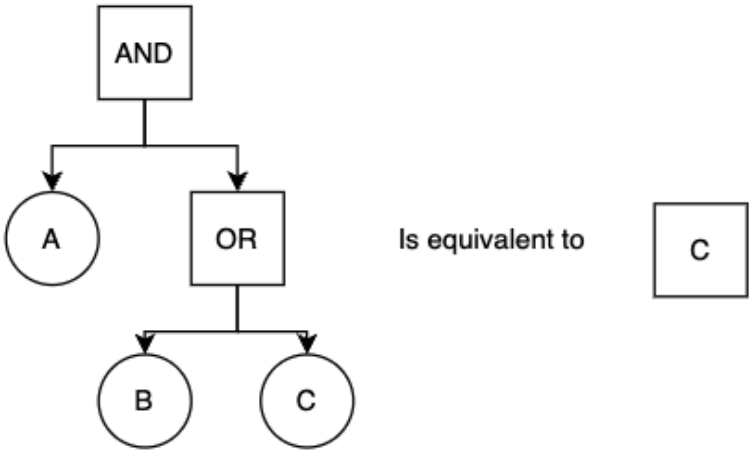

Figure 4.7: Logical strategy equivalence - disjunctive dominance

```
ipdb> df = pd.DataFrame([dict(a=1, b=2, c=3), dict(a=0,b=0, c=0)])
ipdb> df
   a  b  c
0  1  2  3
1  0  0  0
ipdb> df.query('a > 0 and b > 0 ')
   a  b  c
0  1  2  3
ipdb> df.query('a > 0 and b > 0 or a == 0')
   a  b  c
0  1  2  3
1  0  0  0
ipdb> df.query('a > 0 and (b > 0 or a == 0)')
   a  b  c
0  1  2  3
```

Figure 4.8: Example querying in pandas demonstrative disjunctive dominance

Point 3 (above) is about negated conjunctions such as 'not and' / 'not or'. Upon analysis of the initial set of results, a large proportion of the strategies containing negation were meaningless and given that mutation was only implemented at the indicator threshold level (per algorithm 3 and earlier discussion), the negation operators were removed from the strategy generation as there was limited benefit to this extra complexity.

Point 4 relates to the dimensions of the genetic population and its iterations. Initially, a small population size ($N = 10$) was run with the idea to evolve the population over a larger number of iterations. However, upon analysis, it was decided this was suboptimal due to the ranked-based selection method (in combination with elitism). Given that the two best-fit solutions would automatically be placed in the subsequent generation, and two offspring would be created from two randomly selected parents based on their fitness ranking, with a population size of ten 40% of each population is potentially dominated by the same two root chromosomes, and thus there is a high risk of the algorithm becoming trapped in local optima. By using a larger population size this risk is mitigated as the proportional representation of the carried-through chromosomes is diminished. Therefore a population size of 50 was chosen (meaning that 8%of the previous population was maintained either directly or via mutation).

4.0.3 Runtime Performance Analysis

Recall in the initial experimentation that the runtime on a single machine was approximately 350 minutes for 100 generations where the population size was 10, i.e. it takes roughly 5 hours to evaluate 1000 strategies. The reason for this slow performance is that for every strategy entry point (i.e. every instance for which the strategy predicate is true), we have to examine the entry point + 24 hours (i.e. the trading window) and find all trades that occur in that window. Given N strategies with M trading windows there are $N * M$ trading windows to evaluate (where $N =$ population size, and $M \leq$ total entry points in the data set). We then need to apply the fitness function for the strategy based on all trades found in the window (for each strategy). Possible solutions to improve the computation time are listed below:

1. Use caching to avoid re-evaluation of known strategies.

2. Use short-cutting methods to avoid evaluating strategies with a high percentage of entry points.
3. Further parallelise the code during backtesting.
4. Run on a larger machine with more CPUs (vertical scaling)
5. Use a cloud service provider to run the service on multiple machines concurrently (horizontal scaling).

Each of these points is discussed below.

1. Caching can be used for storing results for a given strategy. This way when a strategy is reused (i.e. after it is passed into the next generation via elitism), the performance can be retrieved rather than recalculated. the time savings are variable but could be significant for a prevalent trading strategy.
2. If a strategy predicate is true for $\geq K\%$ where K is a percentage of the total number of trades we can discount the strategy as it is likely that it identifies many false positives. From a data science perspective, we can say that such strategies are low precision and high recall. This equilibrium of precision vs recall is a common consideration for data-driven applications and the domain will guide the optimisation decision (i.e. does our system prefer type 1 or type 2 errors). In our domain, the implication of false positives is that trades will be entered erroneously leading to an increased percentage of losing trades (i.e. monetary loss). The implication of false negatives is that winning trades will be ignored resulting in a lower percentage of winning trades (i.e. opportunity loss). Optimising for fewer false positives comes with the performance benefit of allowing the algorithm to ignore strategies which identify a high percentage. The code was reimplemented with $k = 1/3$, as it is safe to assume that any strategy which identified more than a third of all possible 15 minutes time steps as trading entries over a four-year window is highly dubious.
3. The implementation of backtesting as discussed in section 3.3.3 is to set a window for the entry point + 24 hours for each entry point identified and find the trades within this window (iterating for all windows for all strategies, hence $N * M$). This iteration can be refactored to run in parallel using the previously described concurrent futures wrapper shown in listing 3.6. A limitation of this approach is that the code speedup achieved is limited by the CPUs available in the host machine. Given that python is not well suited for multi-threaded applications which are CPU bound (i.e. as opposed to I/O bound) due to the GIL implementation (discussed previously in section 3.5). This means that CPU-bound applications have be to parrallelised into multiple processes, each of which is bound to a CPU core. Therefore, the limitation of this approach is the physical number of cores in the CPU. Significant performance improvement is therefore only possible on a highly CPU-optimised machine with a large number of cores. This is known as vertical scaling, i.e. running the process on a single machine but investing in the performance capabilities of the machine to improve performance.
4. An alternative to vertical scaling is horizontal scaling - instead of increasing the performance capacity of a single node, we add multiple nodes to a cluster and add parallelism by distributing the workload. Horizontal scaling has been increasingly the preferred option in modern application architecture as it avoids the pitfalls of vertical scaling (moving to a 'servers as cattle, not pets' model [11]).

4.0.4 Adding Further Parallelism to the System

There is additional complexity around running and maintaining horizontally scaled applications once they are distributed. Factors such as network consistency and handling network quorum (i.e. which side of the CAP theorem the system prioritises), and additional factors such as resource allocation when connecting to external services. Of course, with a greater number of running services comes added complexity for managing ancillary non-functional tasks such as application monitoring. However, through containerisation, modern orchestration tools such as Kubernetes and infrastructure-as-code tooling have made managing complex cloud environments more attainable to applications of all sizes.

Generally speaking, horizontal scaling should be preferred to vertical scaling where possible due to the reasons highlighted of maintenance and eliminating single points of failure. Figure **??** provides a design for how such an architecture might be implemented. The implementation of this architecture was out of the scope of this project but would offer an interesting piece of further research to provide optimisation for genetic algorithmic system runtimes.

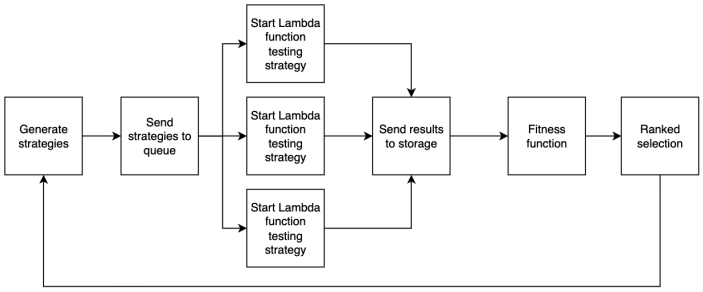

Figure 4.9: High level parallel architecture

The vertical scaling approach was followed by deploying the code to an 'r6i.4xlarge' AWS EC2 instance (a 16 core, 128 GB cloud server) [2]. To facilitate ease of deployment and maintenance, the application was containerised using Docker, and the infrastructure was provisioned using Terraform in line with modern engineering best practices to remove developer toil and automate where possible. Running a CI/CD pipeline was unnecessary for this project, but the fact that all elements of the infrastructure were scripted could facilitate a fully automated build process.

The docker and Terraform components are not discussed in great detail as they are declarative code to define the run time and offer little value to analyse further. The system used an EC2 instance which connected to S3 to save the results data. Figure 4.10 shows an example of the dockerised container running in the genetic algorithm in AWS.

To support running the application via docker, the main entry point to the application was written to support a high degree of customisation. A command line interface (CLI) was built which supported passing in variables to alter the program flow (e.g. by specifying the number of generations or where to save data to - see C). Additionally, the system was built in such as way that these parameters could be stored as environment variables in the system and read in at runtime. The system is now a fully portable containerised application; indeed anyone with access to the code should be able to start a running instance of the entire system with minimal effort.

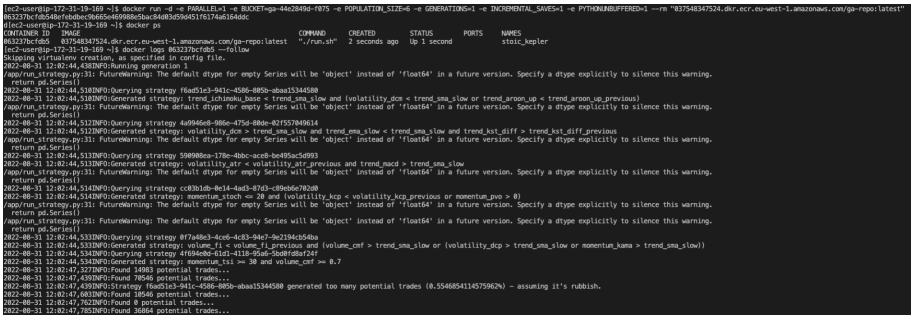

Figure 4.10: AWS Dockerised log data

4.0.5 Main Experimentation Results

In the main experiment, the genetic algorithm was run in AWS EC2, with a population size of 50 for 50 generations. i.e. 2500 trading strategies were tested in total (complete parametrisation available in appendix D).

4.0.6 Performance of Bitcoin

In order to assess the success of the genetic algorithm to find novel trading strategies, it is important to assess the performance of Bitcoin without taking into consideration any trades. At the start of the trading data (2018-12-15 03:00:00), Bitcoin closed at $3312.32. The final closing data point closed at $23392.52 USD (2022-08-03 20:00:00). This represents an increase of $\geq 600\%$. Therefore the simple buy and hold strategy for Bitcoin over the period starting for an account containing $10000 worth of Bitcoin, would be worth $60,600 USD. This represents an annualised return over the three years and eight months of 68%. It is these type of numbers that has led to the huge growth in the cryptocurrency market in recent years. It should also be noted that this is a relative trough in the time period as BTC reached a high of $68719.28.

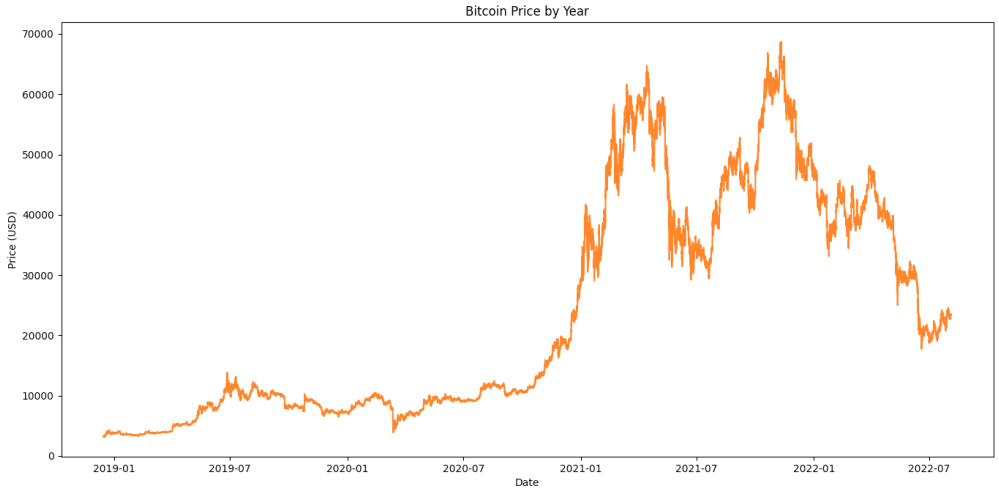

Figure 4.11: Bitcoin price data

4.0.7 Genetic Algorithm Performance

We would expect to see the performance of strategies continually increase over the course of the generations to indicate that a local optimum has been escaped. We can see in Figure 4.12 that this is indeed the case as the fitness function continued to progress towards the top right quadrant throughout the generations. This is a positive result as it indicates that algorithmic trading strategy generation is a suitable candidate for GA for Bitcoin. Although only tested on Bitcoin, there is conceptually no reason to believe that this approach would not generalise to other asset classes (both in cryptocurrency and traditional financial markets) as the technical indicators which are fundamental to the study are generic (and obviously originated in traditional markets). With further research, this program could be extended to fine-tune the algorithm further and provide more comparisons across asset classes and timeframes.

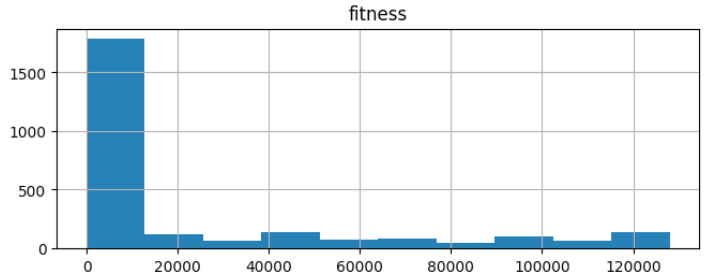

Figure 4.12: Fitness Histogram. The fitness function used was profit-based, and so the Y-axis represents USD.

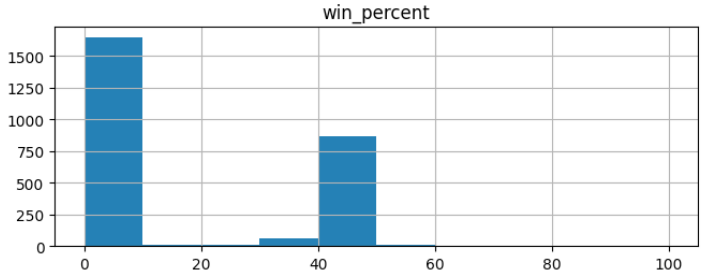

Figure 4.13: Win Percent Histogram.

Figures 4.14 and 4.15 show the summary results from the first generation and the last. Note that every one of the positions is higher in the final generation, demonstrating that the GA successfully evolved the population of strategies towards improved fitness.

```
ipdb> df = top_15[0]
ipdb> df
                                                 strategy   fitness
0   {'id': 'd7e15c15-5a2a-4315-9506-642507891131',...  108067.5
1   {'id': '9f85df57-295a-455a-be73-6a5abd48b675',...  100725.0
2   {'id': '45c5bc41-fabd-4e0c-b558-9241f95e24d0',...   96967.5
3   {'id': '50e95db4-5dd3-4c27-ad76-bac720478b00',...   93232.5
4   {'id': 'e149bf9f-5f16-41ce-8d30-090f586de0d0',...   85845.0
5   {'id': '67046e20-05d5-436f-81e1-dc9ca0df30c2',...   82672.5
6   {'id': 'b6104e22-3e53-47b9-b192-c86e0d0e2eca',...   79080.0
7   {'id': '84e9980d-02ef-4614-b333-b06cedb636fa',...   68835.0
8   {'id': '75ff3884-aff5-4585-9b72-96c931eae2aa',...   56070.0
9   {'id': '1bb9f7da-cfa1-46a4-be0b-3bb07b77559b',...   56047.5
10  {'id': '52ab941b-0e03-4b9a-9631-53261b4595cb',...   51705.0
11  {'id': '120af9c0-3e5b-42a3-8a52-f84e9dae44a9',...   50677.5
12  {'id': 'deae3ce8-9027-4ee7-a0a9-7092c20bca20',...   44767.5
13  {'id': '80cc4d33-77fc-4795-8bcb-b3affcdfe23e',...   31860.0
14  {'id': '62bed9f1-0915-4cd9-8ac2-c1a708f0858e',...   31290.0
```

Figure 4.14: Top 15 Strategies after generation 1

	strategy	fitness
0	{'id': '8835d994-1070-4ae7-b91e-1cbbd6911c5e',...	128002.5
1	{'id': 'ec7b8b8a-a1f2-4902-beb7-fa8af331d6b3',...	128002.5
2	{'id': '2f4ada80-5471-4798-ae1f-0df246e0b89f',...	128002.5
3	{'id': '5bbd6d04-ac82-4bfe-a1e5-52a089c30211',...	127342.5
4	{'id': '1195c061-430d-4760-8d8a-641b1e6f7558',...	123180.0
5	{'id': 'f1c0f3ca-c9ec-4f51-8b21-02a2b0415c65',...	123180.0
6	{'id': '9fffdc82-7a7c-4919-ae15-36a667d281ed',...	123180.0
7	{'id': '4a9c1e81-5889-4404-89c9-c92538678341',...	97830.0
8	{'id': 'e9e004fa-f500-448e-860e-814c4f40f116',...	95107.5
9	{'id': 'ce8e1e73-61f5-4220-b974-9560d0a4beda',...	95107.5
10	{'id': 'a5abcde3-ac01-41c0-86c9-2bd9d8250596',...	95107.5
11	{'id': 'a14ae271-39af-4dcb-b51c-40fb5f3c559f',...	95107.5
12	{'id': '36730442-5738-4a73-a4a2-62741cff5114',...	92917.5
13	{'id': '17873564-70f5-423f-876a-adcde43aea58',...	60090.0
14	{'id': '19e0c701-f2a4-47ac-b4bb-82db76a0eb4e',...	60090.0

Figure 4.15: Top 15 Strategies after generation 50

4.0.8 Trading Strategy Performance

In total 2500 strategies were tested in the GA, via 50 generations of 50 chromosome populations. Of these, many were rejected by the GA as they generated $\geq 33\%$ entry signals or conversely offered no entry points at all. The best 15 performing strategies from each of the 50 generations have been isolated and assessed. Due to the fact that the strategies are randomly generated, there is no reason to assess the performance of the entire set, as it is expected in any GA that a large percentage will perform very poorly. For this reason, statistical measures of the strategies' performance such as precision, recall and F1 score are also omitted. This is illustrated below in figures 4.12-4.13 in which we see a strong grouping in the histograms around 0. The algorithm simply identifies entry points based on a given strategy and then assesses whether these entry points returned a profit or loss. There is no predictive element to the system therefore notions of true/false positives would be ill-fitting to this context. Instead, the relative performance of each of the top 15 strategies emerging from each generation is evaluated using a financial measure such as cumulative gains and annualised gains and compared to the buy and hold strategy.

Selecting the top 15 strategies from each generation gives the 750 best-performing strategies, of which 410 are duplicates (i.e. via elitism). Of these 340 unique strategies, 176 beat the buy and hold strategy for the period (although some only marginally so, and therefore would be worse in real terms when considering trading fees). The top performing best was (with a recorded profit of 128002.5):

```
volume_obv > volume_obv_previous and
(momentum_roc > 0.0 or volatility_bbm > trend_sma_slow and
trend_ema_slow > trend_sma_slow)
```

Due to the complexity of the strategies, a full analysis of strategies was not possible. However, a word cloud was used to visualise the most commonly occurring indicators, which illustrated the high proportion of volatility and volume-based indicators present in the top-performing strategies.

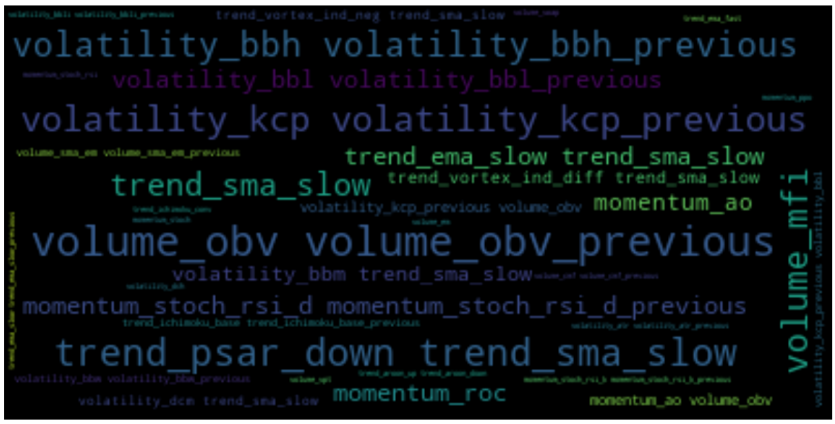

Figure 4.16: Word Cloud of Parsed Strategies

Appendix E provides a screen shot of the top summary results.

A word of caution on the profits returned by the strategies. The mechanism used for trading was fairly crude in that at each was for $1000 USD regardless of market conditions. As the stop loss target was set as 0.0075%, and the account started with $10000 USD, the strategy would have to lose $10000/7.5 = 1,333$ times for every win in order for the performance to register as ≤ 0. Additionally, the results do not consider ancillary trading costs such as fees and slippage, these are key components to the profitability of a trading strategy and can be challenging to calculate accurately due to external factors in the exchange [37].

Chapter 5

Conclusion

5.0.1 Research Objectives

1. **Explore if technical indicator-based trading is a viable option for algorithmic trading of Bitcoin.**

   The research was cautiously optimistic that this object was satisfied. The strategies intended did outperform the buy and hold strategy, and there were no clear indications that successful strategies were constrained to favourable market conditions, (e.g. an up trend) hints at the greater transferability of the TI-based approach over DNN methods. However, whilst the results are suggestive of the generalisability of technical indicator-led strategies, the trading scenarios followed in this study were artificial as they didn't assess how much to invest on a given trade but rather treated all entry signals as equal. In this sense the trading component was a blunt instrument but they do serve to prove the concept that technical indicators can not be ruled out. Finally, the alarming positivity of the results should be taken with a pinch of salt and hints that there could be a calculation error that would require further investigation.

2. **Determine if genetic algorithms offer a means to the discovery of novel trading strategies.**

   Whilst we have to be cautious about the suitability of TI-based strategies, the experimental results show strong suitability for the application of genetic algorithms in this domain. We can conclude that the GA was successful as the results showed an improvement in the fitness function of the top strategies in every generation. Not only this but the lesser strategies also showed signs of improvement compared to previous generations. Although 50 generations of 50 populations were run in the final genetic algorithm, this is too shallow to say with certainty that the genetic process reached a global maximum. Running the algorithm across greater generations would allow for a greater evolutionary process.

3. **Produce a fully functioning end-to-end genetic algorithm software system for this specific context.**

   This goal was achieved as the final version of the code was containerised into a Docker application and deployed and run in AWS via infrastructure as code tools. This means that without significant uplift the service could be horizontally scaled to run on multiple machines

with greater parallelism. Additionally, the underlying Python codebase works as a standalone Python package and embeds many of the interesting implementation challenges such as multiprocessing, modularity and extensibility.

5.0.2 Ethical considerations

There are no specific ethical considerations with this study as there were no human participants. However, given the potential financial implications of the results consideration does need to be given to how the results might be interpreted or used if shared or made public. The research itself is agnostic to any particular type of strategy generated and does not seek to make recommendations. As discussed in the introduction, an ancillary concern surrounding high-frequency trading is the potential triggering of 'flash crashes'. Therefore, it is of importance that algorithmic trading systems are tested thoroughly and with rigour before releasing which is achieved in this study through the use of 4 years of back testing data.

The experimentation intended to follow the scientific measure with rigour and discipline. All bias was removed by only back testing the strategies, therefore there was no financial gain or loss at stake. Incremental runs of the algorithm were made to assess and improve performance however, at no point was the algorithm or data altered to present results in a favourable light.

5.0.3 Recommendations

The following limitations were not addressed within this study and are recommended areas for future research:

As previously mentioned, the algorithm always traded $1000 at a time. A more sophisticated approach should assess the strength of the signal and use this as an additional data point with regard to the trading amount. This would give more realistic portfolio results.

Additional investigation into whether certain combinations of indicators led to better performing strategies would be valuable to look for patterns and correlations (particularly in differing market conditions). Running the GA for a greater number of iterations would also support this to ensure that the process had escaped local optima.

The research could be expanded to include multiple asset classes (both cryptocurrency and non-cryptocurrency) as well as assess different time frames for trading (i.e. not just the 15-minute time frame).

Greater consideration should be given to market conditions to attempt to establish correlations between certain market conditions and the performance of specific strategies. As identified in the research and experienced in this study, there is a lack of formalisation for assessing and reporting of algorithmic trading results in academia. Further work to attempt to develop an academic standard would be a vital contribution to future research.

Finally, the architecture could be expanded to include horizontal scaling using the designs provided but not implemented in this study.

Bibliography

[1] a crypto wallet shows an investor made an $8,000 shiba inu coin purchase last year. today, it is worth $5.7 billion, 2022.

[2] amazon ec2 r6i instances – compute –amazon web services, 2022.

[3] concurrent.futures — launching parallel tasks — python 3.10.5 documentation, 2022.

[4] Data model — python 3.10.6, 2022.

[5] functools — higher-order functions and operations on callable objects — python 3.10.5, 2022.

[6] Globalinterpreterlock, 2022.

[7] Technical indicator definition 2022, 2022.

[8] Khalid Abouloula, Brahim EL Habil, and Salah-ddine Krit. Money management limits to trade by robot trader for automatic trading. *International Journal of Engineering, Science and Mathematics*, 7(3):195–206, 2018.

[9] Dino Arnaut and Damir Bećirović. The potential for forecasting cryptocurrency price movements using the lstm method. In *E-business technologies conference proceedings*, volume 1, pages 120–124, 2021.

[10] Wolfgang Banzhaf, Randal S. Olson, William Tozier, and Rick Riolo. *Genetic programming theory and practice XV [electronic resource]*. Genetic and evolutionary computation series. Cham, Switzerland, 2018.

[11] Randy Bias. The history of pets vs cattle and how to use the analogy properly, 2022.

[12] Rogério M Branco, Antônio S Coelho, and Sérgio F Mayerle. Hybrid genetic algorithms: solutions in realistic dynamic and setup dependent job-shop scheduling problems. *International Journal of Production Management and Engineering*, 4(2):75–85, 2016.

[13] Marco Dorigo and Marco Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial intelligence*, 71(2):321–370, 1994.

[14] Fan Fang, Carmine Ventre, Michail Basios, Leslie Kanthan, David Martinez-Rego, Fan Wu, and Lingbo Li. Cryptocurrency trading: a comprehensive survey. *Financial Innovation*, 8(1):1–59, 2022.

[15] David Farley. *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley Professional, 2021.

[16] Jason Fernando. Relative strength index (rsi) indicator explained with formula, 2022.

[17] Manuel Fogue, Julio Sanguesa, Francisco Martinez, and Johann Marquez-Barja. Improving roadside unit deployment in vehicular networks by exploiting genetic algorithms. *Applied Sciences*, 8:86, 01 2018.

[18] Marta Gasparin and Christophe Schinckus. The performativity of algorithmic trading: The epistemology of flash crashes. *Knowledge Cultures*, 10(1):104–122, 2022.

[19] Dirk F. Gerritsen, Elie Bouri, Ehsan Ramezanifar, and David Roubaud. The profitability of technical trading rules in the bitcoin market. *Finance Research Letters*, 34:101263, 2020.

[20] Sungjoo Ha and Byung-Ro Moon. Finding attractive technical patterns in cryptocurrency markets. *Memetic Computing*, 10(3):301–306, 2018.

[21] John H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992.

[22] Jing-Zhi Huang, William Huang, and Jun Ni. Predicting bitcoin returns using high-dimensional technical indicators. *The Journal of Finance and Data Science*, 5(3):140–155, 2019.

[23] Johannes Rude Jensen, Victor von Wachter, and Omri Ross. An introduction to decentralized finance (defi). *Complex Systems Informatics and Modeling Quarterly*, (26):46–54, 2021.

[24] Suhwan Ji, Jongmin Kim, and Hyeonseung Im. A comparative study of bitcoin price prediction using deep learning. *Mathematics*, 7(10):898, 2019.

[25] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126, 2021.

[26] Massimiliano Kaucic. Investment using evolutionary learning methods and technical rules. *European Journal of Operational Research*, 207(3):1717–1727, 2010.

[27] Timothy King and Dimitrios Koutmos. Herding and feedback trading in cryptocurrency markets. *Annals of Operations Research*, 300(1):79–96, 2021.

[28] Robert L Kissell. *Algorithmic trading methods: Applications using advanced statistics, optimization, and machine learning techniques*. Academic Press, 2020.

[29] Salim Lahmiri and Stelios Bekiros. Deep learning forecasting in cryptocurrency high-frequency trading. *Cognitive Computation*, 13(2):485–487, 2021.

[30] Raymond ST Lee. Cosmos trader–chaotic neuro-oscillatory multiagent financial prediction and trading system. *The Journal of Finance and Data Science*, 5(2):61–82, 2019.

[31] Yang Li, Wanshan Zheng, and Zibin Zheng. Deep robust reinforcement learning for practical algorithmic trading. *IEEE Access*, 7:108014–108022, 2019.

[32] Alexander Loginov and Malcolm I Heywood. On evolving multi-agent fx traders. In *European conference on the applications of evolutionary computation*, pages 203–214. Springer, 2014.

[33] Dave McKenney and Tony White. Stock trading strategy creation using gp on gpu. *Soft Computing*, 16(2):247–259, 2012.

[34] Darío López Padial. Ta. https://github.com/bukosabino/ta, 2021.

[35] Emmanuel Pintelas, Ioannis E Livieris, Stavros Stavroyiannis, Theodore Kotsilieris, and Panagiotis Pintelas. Investigating the problem of cryptocurrency price prediction: a deep learning approach. In *IFIP International conference on artificial intelligence applications and innovations*, pages 99–110. Springer, 2020.

[36] Gautam B Singh. A deep learning architecture for intelligently trading cryptocurrencies. *Name Page No. 1. Establishing Sustainability Context: First Step for Businesses on their Sustainability Journey*, page 91.

[37] Thibaut Théate and Damien Ernst. An application of deep reinforcement learning to algorithmic trading. *Expert Systems with Applications*, 173:114632, 2021.

[38] Yun Wan and Xiaoguang Yang. An empirical study of the self-fulfilling prophecy effect in chinese stock market. *The Journal of Finance and Data Science*, 5:116–125, 2019.

[39] Jim Woodcock and Jim Davies. Using z: Specification refinement and proof. 1996.

[40] Chuheng Zhang, Yuanqi Li, Xi Chen, Yifei Jin, Pingzhong Tang, and Jian Li. Doubleensemble: A new ensemble method based on sample reweighting and feature selection for financial data analysis. In *2020 IEEE International Conference on Data Mining (ICDM)*, pages 781–790. IEEE, 2020.

Note all source code and data is available upon request from https://github.com/mdmjsh/gen_trade.

Appendix A

absolute_signals.json - Signals used as inputs for strategy generation with absolute values


```
[
    {
        "indicator": "volume_cmf",
        "type": "volume",
        "name": "chaikin-money-flow",
        "absolute": true,
        "op": ">=",
        "abs_value": 0.7,
        "rel_value": null
    },
    {
        "indicator": "volume_cmf",
        "type": "volume",
        "name": "chaikin-money-flow",
        "absolute": true,
        "op": "<=",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "volume_mfi",
        "type": "volume",
        "name": "Money Flow Index",
        "absolute": true,
        "op": ">=",
```

```
        "abs_value": 80,
        "rel_value": null
    },
    {
        "indicator": "volume_mfi",
        "type": "volume",
        "name": "Money Flow Index",
        "absolute": true,
        "op": "<=",
        "abs_value": 20,
        "rel_value": null
    },
    {
        "indicator": "volume_mfi",
        "type": "volume",
        "name": "Money Flow Index",
        "absolute": true,
        "op": ">=",
        "abs_value": 90,
        "rel_value": null
    },
    {
        "indicator": "volume_mfi",
        "type": "volume",
        "name": "Money Flow Index",
        "absolute": true,
        "op": "<=",
        "abs_value": 10,
        "rel_value": null
    },
    {
        "indicator": "trend_trix",
        "type": "trend",
        "name": "trend_trix_gt",
        "absolute": true,
        "op": "<",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "trend_trix",
        "type": "trend",
        "name": "trend_trix_gt",
        "absolute": true,
        "op": ">",
        "abs_value": 0,
```

```
        "rel_value": null
    },
    {
        "indicator": "trend_mass_index",
        "type": "trend",
        "name": "trend_mass_index_gt",
        "absolute": true,
        "op": "<=",
        "abs_value": 27,
        "rel_value": null
    },
    {
        "indicator": "trend_stc",
        "type": "trend",
        "name": "trend_stc_gt",
        "absolute": true,
        "op": ">=",
        "abs_value": 75,
        "rel_value": null
    },
    {
        "indicator": "trend_stc",
        "type": "trend",
        "name": "trend_stc_gt",
        "absolute": true,
        "op": "<=",
        "abs_value": 25,
        "rel_value": null
    },
    {
        "indicator": "trend_adx",
        "type": "trend",
        "name": "trend_adx_gt",
        "absolute": true,
        "op": ">",
        "abs_value": 20,
        "rel_value": "MA"
    },
    {
        "indicator": "trend_adx",
        "type": "trend",
        "name": "trend_adx_gt",
        "absolute": true,
        "op": "<",
        "abs_value": 20,
        "rel_value": "MA"
```

```
},
{
    "indicator": "trend_cci",
    "type": "trend",
    "name": "trend_cci_gt",
    "absolute": true,
    "op": ">",
    "abs_value": 100,
    "rel_value": null
},
{
    "indicator": "trend_cci",
    "type": "trend",
    "name": "trend_cci_gt",
    "absolute": true,
    "op": "<",
    "abs_value": -100,
    "rel_value": null
},
{
    "indicator": "momentum_rsi",
    "type": "momentum",
    "name": "momentum_rsi_gt",
    "absolute": true,
    "op": ">=",
    "abs_value": 70,
    "rel_value": null
},
{
    "indicator": "momentum_rsi",
    "type": "momentum",
    "name": "momentum_rsi_gt",
    "absolute": true,
    "op": "<=",
    "abs_value": 30,
    "rel_value": null
},
{
    "indicator": "momentum_rsi",
    "type": "momentum",
    "name": "momentum_rsi_gt",
    "absolute": true,
    "op": ">=",
    "abs_value": 80,
    "rel_value": null
},
```

```
{
    "indicator": "momentum_rsi",
    "type": "momentum",
    "name": "momentum_rsi_lt",
    "absolute": true,
    "op": "<=",
    "abs_value": 20,
    "rel_value": null
},
{
    "indicator": "momentum_stoch_rsi",
    "type": "momentum",
    "name": "momentum_stoch_rsi_gt",
    "absolute": true,
    "op": ">=",
    "abs_value": 0.8,
    "rel_value": null
},
{
    "indicator": "momentum_stoch_rsi",
    "type": "momentum",
    "name": "momentum_stoch_rsi_lt",
    "absolute": true,
    "op": "<=",
    "abs_value": 0.2,
    "rel_value": null
},
{
    "indicator": "momentum_tsi",
    "type": "momentum",
    "name": "momentum_tsi_gt",
    "absolute": true,
    "op": ">=",
    "abs_value": 30,
    "rel_value": null
},
{
    "indicator": "momentum_tsi",
    "type": "momentum",
    "name": "momentum_tsi_gt",
    "absolute": true,
    "op": ">=",
    "abs_value": 20,
    "rel_value": null
},
{
```

```
        "indicator": "momentum_tsi",
        "type": "momentum",
        "name": "momentum_tsi_lt",
        "absolute": true,
        "op": "<=",
        "abs_value": -30,
        "rel_value": null
    },
    {
        "indicator": "momentum_tsi",
        "type": "momentum",
        "name": "momentum_tsi_lt",
        "absolute": true,
        "op": "<=",
        "abs_value": -20,
        "rel_value": null
    },
    {
        "indicator": "momentum_uo",
        "type": "momentum",
        "name": "momentum_uo_gt",
        "absolute": true,
        "op": ">=",
        "abs_value": 70,
        "rel_value": null
    },
    {
        "indicator": "momentum_uo",
        "type": "momentum",
        "name": "momentum_uo_gt",
        "absolute": true,
        "op": "<=",
        "abs_value": 30,
        "rel_value": null
    },
    {
        "indicator": "momentum_stoch",
        "type": "momentum",
        "name": "momentum_stoch_gt",
        "absolute": true,
        "op": ">=",
        "abs_value": 80,
        "rel_value": null
    },
    {
        "indicator": "momentum_stoch",
```

```
        "type": "momentum",
        "name": "momentum_stoch_lt",
        "absolute": true,
        "op": "<=",
        "abs_value": 20,
        "rel_value": null
    },
    {
        "indicator": "momentum_wr",
        "type": "momentum",
        "name": "momentum_wr_gt",
        "absolute": true,
        "op": ">",
        "abs_value": -20,
        "rel_value": null
    },
    {
        "indicator": "momentum_wr",
        "type": "momentum",
        "name": "momentum_wr_lt",
        "absolute": true,
        "op": "<=",
        "abs_value": -80,
        "rel_value": null
    },
    {
        "indicator": "momentum_ao",
        "type": "momentum",
        "name": "momentum_ao_lt",
        "absolute": true,
        "op": "<",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "momentum_roc",
        "type": "momentum",
        "name": "momentum_roc_gt",
        "absolute": true,
        "op": ">",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "momentum_roc",
        "type": "momentum",
```

```
        "name": "momentum_roc_lt",
        "absolute": true,
        "op": "<",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "momentum_ppo",
        "type": "momentum",
        "name": "momentum_ppo_gt",
        "absolute": true,
        "op": ">",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "momentum_ppo",
        "type": "momentum",
        "name": "momentum_ppo_lt",
        "absolute": true,
        "op": "<",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "momentum_pvo",
        "type": "momentum",
        "name": "momentum_pvo_gt",
        "absolute": true,
        "op": ">",
        "abs_value": 0,
        "rel_value": null
    },
    {
        "indicator": "momentum_pvo",
        "type": "momentum",
        "name": "momentum_pvo_lt",
        "absolute": true,
        "op": "<",
        "abs_value": 0,
        "rel_value": null
    }
]
```

Appendix B

relative_signals.json - snippet of programmatically created relative signals. For brevity, a small sample is provided.


```
[
    {
        "indicator": "volume",
        "type": "volume",
        "name": "volume_gt",
        "absolute": false,
        "op": ">",
        "abs_value": null,
        "rel_value": "PREVIOUS_PERIOD"
    },
    {
        "indicator": "volume",
        "type": "volume",
        "name": "volume_gt",
        "absolute": false,
        "op": "<",
        "abs_value": null,
        "rel_value": "PREVIOUS_PERIOD"
    },
    {
        "indicator": "volume",
        "type": "volume",
        "name": "volume_gt",
        "absolute": false,
```

```
        "op": ">",
        "abs_value": null,
        "rel_value": "MA"
    },
    {
        "indicator": "volume",
        "type": "volume",
        "name": "volume_gt",
        "absolute": false,
        "op": "<",
        "abs_value": null,
        "rel_value": "MA"
    }
]
```

Appendix C

Command line interface menu for genetic.py

```
poetry run python genetic.py --help
usage: Main genetic algorithm. [-h] [--population_size POPULATION_SIZE]
[--max_indicators MAX_INDICATORS]
[--max_same_class MAX_SAME_CLASS]
[--write_s3 WRITE_S3]
[--write_local WRITE_LOCAL]
[--s3_bucket S3_BUCKET]
[--generations GENERATIONS]
[--serial_debug SERIAL_DEBUG]
[--strategies_path STRATEGIES_PATH]
[--fitness_function FITNESS_FUNCTION]
[--output_path OUTPUT_PATH]
[--incremental_saves INCREMENTAL_SAVES]

options:
  -h, --help            show this help message and exit
  --population_size POPULATION_SIZE
                        number of strategies to generate
  --max_indicators MAX_INDICATORS
                        max number of indicators in a strategy
  --max_same_class MAX_SAME_CLASS
                        max number of same class indicators in a strategy
  --write_s3 WRITE_S3   exports data to s3.
  --write_local WRITE_LOCAL
                        exports data to local FS.
  --s3_bucket S3_BUCKET
                        bucket name to which data is written.
  --generations GENERATIONS
                        N generations to run.
```

```
--serial_debug SERIAL_DEBUG
                      run without async Future - for debugging
--strategies_path STRATEGIES_PATH
                      load strategies from this path rather than generating on the fly
--fitness_function FITNESS_FUNCTION
                      fitness function use (h=ha_and_moon, o=original, p=profit)
--output_path OUTPUT_PATH
                      path to save outputs
--incremental_saves INCREMENTAL_SAVES
                      when true, saves the output for every 10 strategies tested
```

Appendix D

user_data.sh - AWS EC2 user data script used to run dockerised application

```
sudo yum update -y
sudo yum install -y jq
sudo amazon-linux-extras install -y docker
sudo service docker start
sudo usermod -a -G docker ec2-user
sudo chkconfig docker on
aws ecr get-login-password --region eu-west-1 | sudo docker login --username AWS
--password-stdin "<REDACTED_ID>.dkr.ecr.eu-west-1.amazonaws.com"
sudo docker pull "<REDACTED_ID>.dkr.ecr.eu-west-1.amazonaws.com/ga-repo:latest"

docker run -d -e PARALLEL=1 \
-e BUCKET=ga-44e2849d-f075 \
-e POPULATION_SIZE=50 \
-e GENERATIONS=50 \
-e INCREMENTAL_SAVES=1 \
-e PYTHONUNBUFFERED=1 \
--rm "<REDACTED_ID>.dkr.ecr.eu-west-1.amazonaws.com/ga-repo:latest"
```

Appendix E

Summary Results- all strategies that outperformed buy and hold

	parsed_strategy	fitness (profit)
2	volume_obv > volume_obv_previous and (momentum_roc > 0.0 or volatility_bbm > trend_sma_slow and trend_ema_slow > trend_sma_slow)	128002.5
3	volume_obv > volume_obv_previous and (momentum_roc > 0.0 or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and trend_ema_slow > trend_sma_slow)	127342.5
4	volume_obv > volume_obv_previous and (trend_psar_down > trend_sma_slow or trend_vortex_ind_diff < trend_sma_slow and momentum_ao < 0.0)	124785
5	volume_obv > volume_obv_previous and (trend_psar_down > trend_sma_slow or (volume_mfi <= 11.754279275397126 or momentum_ao < 0.0))	124777.5
6	volume_obv > volume_obv_previous and (trend_psar_down > trend_sma_slow or volatility_bbm > trend_sma_slow and trend_ema_slow > trend_sma_slow)	123180
7	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or (volume_mfi <= 17.860953600000006 or momentum_ao < 0.0))	120607.5
8	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or (volume_mfi <= 16.537920000000003 or momentum_ao < 0.0))	120585
9	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or (volume_mfi <= 16.610686848000004 or momentum_ao < 0.0))	120585
10	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or trend_vortex_ind_diff < trend_sma_slow and momentum_ao < 0.0)	120570
11	volume_obv > volume_obv_previous and (momentum_roc > 0.0 or (volume_mfi <= 13.557856794832896 or volatility_kcp < volatility_kcp_previous))	119550
12	volume_obv > volume_obv_previous and (momentum_roc > 0.0 or volatility_bbm > trend_sma_slow and volatility_kcp < volatility_kcp_previous)	117307.5
13	volume_obv > volume_obv_previous and momentum_roc > 0.0 and (trend_vortex_ind_diff < trend_sma_slow or volatility_kcp < volatility_kcp_previous)	117082.5
14	volume_sma_em > volume_sma_em_previous and (momentum_roc > 0.0 or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and trend_aroon_up > trend_aroon_down)	116130
15	volume_sma_em > volume_sma_em_previous and (trend_ema_slow < trend_ema_slow_previous or (momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous or trend_vortex_ind_neg > trend_sma_slow))	115582.5
16	volume_sma_em > volume_sma_em_previous and (momentum_roc > 0 or (volume_mfi <= 21.200000000000003 or trend_vortex_ind_neg > trend_sma_slow))	114390
17	volume_obv > volume_obv_previous and (volatility_ui > volatility_ui_previous or volume_vwap < trend_sma_slow and volatility_atr < volatility_atr_previous)	111652.5
18	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or volatility_dcm < trend_sma_slow and trend_aroon_up > trend_aroon_down)	108457.5
19	volume_sma_em > volume_sma_em_previous and (momentum_roc > 0 or volume_mfi <= 20 and trend_ema_slow > trend_sma_slow)	108067.5
20	volatility_bbl > volatility_bbl_previous and (volatility_dch < trend_sma_slow or (volume_mfi <= 15.142119552000004 or trend_ema_slow > trend_sma_slow))	107602.5
21	volatility_bbl > volatility_bbl_previous and (volatility_dch < trend_sma_slow or (volume_mfi <= 13.779328792320005 or trend_ema_slow > trend_sma_slow))	107572.5
22	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or trend_vortex_ind_diff < trend_sma_slow and volatility_kcp < volatility_kcp_previous)	107467.5
23	volume_obv > volume_obv_previous and (trend_psar_down > trend_sma_slow or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and trend_ema_slow > trend_sma_slow)	106972.5
24	volume_sma_em > volume_sma_em_previous and (trend_ema_slow < trend_ema_slow_previous or trend_vortex_ind_diff < trend_sma_slow and volatility_kcp < volatility_kcp_previous)	106800
25	volatility_bbl > volatility_bbl_previous and (trend_psar_down > trend_sma_slow or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and trend_ema_slow > trend_sma_slow)	105810
26	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or volatility_bbm > trend_sma_slow and volatility_kcp < volatility_kcp_previous)	105405
27	(trend_ichimoku_base > trend_ichimoku_base_previous or momentum_roc > 0.0 and momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and volatility_kcp < volatility_kcp_previous)	105345
28	volume_obv > volume_obv_previous and (volatility_dch < trend_sma_slow or (volatility_bbm > trend_sma_slow or volatility_kcp < volatility_kcp_previous))	105150
29	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and momentum_ao < 0.0)	104190
30	volatility_bbl > volatility_bbl_previous and (volatility_bbh < volatility_bbh_previous or volatility_dcm < trend_sma_slow and volatility_kcp < volatility_kcp_previous)	104077.5
31	volatility_bbl > volatility_bbl_previous and (volatility_bbh < volatility_bbh_previous or trend_vortex_ind_diff < trend_sma_slow and momentum_ao < 0.0)	104002.5
32	volatility_bbl > volatility_bbl_previous and (volatility_bbh < volatility_bbh_previous or (volume_mfi <= 11.365281994752005 or momentum_ao < 0.0))	103995
33	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or volatility_bbm > trend_sma_slow and momentum_ao < 0.0)	103425
34	volatility_bbl > volatility_bbl_previous and (trend_psar_down > trend_sma_slow or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and trend_aroon_up > trend_aroon_down)	102090
35	volume_obv > volume_obv_previous and momentum_roc > 0.0 and (volatility_bbm > trend_sma_slow or momentum_ao < 0.0)	101782.5
36	trend_ichimoku_conv < trend_sma_slow and (trend_kst_sig > trend_sma_slow or (volatility_atr > trend_sma_slow or volatility_atr < volatility_atr_previous))	101422.5
37	volume_sma_em > volume_sma_em_previous and (momentum_stoch_rsi <= 0.17800000000000002 or (volume_mfi <= 20 or trend_ema_slow > trend_sma_slow))	101370
38	volume_sma_em > volume_sma_em_previous and (trend_psar_down > trend_sma_slow or trend_vortex_ind_diff < trend_sma_slow and volatility_kcp < volatility_kcp_previous)	101265
39	trend_adx_pos > trend_adx_neg and (volume_mfi > volume_mfi_previous or volume_vwap < trend_sma_slow and volatility_atr < volatility_atr_previous)	100942.5
40	volume_em > volume_em_previous and (trend_sma_slow < trend_sma_slow_previous or momentum_kama < trend_sma_slow and volume_vwap > volume_vwap_previous)	100725
41	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and volatility_kcp < volatility_kcp_previous)	100522.5
42	volatility_bbl > volatility_bbl_previous and (trend_psar_down > trend_sma_slow or trend_vortex_ind_diff < trend_sma_slow and momentum_ao < 0.0)	100402.5
43	volatility_bbl > volatility_bbl_previous and (trend_psar_down > trend_sma_slow or (volume_mfi <= 13.896194004000002 or momentum_ao < 0.0))	100365
44	volatility_bbl > volatility_bbl_previous and (volatility_bbh < volatility_bbh_previous or volatility_bbm > trend_sma_slow and momentum_ao < 0.0)	99195
45	volatility_bbw < volatility_bbw_previous and (trend_psar_down > trend_sma_slow or volatility_dcm < trend_sma_slow and volatility_kcp < volatility_kcp_previous)	98445
46	volume_obv > volume_obv_previous and (trend_psar_down > trend_sma_slow or volatility_dcm < trend_sma_slow and trend_ema_slow > trend_sma_slow)	98400
47	volatility_bbl > volatility_bbl_previous and (volatility_bbh < volatility_bbh_previous or momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous and momentum_ao < 0.0)	98212.5
48	volatility_bbl > volatility_bbl_previous and momentum_roc > 0.0 and (volatility_bbm > trend_sma_slow or trend_ema_slow > trend_sma_slow)	97830
49	momentum_stoch_rsi_k < momentum_stoch_rsi_k_previous and (momentum_roc > 0.0 or volatility_dcm < trend_sma_slow and volatility_atr < volatility_atr_previous)	97807.5
50	volatility_bbl > volatility_bbl_previous and momentum_roc > 0.0 and (momentum_stoch_rsi_d < momentum_stoch_rsi_d_previous or trend_ema_slow > trend_sma_slow)	97110
51	trend_macd_signal > trend_macd_signal_previous and (trend_psar_down > trend_sma_slow or volatility_dcm < trend_sma_slow)	96967.5
52	trend_ichimoku_conv < trend_sma_slow and (trend_psar_down > trend_sma_slow or (volume_mfi <= 10.085752908 or trend_ema_slow > trend_sma_slow))	96502.5
53	volume_obv > volume_obv_previous and (volatility_dch < trend_sma_slow or (volume_mfi <= 18.932610816000008 or momentum_ao < 0.0))	96060
54	volume_obv > volume_obv_previous and (volatility_dch < trend_sma_slow or (volume_mfi <= 15.142119552000004 or momentum_ao < 0.0))	95947.5
55	trend_ichimoku_conv < trend_sma_slow and (trend_psar_down > trend_sma_slow or trend_vortex_ind_diff < trend_sma_slow and trend_ema_slow > trend_sma_slow)	95760
56	volatility_bbl > volatility_bbl_previous and volatility_bbh < volatility_bbh_previous and (trend_vortex_ind_diff < trend_sma_slow or momentum_ao < 0.0)	95595
57	volatility_bbl > volatility_bbl_previous and volatility_bbh < volatility_bbh_previous and (trend_vortex_ind_diff < trend_sma_slow or trend_ema_slow > trend_sma_slow)	95595
58	volatility_bbl > volatility_bbl_previous and volatility_bbh < volatility_bbh_previous and (trend_vortex_ind_diff < trend_sma_slow or volatility_kcp < volatility_kcp_previous)	95595
59	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or volatility_dcm < trend_sma_slow and volatility_kcp < volatility_kcp_previous)	95542.5
60	volume_obv > volume_obv_previous and (volatility_bbh < volatility_bbh_previous or volume_mfi <= 18.026332800000006 and momentum_ao < 0.0)	95220

Figure E.1: Screenshot of summary results (full raw data available upon request)